

# Transparency Debugging with Explanations for Novice Programmers

Philip A Smith and Geoffrey I Webb

*School of Computing and Mathematics, Deakin University, Geelong, Australia*

## Abstract

Novice programmers often find programming to be a difficult and frustrating task. Because of their lack of experience in programming novices have different needs to experts when it comes to debugging assistants. One way a debugging assistant could be tailored to novices, as proposed by Eisenstadt, is to provide them with an explicit model of how their program works and, hence encourage them to find errors for themselves. We discuss such a transparency debugger, Bradman, that we have been developing to assist novice programmers understand and debug their C programs. We also present the results of an experiment, conducted on volunteer novice programmers, in which approximately half of the students had access to an explanation of each statement as it was executed and the other half did not. We show that access to such explanations provided beneficial results for a significant number of students.

## 1 Introduction

The acquisition of expertise in programming, like the acquisition of expertise in other skills, requires both theoretical knowledge and practical experience. One important skill novices need to acquire is how to find and rectify errors in their programs. Both novices and experts encounter bugs in their code. However, with their experience experts are often able to hypothesise about the likely cause of an error and are equipped with skills in the isolation and identification of errors. Frequently novices are unable to make such hypotheses and do not have the necessary debugging skills. As a result, they are unable to proceed with the development of their program. They often need to make up for this shortcoming by consulting someone with the necessary proficiency in programming. Human advice is not always available at times when it is required by the novice programmer and so automated environments which provide such advice have been developed. Further, even when it is available, the advisor will often be inclined to show the novice the solution to their problem rather than to encourage novices to diagnose and repair the problem for themselves. There is potential for automated advisors to overcome some of the deficiencies of human advisors.

An expert programmer generally finds the so-called visibility debugger (Moher, Wilson) to be of great assistance. Such a debugger provides an environment in which various aspects of the program are available for viewing and manipulation by the user. For example, the setting of breakpoints, stepping through individual statements, displaying the value of critical variables etc. These debuggers provide no active assistance to the user in hypothesising about the possible cause of a bug and require the user to have a certain amount of expertise.

A different type of debugger is that which analyses the program code without executing the program (Johnson 1990). This type of debugger normally performs a comparison between an idealised model of the program and the actual program code. Errors are deduced from significant differences between the two. Such debuggers have the capacity to automatically finding bugs which can normally not be found by ordinary debuggers and hence are of more use to novices and, indeed, are often aimed specifically at them.

However, some of the problems associated with static analysers are

- Limitations in the range of problems they are able to address. Before they can analyse a program a program specification must be developed in a form that the analyser can

understand. As the complexity of a programming problem increases the problem of representing it adequately increases.

- False alarms. A false alarm is a piece of correct code which is not recognised by the system and is tagged as an error.
- Inflexibility. One recognised method of developing a program is to build it up piece by piece. That is, one task is coded, tested and debugged before addressing another task. Syntax analysers cannot provide assistance until the full program has been developed.

Eisenstadt, Price & Domingue (1993) also pointed out that such debuggers were locked into a “we know best” situation in which students were forced to follow the methodology indicated by the system rather than explore their own intuitive ideas. They suggested that a system which gives the novice an idea of inner workings of the computer by enhancing visibility might be of more benefit. The Transparent Prolog Machine (Eisenstadt, Brayshaw 1987) was designed to give an explicit graphical picture of a given Prolog program and provides facilities for both novices and experts. It was designed to assist users to find their own bugs rather than automatically finding the bug for them.

## 2 Bradman

We have been developing a debugging assistant, Bradman, which is designed to give assistance to novice programmers in finding un-time errors in their programs. Like Eisenstadt’s machine, Bradman assists the user by giving him/her a visible model of the workings of the program. Bradman does not attempt to build a deep understanding of the purpose of the program. Hence it is unable to automatically locate bugs which are a result purely of a mismatch between the user’s intentions for the program and the implementation thereof. However, it is intended to provide maximal support for inexperienced programmers attempting to diagnose such bugs. Features of Bradman include:

- provision of an explicit model of the internal working of the computer. This model extends previous visibility systems by providing an explicit, visible depiction of the state changes wrought by statement execution.
- explicit, detailed explanation of the effect of each statement as it is executed.
- immediate reporting of errors as they are uncovered, relating them to the context in which they occur with clear description of both the error and its causes,
- an environment in which errors can be remedied.

Bradman takes the user’s source code as input. This code is compiled and converted into an internal representation which takes the form of a syntax tree. During execution, this syntax tree is interpreted expression by expression. As each expression is interpreted Bradman extracts several types of information and makes it available to the user in various windows displayed on the screen. The types of information displayed include:

- The values of all active variables both before and after statement execution. This aids the novice’s understanding of the operation of a computer program by providing an explicit model of how a program changes program states. The user is shown explicitly by name to which variable a pointer variable is directed. This feature is particularly useful for showing novices how passing pointers to functions can cause permanent changes to the value of a variable. If a variable has not been assigned a value by the program then it is described as **as yet unset** rather than having the indeterminate initial value displayed.

- Bradman displays all input/output transactions in a separate window. This information is similar to most visibility debuggers. One added feature is that any buffered input is displayed explicitly.
- If an error is encountered when an expression is interpreted then a new window is displayed which describes the error. When this occurs the user is not able to continue execution. The program must be edited, the error rectified and the debugging process restarted.
- As each expression is interpreted an explanation of what it is and how it affects the program is displayed in a window dedicated to this purpose. The explanations window is a feature that is aimed directly at the novice programmer and is designed to be both a debugging and a teaching aid. It is non-intrusive. The user need not attend to it at any time. Each new explanation is appended to a scrollable text. The user can scroll back through this text obtaining a complete and detailed explanation of all operations that led to the current program state. The user receives immediate feedback about the effect of the executed statements.

## 2.1 Example:

The following is an example of how we might expect Bradman to work. The student is given the following programming problem to solve:

```
Read an integer from the screen
Sum all integers between 1 and this integer
Write the results to the screen
```

The student writes the following program to solve this problem

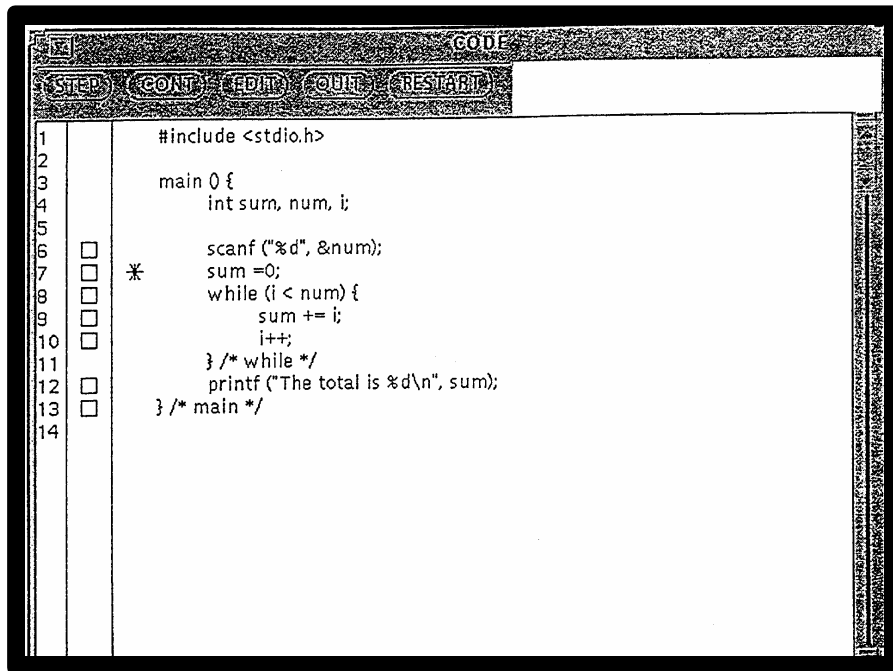
```
#include <stdio.h>
maino(){
    int sum, num, i;

    scanf ("%d", &num);
    sum = 0;
    while (i < num){
        sum + = i;
        i++;
    }/* while*/
    printf ("The total is %d\n", sum);
}/* main*/
```

When this program is run with the input 5 the expected output is 15. However, instead of giving this output the program hangs without terminating. Bradman is invoked to see if it can shed any light on the error.

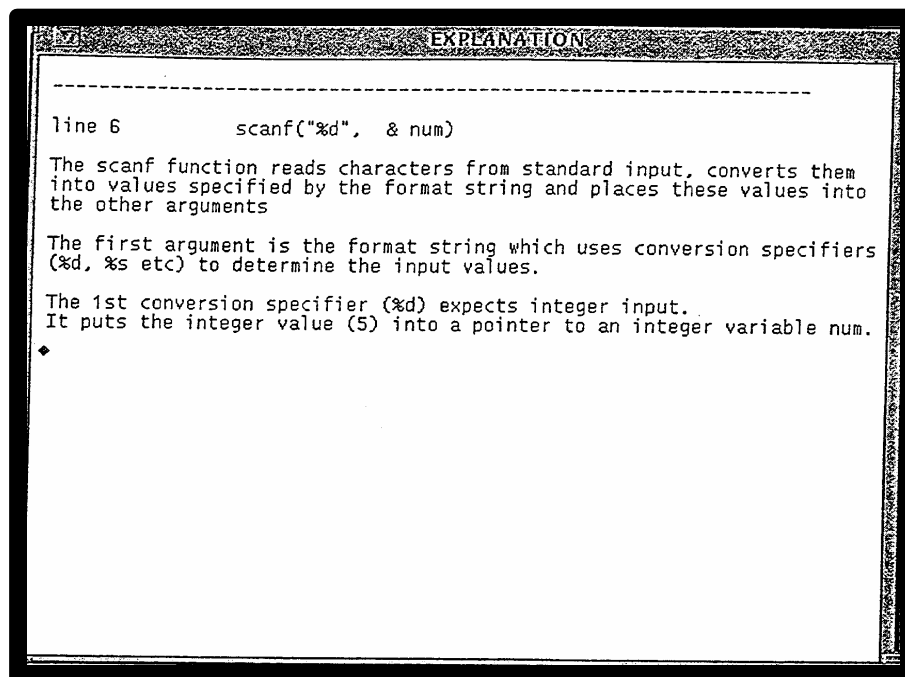
The rate of execution is controlled by the code window. The user clicks on the step button to execute each statement. The statement about to be executed is indicated by the “spider” on the left of the code window. As the user steps through the program information is updated and displayed in the various windows. The user clicks the step button and types 5 in the input/output window.

The Code window looks as follows



```
1 #include <stdio.h>
2
3 main () {
4     int sum, num, i;
5
6     scanf ("%d", &num);
7     * sum = 0;
8     while (i < num) {
9         sum += i;
10        i++;
11    } /* while */
12    printf ("The total is %d\n", sum);
13 } /* main */
14
```

The explanations and variables windows look as follows

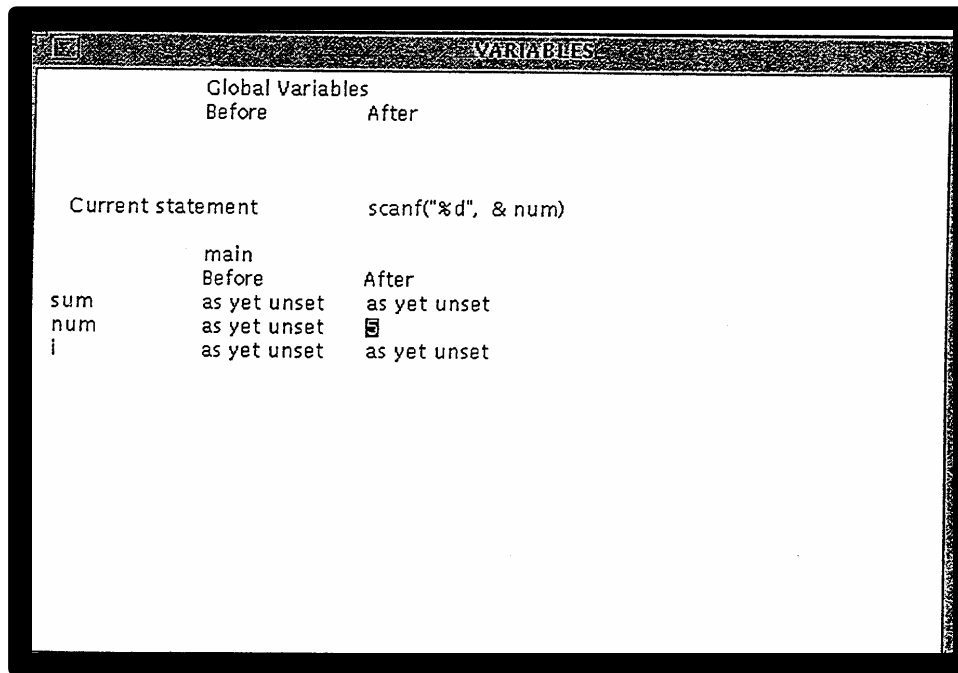


```
-----
line 6      scanf("%d", & num)

The scanf function reads characters from standard input, converts them
into values specified by the format string and places these values into
the other arguments

The first argument is the format string which uses conversion specifiers
(%d, %s etc) to determine the input values.

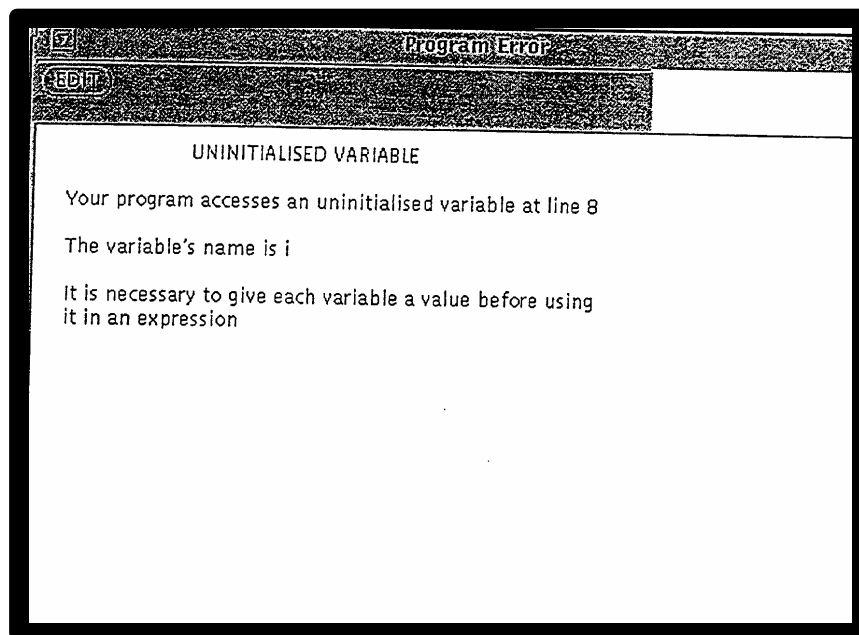
The 1st conversion specifier (%d) expects integer input.
It puts the integer value (5) into a pointer to an integer variable num.
♦
```



The user can continue to step through the program. However, when the statement

```
While (i<num)
```

is reached a fifth window appears. This window gives a description of the error in the current context of the program. The errors window looks as follows



While the user has not been told how to fix the error the search area of possible errors has been narrowed. The user rectifies the error by putting the statement

```
i = 0;
after
sum = 0;
```

The program is tested again. This time the program terminates and output is given. However, the expected output is 15 while the actual output is 10. This time Bradman will find no explicit error and will simply execute each statement until the program terminates. The onus is on the user to observe the information displayed in the explanations window and the variables window to see when a deviation from the expected has occurred.

Eventually, the program will reach a stage where the value of `i` is 5, the value of `sum` is 10, the value of `num` is 5 and the while loop is about to be executed. The explanation window will explain that since `i = 5` and `num = 5` the logical expression `i < num` is false and that the while loop will not be entered. When the user steps, the while loop is skipped. It can be seen that the final 5 is not added to the running total contained in `sum`. The programmer would then need to decide how to alter the program to remedy this mistake.

## 2.2 Experiment:

The explanations in the explanations window are generated from the semantics of the language as the program is executed. Since the explanations window was designed for novices, we conducted an experiment with volunteer first year university students learning C. This experiment was designed to test the effectiveness of the explanations window. To this end we provided two versions of Bradman - one which had the explanation window and one which did not. When students found that they had an error in their program they were encouraged to use Bradman to look for it. When they had finished a session with Bradman they were given a questionnaire with two questions regarding how much assistance Bradman had been to them. Each student was allocated to one of the two experimental treatments when they first invoked Bradman. These allocations were performed on an alternating basis i.e. if one student got version 1 then the next student got version 2 and the following student got version 1 etc. The results of these experiments were collected.

The students were given the problem described earlier and asked to write a program to solve it. They were given assistance with syntax errors and also assistance in basic concepts of C. For example, some needed to be told that a looping construct such as a while loop needed to be used. If they had trouble conceptualising how to do all of the program they were encouraged to build it up piece by piece. Hence they could write a program that simply read a number and wrote it out again. When a program compiled but displayed unexpected behavior when it ran, the student was encouraged to call upon Bradman.

Since we asked for volunteers, the people tested tended to be the weaker students who were having trouble coping with the set C course and were seeking additional tuition. Most of them, after an initial reluctance to learn the new system, warmed to it and used it without prompting. With a couple of exceptions the response was overwhelmingly favourable supporting Eisenstadt's claim that novices appreciate an explicit model of how their program is working.

The first question students were asked after a session with Bradman was designed to test whether the provision of the explanation window was of benefit to novice programmers searching for bugs.

Did Bradman help you find your bug?

64 responses to this question were received from people who had the version of Bradman with the explanation window. These responses were split up as follows

Strongly Agree	19
Agree	20
Neutral	18
Disagree	7
Strongly Disagree	0

60 responses to this question were received from people who had the version of Bradman without the explanation window

Strongly Agree	10
Agree	22
Neutral	18
Disagree	13
Strongly Disagree	4

As can be seen, the students with the explanation window more frequently strongly agreed and less frequently disagreed or strongly disagreed with the proposition. These results were compared statistically using a Mann-Whitney test. The resulting p-value was

$$p = 0.0084$$

which is statistically significant at the 0.005 level. This indicates that the difference in results is genuine and, hence, that the found the system more helpful when it included the explanations window.

The second question was more general and designed to test whether the explanations window assisted the user in increasing their general understanding of the language.

Did Bradman help your understanding of C?

62 responses were received from people who had the version of Bradman with the explanation window. These were split up as follows

Strongly Agree	9
Agree	19
Neutral	18
Disagree	9
Strongly Disagree	5

60 responses were received from people who had the version of Bradman without the explanations window. These were split up as follows

Strongly Agree	8
Agree	19
Neutral	23
Disagree	7
Strongly Disagree	3

While a slight bias still seems to exist in favour of the explanations window it is far less pronounced than that of the first question. Indeed when these results were compared they were found to be not statistically significant. Hence while the provision of an explanations window increased the student's belief in the degree to which Bradman helped them uncover program bugs, it did not affect their evaluation of the system's assistance in teaming the C language.

The above results may have been biased by the fact that Bradman was only called by the students when they had found a bug. Hence some students used the system more often than others and a training effect was possible. That is, familiarity with the system might have assisted in the finding of bugs. On average, students with access to the explanations window used the system 6.6 times while those without access used it 4.6 times.

Examples of where Bradman could be useful were found even at the simplest levels. The following is an example of a student using stepwise refinement. The program is designed to simply read a value into the variable `i` and print it out again.

```

Main () {
    int i;

    printf ("Please enter an integer.\n");
    scanf ("%d",i);
    printf ("%d\n",i);
}

```

The student expected the following transaction

```

Please enter an integer
5
5

```

But got

```

Please enter an integer
5
-268438044

```

This was a problem encountered by several students. The error is that the user has passed the variable `i` to `scanf` instead of a pointer to the variable `i` (that is `&i`). When Bradman executed the program, execution was stopped when the `scanf` statement was reached and an error message describing the problem. Thus the student was given an explanation of what the problem was and how it might be rectified.

### 3 Extensions and Further Research:

As well as assisting the novice by itself another possibility which became apparent during testing of Bradman is the potential of using the system as a teaching aid. In some cases concepts are more easily explained by the human tutor when using Bradman. For instance, it is often difficult to explain to students the concept of passing addresses to functions as parameters in order for the program to produce the side-effect of permanently changing the value of the variables involved. This effect was relatively simple to explain, however when the students were able to see the value of the variables (in the variables window) changing in the calling function. When the variables themselves were passed the student could see the values of the variables change as indirect references were used for assignment. They then saw the variables local to (he called function disappear when the function terminated. The efficacy of Bradman as a teaching tool is something we wish to explore more deeply.

We believe one of the most important innovations incorporated into Bradman is the explicit depiction of program state changes. Future research will evaluate the effectiveness of this feature.

There are two extensions to Bradman that we would like to test on novices. Firstly, *execution-back-tracking* (Agrawal, DeMillo, Spafford 1991), which is a facility to allow a user to move backwards through a program as well as forwards. This obviates the need for a program to re-execute when a user wants to re-examine previously executed code. As proposed by the authors a "what-if" facility, which allows the user to test the effect of changing the value of critical variables, could be used in conjunction with execution-backtracking.

Secondly, *dynamic program-slicing*, (Agrawal, Horgan 1990) which enables the user to extract parts of a program involved in determining the value of a variable at a given stage of execution. We would envisage the user indicating to Bradman which output was faulty. Bradman would then relate this output to the expression involved in the output statement. Slices of the variables involved could be used to simplify the search space for the novice.



We will be interested to see how much benefit novice programmers can get from such facilities.

## 4 Conclusion

Bradman is a system which approaches the question of debugging assistance for novice programmers by providing an environment which increases the user's understanding of how the program works making it easier for him/her to discover the bug by him/herself. Novices lack the skills which enable experts to isolate and rectify errors in their programs. Bradman attempts to compensate for this shortcoming by providing an explicit model of the program and its operations which makes understanding of it easier for novice programmers and by providing active support during the process of acquiring debugging skills. Bradman contains two major innovations. The first is the provision of an explicit representation of state changes during program execution. The second is the provision of explicit detailed explanations of the programs operation. We have demonstrated novices appreciate having such information made explicit and that a facility that explains individual statements supports them in their debugging endeavours.

## BIBLIOGRAPHY

- [1] Anderson, Reiser "The Lisp Tutor", *Byte* April 1985, pp 159-175.
- [2] Johnson, W.L. "Understanding and Debugging Novice Programs", *Artificial Intelligence* 42, 1990, pp 51-97.
- [3] Lukey, F.J. "Understanding and Debugging Programs", *The J Man-Machine Studies* 12 1980, pp 189-202.
- [4] Eisenstadt M., Price, B.A., Domingue J. "Software Visualisation as a Pedagogical Tool", *Instructional Science* 21, 1993, pp 335-364, Kluwer Academic Publishers.
- [5] Eisenstadt, M., Brayshaw, M., "The Transparent Prolog Machine (TPM): An Execution Model and Graphical Debugger for Logic Programming", *HCRL Technical Report* No.21A, Oct 1987.
- [6] Moher, T.G., Wilson, P.R., "Offsetting Human Limits With Debugging Technology", *IEEE Software*, May 1991, pp 11-13.
- [7] Agrawal, H., Horgan, J.R., "Dynamic Program Slicing", *Proceedings of the ACM SIGPLAN '90*, June 1990, pp 246-256.
- [8] Agrawal, H., DeMillo, R.A., Spafford, E.H., "An Execution-Backtracking Approach to Debugging", *IEEE Software*, May 1991, pp 21-26.