Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 178 (2007) 121–128
www.elsevier.com/locate/entcs

Annotations for Defining Interactive Instructions to Interpreter Based Program Visualization Tools

Essi Lahtinen[1] Tuukka Ahoniemi[2]

Institution of Software Systems
Tampere University Of Technology
Tampere, Finland

Abstract

An interpreter based visualization tool can be used for creating visualization exercises that engage the student to work with the visualization instead of just watching it passively. This article describes how to make the interpreter instruct the student interactively. The idea is to use the interpreter for testing the code that the student has written and give the instructions according to the test results. This only requires minor work for implementation of the visualization tool.

Keywords: Computer Science Education, Programming, Visualizations, Novice Programmers

1 Introduction

To engage the student to learn when using a program visualization, the visualization needs to activate the student to take part in it [4]. Thus the developers of program visualization tools should include features that enable interactive communication between the student and the visualization tool.

This paper introduces a way to make an interpreter based visualization tool to give feedback about the students' own programming solutions. First we introduce an existing technique of annotated instruction of a program visualization tool called VIP. Then we introduce our work in progress and discuss the idea.

[1] Email: essi.lahtinen@tut.fi
[2] Email: tuukka.ahoniemi@tut.fi

doi:10.1016/j.entcs.2007.01.041

2 Program Visualizations

There are numerous ways of implementing program visualizations. At its simplest the visualization can be a line of moving pictures, e.g. implemented with a technique like Flash. The problem of this kind of approach is that the interaction between the user and the tool is very limited. Visual interpreters that work on any program code are easier to adjust to communicate with the user. This kind of program visualization tools, e.g. Jeliot 3 [3] and VIP [5], are available for free use on introductory programming level.

Even if the program visualization tools include features that enable interaction with the user, they are most often used as *illustrative visualizations* [2] just to present concepts and run some example programs. Using the visualization tool as an exercise environment is more beneficial for the students [1].

3 The Instructions and the Annotations

A program visualization tool typically shows the visualized program code in a window. This window can include short comments as a part of the code. The purpose of the visualization is to explain the run of the program to the student. To aid this the visualization tool can include extra instructions to describe the program and to draw the students attention to the essential.

To make sure that the additional instructions do not make the source code unneccessarily long, they can be placed in a separate instruction window. The instruction window can show only the instructions related to the line of code that is currently executed. This way the students who need more explanation can follow long intructions and the students who do not need it can skip reading since the instruction is separate from the actual visualization.

The teacher can write the instructions in the visualized program source code inside comments that contain some special characters to mark them. This way the code can be compiled normally to test that it works properly. The visualization tool will just parse the beginning of the comment to see, whether it will be displayed in the code window or in the instruction window. This is also easy for the teacher since only one text file is required for making a visualization. It is also possible to make the instructions versatile by using some markup language, e.g. HTML. This way the teacher can emphasize some parts of the instruction by using colour or italic, or clarify the text with a little picture.

3.1 The Existing Annotations

To improve the expression of the instructions, the visualization tool developer can add annotations to define the visibility of the instructions. For example, a certain instruction could be shown only when the execution reaches the statement for the first time. When the execution returns again to that statement there will be a new instruction. This way the teacher can build the instructions more useful and interesting for the student. E.g. when executing a function call statement on

Table 1
Annotations that are implemented in VIP.

Annotation	Meaning
/*@<nr>	The instruction is shown on the <nr>:th time when the associated statement is executed.
/*@n	The instruction is shown on all the times when no other number is specified. If used alone, the instruction is shown every time the the statement is executed.
/*@odd /*@even	The instruction is shown on the odd/even numbered times when the statement is executed. (Useful, e.g., in function calls inside a loop structure.)
/*@init	The instruction is shown before the program starts running. Before VIP starts the program run, it goes through all these instructions highlighting the line they are assosiated with. (Can be used to explain libraries, namespaces etc. lines never ran.)
/*@lock on /*@lock off	Editing lock on/off. The locked lines can not be edited in the VIP code editor. Neither do they lose their instructions like unlocked code does after editing. The code is editable as default.

the first time the program reaches that line, the teacher would probably want to explain how the parameter passing happens. When the function call finishes and the execution comes back to the same line the teacher can explain the passing of the return value of the function.

VIP visualization tool includes a few simple annotations that are explained in Table 1. In VIP the instructions are always related to the code line right after the instruction. When using multiple annotated instructions for one line of code the interpreter shows the first instruction whose annotation is valid.

The existing annotations are connected to the execution order of the program. Thus they enable writing wide explanations of the program for the students but do not really support interactivity. If the visualization tool provides a possibility to edit the visualized program source code the tool can also be used for visualization exercises described by Lahtinen and Ahoniemi [2].

3.2 Work in Progress

The instructions can be developed futher so that the values of the variables in the program code can be examined and the instruction text is shown only if the internal state of the visualized program is certain. For example, the teacher can specify a "well done" text to be shown if the student has succeeded to give the program an input that led to a certain state. Or in *a utilizing visualization* if the student has succeeded to modify the program in a right way he can be granted for the exercise.

The annotations to enable this need to include statements that are interpreted.

Table 2
The annotations in development.

Annotation	Meaning
/*@hide on /*@hide off	Hidden code for testing the students solution. Is not run normally or visualized in the visualization window. Is also locked.

Our idea is to use the visual interpreter for interpreting these too. An annotation can mark some parts of the code hidden. In the hidden code, the teacher can test the program. E.g., if the task for the student is to implement a function, the hidden test code can run the function a couple of times and display the instruction text for the student according to the results of the tests.

Figure 1 shows an example code of a programming exercise visualization where the student is supposed to implement a function to sort the integers in an array. The teacher has implemented a main function that calls the function that the student is supposed to design, write and test. The hidden code in the end of the example first tests the students solution and then instructs the student to the right direction. The instruction texts are written to a special output stream named `__vip_out` whose content is displayed in a pop up window. The new annotations used in this code are introduced in Table 2.

When the student is visualizing the code the input stream `cin` naturally works in a normal way: asks the user to write the input. When the code written by the student is tested it usually is not a good idea to ask the input from the user. First of all it could be confusing for the student because he does not know what is happening during the testing and secondly in some tests it would allow the student to cheat. To make sure that the testing is done carefully it is better that the teacher defines the input beforehand.

To able the teacher to predefine content for `cin` stream we are implementing an operator << for a new input stream called `__vip_in`. This actually works as an alias for `cin` stream allowing the use of it as an `iostream`. The reason why `cin << value` is not allowed is that a novice level student could easily typo his code this way and would not get an error message of the typo. An example of the use of the `__vip_in` stream is also shown in Figure 1. In this particular example the feature is used to initialize the array *numbers* in reverse order when the test is run.

To provide more possibilities for the teacher the `__vip_in` stream also has a method called `ok()` that returns a boolean value `true` if everything that was put into the stream with << was also read (if the stream is empty). This can be used for checking that the students solution to a problem actually reads the given input and processes it.

Figure 2 shows how the student will interact with the visualization tool in the visualization exercise described in Figure 1. The phases are marked in Figure 2 as follows: The assignment is presented for the student in a website (1) with a button to open the visualization tool (2). In the visualization tool window the student can start studying the given code and visualize it step-by-step using the “step” button (3). By pressing the “edit code” button (4) the student can proceed to writing his

```
/*@lock on */
#include <cstdlib>
#include <iostream>

using namespace std;

void sort( int array[], unsigned int size ) {
/*@lock off */
   // Press the "edit code" button and write your implementation here!
/*@lock on */
}

int main() {
/*@hide on */
   __vip_in << 3 << 2 << 1;
/*@hide off */
   const int AMOUNT = 3;
   int numbers[ AMOUNT ];

   cout << "Enter " << AMOUNT << " integers: ";
   for( int i = 0; i < AMOUNT; ++i ) {
      cin >> numbers[ i ];
   }

   sort( numbers, AMOUNT );    // The function sorts the elements of the array ascending

   cout << "The integers sorted: " << endl;
   for( int i = 0; i < AMOUNT; ++i ) {
      cout << numbers[ i ] << endl;
   }

/*@hide on */
   int _vip_array2[] = { 2, 1, 3 };
   sort( _vip_array2, AMOUNT );
   bool _vip_test1 = true;
   bool _vip_test2 = true;

   for( int _vip_i = 0; _vip_i < AMOUNT - 1; ++_vip_i ) {
      if( numbers[ i ] > numbers[ i + 1 ] ) {
         _vip_test1 = false;
      }
      if( _vip_array2[ i ] > _vip_array2[ i + 1 ] ) {
         _vip_test2 = false;
      }
   }

   if( _vip_test1 && _vip_test2 ) {
      __vip_out << "Your function seems to work correct! Well done! You can continue with"
                << " the next exercise..." << endl;
   } else if( !_vip_test1 && _vip_test2 ) {
      __vip_out << "Test sorting an array that is in a reverse order. Your function might"
                << " need a correction there." << endl;
   } else {
      __vip_out << "The function does not seem to work very well. Review page 123 of the"
                << " course material and try again." << endl;
   }
/*@hide off */
   return EXIT_SUCCESS;
}
/*@lock off */
```

Fig. 1. An example of the use of hidden code annotations.

solution in the code editor. The student can edit only the unlocked code on the white background (5) (the function to be implemented). After making the changes he can try to compile the code with the "compile" button. When he is satisfied with his solution he returns to the visualization window by accepting the changes (7).

After editing the code the student probably tries to run his own code in the visualization tool either step-by-step or using the "run" button (8). In both these situations only the visible code is executed and the student will see how the code he developed works. If he thinks the task is completed the student can run the

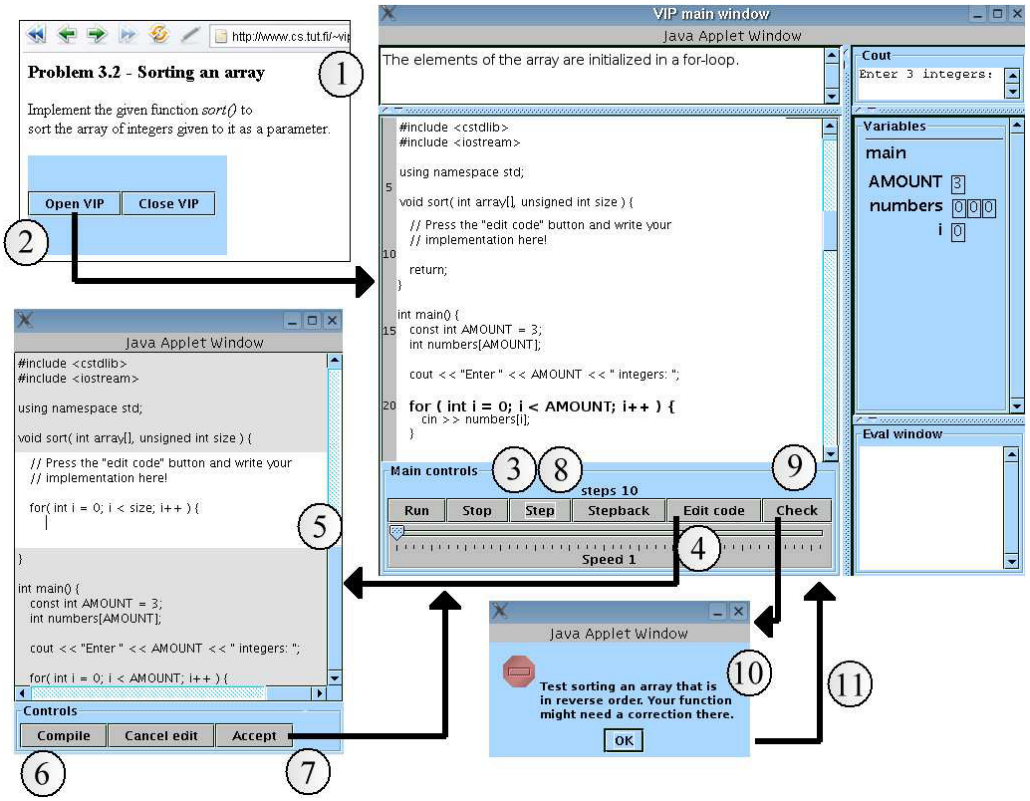

Fig. 2. An example of how the visualization exercise could work.

check (9). As the "check" button is pressed both the visible and the hidden code are executed and a pop-up window containing the results is shown (10). The result window shows the text that was written to the `__vip_out` stream when running the tests in the hidden code.

All of the windows presented in Figure 2 are separate, so the student can still return to the visualization tool or the code editor (11). He will still be able to see the assignment description in the first window and the instructions in the pop up window.

4 Discussion

The functionality introduced here is easy to add to an existing interpreter since it uses the same interpreter to implement the tests in the hidden code. After this addition the visualization tool can be used e.g. for *utilizing visualizations* and *problem-solving visualizations* [2].

This annotation technique is also easy for the teacher. He can check the whole visualization code and the test code with a normal compiler because everything extra is placed inside a comment. The extra output stream can be defined as `cout` using a macro while testing. Also the whole exercise can be created just by writing one text file which is easy for the teacher. Of course designing the exercises requires

time.

The most difficult thing for the teacher in using this approach for testing the students solutions can be that sometimes you need a little bit of imagination to implement the tests. As seen in Figure 1 testing a function implemented by the student is very straight forward. Testing an `if`-statement is also easy if the code inside the `if`-statement modifies the variables of the program. But e.g. testing an `if`-statement that only outputs text needs some more creativity from the teacher: The output texts can be written ready to the solution template inside a locked code. This locked code can also include a hidden code segment that checks that it is executed only when the conditions are correct. The teacher also needs to recognize that the `if`-statement can be run many times like the function is in Figure 1 if the hidden code contains a loop structure.

It is of course possible to add many other types of tests to an interpreter too. E.g., tests for checking all the output of the program or tests for finding certain expressions in the source code. However, this kinds of tests can not be run using only the interpreter. The first one requires an extra process to run the interpreter with different inputs and compare its outputs. The second one requires parsing the source code for a different purpose.

When implementing tests like this to a visual interpreter also the effectiveness of the interpreter should be considered. A visual interpreter can be implemented with the idea that it does not need to be very effective, since the visualization needs to be observed and thus is not supposed to be too fast. In this case it should be checked that the interpreter is effective enough to run the tests in the hidden code in a reasonable time. At least the student should be shown some kind of a progress bar to follow while waiting.

5 Conclusions

It is easy to add annotated instructions to an interpreter based visualization tool. With some imagination these instructions can be used for creating visualization exercises that engage the student to work with the visualization. Adding this little feature to the tool widens the pedagogical uses of the tool greatly.

As our future work we will use the developed visualization exercises as a part of the course material and gather experiences on their usage. We intend to evaluate them by measuring the students learning results.

References

[1] Ahoniemi, T. and E. Lahtinen, *Visualizations in Preparing for Programming Exercise Sessions*, Proceedings of the Fourth Program Visualization Workshop (2006).

[2] Lahtinen, E. and T. Ahoniemi, *Visualizations to Support Programming on Different Levels of Cognitive Development*, Proceedings of The Fifth Koli Calling Conference on Computer Science Education (2005), pp. 87–94.

[3] Moreno, A., N. Myller, E. Sutinen and M. Ben-Ari, *Visualizing programs with Jeliot 3*, Proceedings of the International Working Conference on Advanced Visual Interfaces AVI 2004 (2004).

[4] Naps, T., G. Rössling, V. Almstrum, W. Dann, R. Fleischer, C. Hundhausen, A. Korhonen, L. Malmi, M. McNally, S. Rodger and J. Velazquez-Iturbide, *Exploring the role of visualization and engagement in computer science education*, SIGCSE Bulletin **35** (2003), pp. 131–152.

[5] Virtanen, A. T., E. Lahtinen and H.-M. Järvinen, *VIP, a visual interpreter for learning introductory programming with C++*, Proceedings of The Fifth Koli Calling Conference on Computer Science Education (2005), pp. 129–134.