And Now For Something Completely Different...

Sue Black[1] and Philipp Bouillon[2]

[1] South Bank University London
[2] FernUniversität in Hagen

Abstract. A pilot experiment was conducted at Dagstuhl using the 'Beyond program slicing' seminar attendees. Attendees were split into three groups: all were given the same program to understand and a list of program comprehension related questions to answer. Group one had only the source code, group two had the source code and the dynamic trace of the program, group three had the source and a control-flow graph of the program.

1 The Experiment

In the course of the Dagstuhl seminar *Beyond Program Slicing*, we conducted an experiment to see which representation of a program helps an individual most in understanding what a program does. We divided the participants of the seminar into three groups and provided them with different representations of the same program. All three groups had access to the source code of the program, group two were additionally given an execution trace of one program run and group three were given a control flow graph.

Figure 1 shows the program that we gave the participants. The program is a slightly obfuscated version of a program which calculates the variance of a number of floats. Since we did not want to make it too easy for people to analyse the program, we changed the variable names so that they no longer reflected the variables purpose. The original, unmodified program can be seen in Figure 2.

The execution trace of one program run is depicted in Figure 3 while the control flow graph handed out can be seen in Figure 4.

In order to see if the program had been understood correctly, we asked the following questions (the correct answer to each question is given in italics):

- What are the most important variables with regard to the purpose of the program? *Knuth, Church, vNeumann, Dijkstra, and Babbage*
- How do they depend on each other? *They are independent but all calculate the same value*
- What is the purpose of the variable Knuth? *Gives the variance of all float values entered by the user*
- What is the purpose of the variable Adleman? *Calculates the average of all float values entered by the user*
- Do you have any comments about the variable types? *All variables are correctly typed*
- What does this program do? *Calculates the variance of all float values entered by the user. It calculates this variance using five different algorithms which all give the same result.*

The time limit for the questions was 15 minutes.

In addition to these questions, we also asked each participant some meta questions about the experiment so that we could analyse the ’bigger picture’.

2 Results

Some background information about our participants: all candidates are computer scientists with some knowledge of static program analysis. The level of expertise in using different analysis methods varies from person to person, but all of them had at some time used program slicing. All of the participants have seen C or C++ before, and most of them consider themselves at least reasonably proficient at reading C code.

2.1 Group 1 – source code only

This was arguably the group with the lowest chance of answering the questions correctly, the source code only group consisted of nine people, only three of them answered the question on the purpose of the program correctly.

Interestingly, almost everyone in this group created another representation of the program in their mind or on paper. Four of the nine people created “some sort of” program dependence graph while others tried to calculate the mathematical outcome of the program, used a symbol table or “walked through” the program.

When asked what they would need to better understand the program, no one single answer stood out. Instead, the group participants asked for: better variable names, a maths book, a dynamic trace or Google.

As for the C language knowledge of group 1, it was perfectly balanced between three novices, three intermediates and three experts, according to their own judgement. Perhaps surprisingly, the three correct answers were given from one novice, one intermediate and one expert.

Group 1 comprised: one female and eight males, the average age of this group was 33.5 years.

As a conclusion for this group, it can be seen that having only the source code is not a great aid to understanding a program, but how the understanding can be improved is unclear (even to those having to read the code).

2.2 Group 2 – source code and execution trace

According to the answers given by the eight people constituting the source and execution trace group, the trace was hardly used at all (with one exception). Instead, the group members concentrated on the source to figure out what the program does. Four of the group members correctly identified the purpose of the program and one of them stated that they had used the execution trace extensively.

As in group 1, the members of this group also created their own representation of the program: two of them created a program dependence graph (of some sort) while others used formal representations of the formulas, symbolic execution, or slices.

When asked they expressed a wish for better variable names or better commented source code along with a statistics book or a program dependence graph.

The C knowledge level was again perfectly balanced, although two members of this group did not specify their level.

Group 2 comprised: six males and two people who declined to give their age or gender, taking them out of the calculation gives an average age of 34.8 years.

In conclusion, it can be seen that the dynamic execution trace did not seem to help people that much, and the correct identification of the purpose of the program was still hard to figure out.

2.3 Group 3 – source code and control flow graph

With help of the control flow graph, nine out of thirteen members of this group correctly identified the purpose of the program. Most of the people however stated that they hardly used the control flow graph at all.

In this group, only six people created a different representation of the program (that is less than fifty per cent of this group). Of those six representations, some form of program dependence graph was created by three people, the others were using "some form of what can be called a slice", "scribbles", or amorphous slices.

Constituting this group were four novice C readers, five intermediates, and four experts, so the knowledge level could again be called fairly balanced.

Group 3 consisted of eleven men, one woman and one person who did not want to specify their gender. The average age was 34.1 years.

So, apparently a control flow graph seems to be a bigger help in understanding the purpose of a program than a dynamic trace, although many participants expressed a wish for better representations.

3 Conclusion

Understanding programs is hard. No matter what static representation of a program we have, the complexity is still overwhelming. Still, using program control flow graphs seems to be a good way for our minds to disentangle the confusing strands of statements in a program. In our experiment, we could not single out the *most effective* program representation, but we could see that any representation in addition to the source code is helpful for program understanding.

Knowing that we have only tested a very tiny program on a small group of software analysis experts, gives us a bit of a headache. Imagine a software developer having to understand a system like Windows or Eclipse and trying to fix a single bug in the program. Can we draw any worthwhile conclusions from our study for general use? Probably not. We have only scratched the surface of program comprehesion and understanding. It is clear from our results that having more than just source code is probably helpful for program comprehension as many participants scribbled an extra representation of the program on a scrap of paper to help themselves understand the program. So what we need in order to understand programs is either an altogether better representation than we have today, or a suite of representations that can be used interchangeably, or perhaps both.

```
#include <stdio.h>
#define TURING 512

int main(int argc, char **argv) {
  float Adleman, Babbage, Booch;
  float Church, Dijkstra, Hoare;
  float Kay, Knuth, vNeumann;
  float Weiser[TURING], Zuse;
  int   Hopper, Stroustrup;

  Kay = Hoare = Zuse = Booch = 0;
  scanf ("%d", &Hopper);
  for (Stroustrup = 0; Stroustrup<Hopper;
       Stroustrup = Stroustrup + 1) {
    scanf("%f", &Weiser[Stroustrup]);
    Hoare = Hoare + Weiser[Stroustrup];
    Zuse  = Zuse  + Weiser[Stroustrup] *
            Weiser[Stroustrup];
  }

  Adleman  = Hoare / Hopper;
  vNeumann = (Zuse - Hopper *
             Adleman * Adleman) /
             (Hopper - 1);
  Dijkstra = (Zuse - Hoare * Adleman) /
             (Hopper - 1);
  Hoare    = Hoare * Hoare / Hopper;
  Church   = (Zuse  -  Hoare) /
             (Hopper - 1);
  Hoare    = 0;

  for (Stroustrup = 0; Stroustrup<Hopper;
       Stroustrup = Stroustrup + 1) {
    Booch = Weiser[Stroustrup] - Adleman;
    Kay   = Kay + Booch;
    Hoare = Hoare + Booch * Booch;
  }
  Babbage = (Hoare - Kay *
            Kay / Hopper) /
            (Hopper - 1);
  Knuth   = Hoare / (Hopper - 1);
  return 0;
}
```

Fig. 1. Obfuscated version of the variance program.

```
#include <stdio.h>
#define MAX 1024
main() {
  float x[MAX], var2, var3, var4;
  float var5, var1, t1, t2, ssq;
  float avg, dev;
  int  i, n;
  t2 = t1 = ssq = dev = 0;
  scanf ("%d", &n);
  for ( i = 0 ; i < n ; i = i + 1)
  {
    scanf ("%f", &x[i]);
    t1 = t1 + x[i];
    ssq = ssq + x[i] * x[i];
  }
  avg = t1 / n;
  var3 = (ssq  - n * avg * avg) /
          (n - 1);
  var4 = (ssq  - t1 * avg) /
          (n - 1);
  t1 = t1 * t1 / n;
  var2 = (ssq  -  t1 ) / (n - 1);
  t1 = 0 ;
  for ( i = 0 ; i < n ; i = i + 1)
  {
    dev = x[i] - avg ;
    t2 = t2 + dev ;
    t1 = t1 + dev * dev ;
  }
  var5 = (t1 - t2 * t2 / n ) /
          (n -1);
  var1 = t1 / (n - 1);
  printf("variance (one pass, using
     square of sum): %f \n",var2);
  printf("variance (one pass, using
     average): %f \n",var3);
  printf("variance (one pass, using
     average, sum): %f \n",var4);
  printf("variance (two pass,
     corrected): %f \n",var5);
  printf("variance (two pass):
     %f \n",var1);
}
```

Fig. 2. Original version of the variance program.

```
Start of program.
Kay set to 0.
Hoare set to 0.
Zuse set to 0.
Booch set to 0.
Hopper set to 4.
Beginning for loop from 0 to 3.
  Iteration 1 of for-loop. Weiser[0] set to 2.40.
  Hoare set to 2.40. Zuse set to 5.76.
  Iteration 2 of for-loop. Weiser[1] set to 8.40.
  Hoare set to 10.80. Zuse set to 76.32.
  Iteration 3 of for-loop. Weiser[2] set to 2.11.
  Hoare set to 12.91. Zuse set to 80.78.
  Iteration 4 of for-loop. Weiser[3] set to 3.67.
  Hoare set to 16.58. Zuse set to 94.25.
End of for loop.
Adleman set to 4.15.
vNeumann set to 8.50.
Dijkstra set to 8.50.
Hoare set to 68.74.
Church set to 8.50.
Hoare set to 0.00.
Beginning for loop from 0 to 3.
  Iteration 1 of for-loop.
  Booch set to -1.75.
  Kay set to -1.75.
  Hoare set to 3.05.
  Iteration 2 of for-loop.
  Booch set to 4.25.
  Kay set to 2.51.
  Hoare set to 21.15.
  Iteration 3 of for-loop.
  Booch set to -2.03.
  Kay set to 0.48.
  Hoare set to 25.28.
  Iteration 4 of for-loop.
  Booch set to -0.48.
  Kay set to 0.00.
  Hoare set to 25.51.
End of for-loop.
Babbage set to 8.50.
Knuth set to 8.50.
End of program.
```

Fig. 3. Execution trace of a program run.

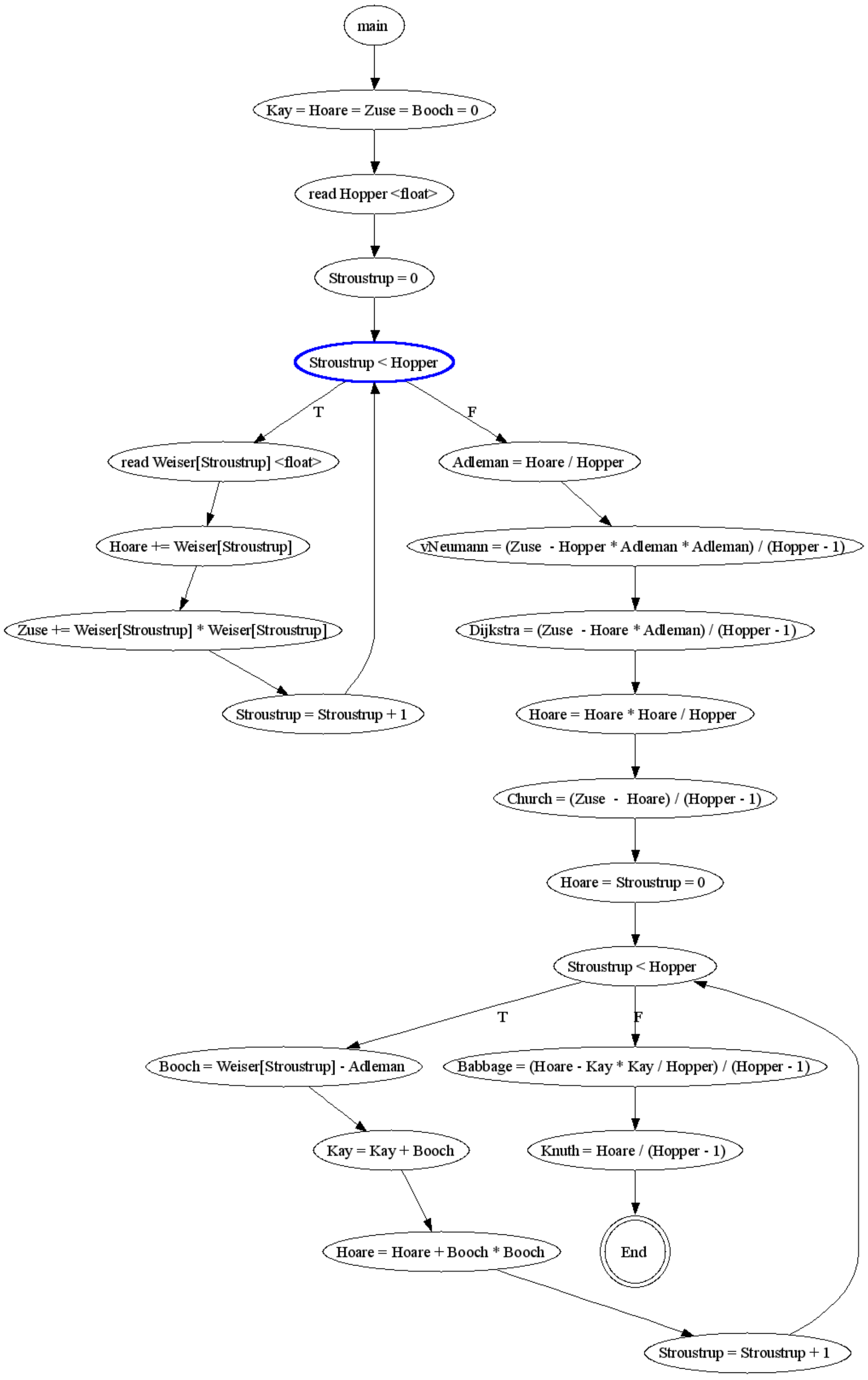
main
Kay = Hoare = Zuse = Booch = 0
read Hopper <float>
Stroustrup = 0
Stroustrup < Hopper
T
F
read Weiser[Stroustrup] <float>
Hoare += Weiser[Stroustrup]
Zuse += Weiser[Stroustrup] * Weiser[Stroustrup]
Stroustrup = Stroustrup + 1
Adleman = Hoare / Hopper
vNeumann = (Zuse - Hopper * Adleman * Adleman) / (Hopper - 1)
Dijkstra = (Zuse - Hoare * Adleman) / (Hopper - 1)
Hoare = Hoare * Hoare / Hopper
Church = (Zuse - Hoare) / (Hopper - 1)
Hoare = Stroustrup = 0
Stroustrup < Hopper
T
F
Booch = Weiser[Stroustrup] - Adleman
Kay = Kay + Booch
Hoare = Hoare + Booch * Booch
Stroustrup = Stroustrup + 1
Babbage = (Hoare - Kay * Kay / Hopper) / (Hopper - 1)
Knuth = Hoare / (Hopper - 1)
End

Fig. 4. Control flow graph of the obfuscated program.