Recent Progress in the Development of a Debugging Assistant for Computer Programs

P.A. Smith and G.I. Webb
Deakin University, Geelong, Australia

Abstract

We present recent progress in the development of a debugging assistant for helping novices debug their computer programs. Bradman, which is still in the implementation phase, is an interactive system which builds two models of the user’s program – one reflecting what the program actually does and the other reflecting what the programmer intended to do. Conflicts between these two models are used by Bradman to find bugs in the program.

1 Introduction

Last year we introduced our thoughts on an interactive system to help novices debug their Pascal programs (Smith and Webb 1991). This system, called Bradman, is still in the implementation stage. Here we present on how our ideas have since developed.

Novices generally find learning their first programming language a difficult and frustrating business. They lack the experience to find errors in their programs and often need to seek help from more experienced programmers. A computer- based debugging assistant, available twenty four hours a day, would be of great benefit to people in this situation.

Last year we described in broad outline how Bradman might operate. This year we would like to discuss more fully how Bradman is to model the user’s program in order to find bugs.

2 Errors

There are many ways that errors can occur in programs. They can occur at any stage of program development and manifest themselves in many different ways. Wertz (1982) divided errors into five classes

- Lexical errors
- Syntactic errors
- Semantic errors
- Teleological errors
- Conceptual errors

We will refer to lexical and syntactic errors as compilation errors and the other types as run-time errors. Teleological and conceptual errors will be referred to as logic errors. Errors were classified across a different dimension by Harandi (1983). He defined two types of error

1. Shallow errors - those that can be diagnosed directly from their symptoms. These include mainly compilation errors (for example misplaced parentheses) and some run-time errors (for example infinite loops).
2. Deep errors - those that can not be diagnosed directly from their symptoms. These are mainly logic errors. We will be concentrating on deep errors since they are the most difficult to find. Deep errors occur in programs that compile, run and terminate successfully but give incorrect output. These programs are not faulty in themselves - alter all they solve some programming problem. They are faulty because they give results which differ from those that the programmer intended. In these cases diagnosis requires knowledge not only of how the program behaves but also of how the user intended it to behave.

2.1 Diagnosing Deep Errors

When programmers with some experience find bugs in their programs they draw on their experience of programming to help them conduct a search for the error. They can also make use of tools which provide a more friendly environment for this search. These tools usually perform a trace of the program execution which enables the programmer to step through his program examining the values of certain variables at critical points etc. These tools make no attempt to diagnose the error. The user retains the responsibility to hypothesize about the possible causes of the error and, once the cause has been found, to modify his code appropriately. This requires programming experie nce thus making these tools inappropriate for novices.

When a novice makes an error he often needs to augment his own knowledge with that of an expert (often a human tutor in a teaching environment). A tutor needs to do two things to assist the novice. Firstly he must find the bug and then he must explain, in terms that the novice will understand, how the bug was caused and how it may be remedied.

To find the error the tutor must first gather information about the program. There are several sources of information that he might utilise

- The program code
- A program specification
- Examples of faulty program behaviour
- His own knowledge of programming
- Consultation with the user

It is necessary for the tutor to use this information to determine why the implemented program does not behave as the user expects.

2.2 Modelling the Program

It would seem natural, therefore, that an automated debugging system should compare two models of the program. These would be an intentions model (reflecting what the program is supposed to do) and an implementation model (reflecting what the program actually does). This is not a new concept. PROUST (Johnson, Soloway 1984), Laura (Adam, Laurent 1980) and Pudsy (Lukey 1980) all operate in this way. These systems are given information regarding the user's intention in the form of a program specification. This specification must be given to the system before diagnosis of an individual program can commence. Bradman differs from these systems by constructing a model of the user's intentions while diagnosis is actually taking place without reference to a predefined program specification.

3 Overview of Bradman

Bradman takes the user's syntactically correct program code as input. This code is parsed and relevant information extracted from each statement. This information is stored in a. tree structure and is called the implementation model. It reflects what the program actually does. At this stage, statements are treated as individual entities and no attempt is made to understand their purpose in relation to other statements.

In the next stage, plans from a plan library are mapped onto the implementation model. Individual statements are used as 'triggers' to decide which plans are tested. When several competing plans are possible candidates they are placed in order of likelihood. If a plan is found to be inappropriate and discarded the next in line is tried. If the current plan does not map perfectly onto the code then a discrepancy has been found and this anomaly is used as a starting point in Bradman's investigations.

During this stage, all of the above hypotheses regarding both plans and statements are tentative since they might not reflect the user's intentions. Anomalies that Bradman finds in the code are used as starting points for an interactive discourse with the user aimed at determining the user's intentions. As this dialogue proceeds Bradman builds a model of the user's intentions. Bugs are found at places where the implementation model diverges from the intentions model.

3.1 Example

The following is an example used by Johnson and Soloway (1985) when describing PROUST.

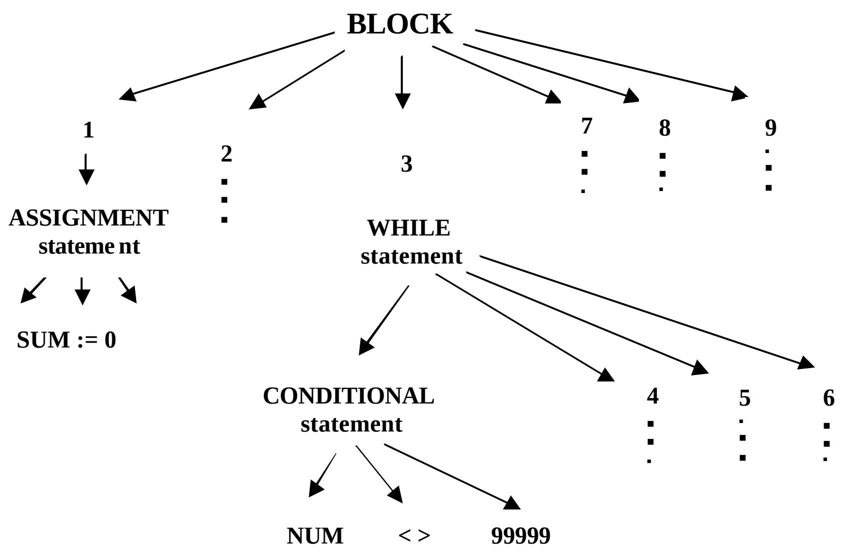
```
Problem: Read in a set of integers and print out their average.
Stop reading numbers when the number 99999 is seen. Do NOT
include 99999 in the average.

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

In this program a value is read from standard input and is processed before being tested against the sentinel, In this case the sentinel value will be included in the calculations.

Bradman parses the program code and creates the implementation model in a tree format. The implementation model might look as shown in the diagram (only some subtrees have been completed in full.)

For example, the third statement is a while statement encountered at line 9. Its conditional expression is 'num <>99999'. It contains a compound statement comprising `99999 -` the fourth, fifth and sixth statements of the program (shown as a sub-tree of the third statement). The other statements contain similar relevant information.



From the implementation model an intermediate list of tentative hypotheses is made about the function of some of the statements some of these hypotheses might be:

- That statement 1 is an initialisation statement.
- That statement 2 is also an initialisation statement.
- That statement 3 is part of a sentinel test plan.
- That statement 5 is intended to sum a series of numbers.
- That statement 6 is intended to count a number of events.

For example, the third statement might be an instantiation of a sentinel test. In other words, it might be intended to read in a series of values and process them and to stop when a specified value is read.

At this stage Bradman can make no further conjecture about the functions of statements 4, 7 and 8. Both the tentative hypotheses and the implementation model are concerned with extracting the functions of individual statements. Interactions between the various code elements are not considered.

Bradman's next task is to use these hypotheses as 'triggers' to match plans to the code. These plans are stereotypical objects designed to recognise certain algorithms and structures. An initialisation plan might map onto an assignment statement which assigns a value to a variable and precedes code that matches a plan that requires initialisation of that variable.

A more complicated plan is one for a sentinel test. A sentinel test requires that an input value be read, tested against a sentinel value and, if it is not the sentinel value, to be processed in some way. There are several ways that such a test might be implemented in the code. For example, it could be implemented using a while loop, a repeat loop, a go to statement, a recursive function etc. Even assuming that a while loop is used there are still different algorithms that might be used.

To overcome this problem Bradman uses a library of plans which cover the various possibilities. Bradman matches these plans against the code and attempts to find the plan that is most likely to have been intended by the user. This is often not a trivial problem when considering faulty code.

At this stage all of Bradman's deductions regarding the programmer's intentions are just conjecture. The purpose of doing the above analysis is to obtain an idea of the programmer's possible intentions. Once this is done Bradman can commence an interactive dialogue with the user to confirm and refine its hypotheses.

With respect to the current example, Bradman concentrates on the sentinel test because, after the plan-matching phase, a discrepancy has been found. None of the plans in the library fit the code perfectly. Bradman first confirms with the user that a sentinel test was indeed intended. It then reports the error and explains methods by which it can be remedied.

3.2 An Example Dialogue

```
1/ Does your program terminate but show incorrect output?
            Y
2/ The while loop at line 9 has conditional statement
            num <> 99999
This condition 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 (often a large integer). Is
            num <> 99999
intended to be a sentinel test?
            Y
3/ Your program will include the sentinel value (99999) in the
calculations. Is this what you intended?
            N
4/ In your program a value is read (at line 10) and then is processed
(at line 11) before being tested against the sentinel value. Hence
the sentinel value will be included in the calculations.
This problem can be solved in two ways
a/ You could duplicate the test so that a sentinel test occurs
between the read statement and the statement that processes the input
value.
b/ You could duplicate the read statement so that one read statement
appears before the sentinel test and the other appears after the
process statements within the scope of the loop.
No Obvious Anomalies
```

In the previous example Bradman was able to detect anomalies in the user's program without needing to construct a very detailed intentions model. However, there are occasions where a program will give incorrect results without showing obvious anomalies. Bradman can still provide assistance in these situations.

Consider the following program.

```
1/ program ave (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 nun <> 99999 do begin
10/               sum := sum * num;
11/             count := count + 1;
12/               read (num);
13/         end;
14/         ave := sum / count;
15/         writeln (ave);
16/ end.
```

In this case as there are no obvious anomalies (we'll assume that Bradman does not have even higher level plans involving averages), Bradman hypothesises about a sentinel test and has a perfectly good plan to cover it. What can Bradman do in this situation?
Bradman starts at certain points in the program. Bradman knows that the program displays incorrect output. There are three possibilities.

1. There is a missing output statement.
2. The variable 'ave' is the wrong variable to output.
3. The variable 'ave' contains the wrong value.

Which of the three possibilities is correct is decided directly by asking the user. If it is one of the first two then the problem is easily fixed. Bradman will not be able to give exact advice. Its contribution is that it has highlighted a problem that has been overlooked by the user.

'ave' contains the wrong value because it has been assigned the wrong value at some stage of the program. Either an essential assignment statement is wrong (or missing) or the variables used to produce the value in the assignment statement are wrong.

Bradman checks the possibility that any existing statements are wrong. There is only one statement affecting the value of 'ave' in the program.

```
ave := sum / count;
```

Bradman explains exactly what the statement does. That is, it divides the value contained in sum by the value contained in count. Is this the user's intention? Yes it is.

If the expression is correct then either 'sum' or 'count' (or both) are incorrect. Bradman chooses one of them to work on. If count is chosen first then subsequent investigations should indicate that it is working correctly.

It is when we get to the value of sum that an error should be found. However, the user may not realise that the statement

```
sum := sum * num;
```

is wrong. In this case, step through execution is the probably the best to continue the investigations.

In this situation, Bradman is far more reliant on the user for correct answers. It is in Bradman's interest, therefore, to be able to find code fragments that will provide most likely causes of errors.

4 Future Directions

We expect to have a working model of the system implemented by the middle of next year. This model will draw information for the programmer's code and the user himself. We will then look at ways of extending Bradman so that it draws information from other sources. For example, it could run the program on an example case arid ask the user what output he expected. Bradman would use this information to help guide its line of enquiry in much the same manner as Teiresias considers case information when debugging expert systems (Davis, Lenat 1982). Also it could be possible to use some form of program specification as a starting point of investigations. This would equate to starting with a full or partially full intentions model before diagnosis begins. These additional optional sources of knowledge could be readily integrated into the general system providing added power when available but not precluding operation when not.

5 Conclusion

Bradman is a debugging system which detects errors by finding discrepancies between a user's implementation of a program and what he intended the program to do. It does this by building models of the implementation and intentions and comparing them. It differs from previous systems by building the intentions model interactively rather than using pre-given program specifications.

References

[1] Adam, A., Laurent, J.P. (1980) LAURA: A System to Debug Student Programs. *Artificial Intelligence* 15. pp 75-122.

[2] Davis, M., Lenat, D.B. (1982) *Knowledge Based Systems in Artificial intelligence*. McGraw-Hill, New York.

[3] Johnson, W.L., Soloway, E.M. (1985) PROUST: An Automatic Debugger for Pascal Programs, BYTE 10, April, pp 179-190.

[4] Lukey, F..J (1980) Understanding and Debugging Programs, *International Journal of Man-Machine Studies,* 22, pp 189-202.

[5] Smith, P.A., Webb, G.I. (1991) Programming using Partial Models, *ASCILITE* 92, pp 581-90

[6] Wertz, H. (1982) Stereotyped program Debugging: an aid of Novice Programmers, *International Journal of Man-Machine Studies* 16, pp 379-392