Explanatory Visualization in an Educational Programming Environment: Connecting Examples with General Knowledge

Peter Brusilovsky

International Centre for Scientific and Technical Information,
Kuusinen str. 21b, Moscow 125252, Russia
E-mail: plb@plb.icsti.su

Abstract: Explanatory program visualization is a name for program visualization extended with natural language explanations. Explanatory visualization can seriously increase students' understanding of program behavior. This paper gives the rationale and background for explanatory visualization and introduces our work on using explanatory visualization in educational programming environments. In particular, we present first experimental results on using explanatory visualization and provide a fine-grained description of the implementation of adaptive explanatory visualization in our ITEM/IP-II system. This system employs student model to adapt the visualization to the student knowledge level.

1 Introduction

Intelligent programming environments for novices (Brusilovsky, 1993a) attempt to bridge the gap between Intelligent Tutoring Systems for programming and novice programming environments. An Educational Programming Environment (EPE) is a set of tools supporting the student in the process of learning introductory programming. Some of these tools support students' activities in program design and debugging, other tools support teachers' activities such as presenting new material or evaluating students' programs. Good examples of tools provided by an EPE are program visualization tools and intelligent program debuggers. A program visualizer is a tool that provides the student with an animated representation of program execution and makes some of the hidden aspects of program behavior visible to the programmer. A program visualization tool is an important component of a learning environment for programming. Such a tool enables the student to build a clear mental model of virtual (notional) machine behavior (du Boulay et al, 1981), as well as understand the semantics of programming language constructs and the behavior of algorithms. There are a number of systems and environments that employ program visualization (see McGlinn, Britt & Woolard, 1989; Sanders & Gopal, 1991 for a good review and reference list). Traditionally program visualization is comprised of flow of control visualization and data structure visualizations on different levels. The traditional role of program visualization is to enhance the understanding of the semantics of programming constructs and algorithms and to support learning through exploratory programming.

For a number of years we have been working on EPE problems. In 1985 we designed the ITEM/IP environment (Brusilovsky, 1992) to support an introductory programming course for first year students at Moscow University. The course was based upon the visual, educational mini-language Turingal (Brusilovsky, 1991), which is a combination of a Turing machine language with Pascal control structures.

ITEM/IP consists of several modules that support student and teacher activities. Two important modules are the visual interpreter for Turingal and the evaluation module. The interpreter produces flow of control and action visualizations: each statement is marked before being executed and the machine head moves along the tape and replaces symbols as the corresponding statement is executed. The interpreter also produces some visual effects while evaluating conditions in ``while" and ``if" statements. The evaluation module checks the student's solution of a programming problem. The module applies a simple, but effective test-based method. Each programming problem presented to the student has a prestored model solution and a set of input tests. The evaluation module compares the results produced by the model solution to the results produced by the student solution for each test. If the results differ for one of the tests, the test is called ``faulty" and the student program is considered to be wrong in some way, otherwise it is considered to be correct.

The visualization tool was widely used by the tutoring module in the explanation and debugging stage. First, visual examples were used to explain the semantics of programming language constructs to the student. Second, the system provided debugging assistance by visually executing the student program on the ``faulty" test found by the evaluation module, thus visually demonstrating student errors. Our first experience with ITEM/IP in 1985 showed that the use of the visual interpreter for explanation and debugging significantly increased students' understanding of both language semantics and of their own bugs. However, more experience with ITEM/IP in 1986-1987 showed that program visualization is sometimes a less effective tool than it is expected to be.

We have revealed an interesting phenomenon - quite often weak students (and sometimes average students) simply cannot make sense of the visualizations. They look at the visualized piece of code but ``do not understand" what is happening on the screen. For example, if an important loop in a buggy program is always skipped due to an incorrectly written condition (the cause of the error), the students sometimes cannot understand why the flow of control pointer passes this loop? They will run the program again and again on the same faulty test with no changes. The interview showed that they really do not understand why the loop is passed. At the same time, most of the students can answer questions about the semantics of the loop correctly. Thus they have good general knowledge, but often fail to apply it to a specific example.

Our ``first aid" to such students was to explain in words how the given construct behaves in the given example. For example, the following explanation can be presented by an assistant when checking the condition of a ``while A" statement: ``The condition `A' of the while statement is false because the tortoise stands on letter `B' which is not `A'." On the next step of the execution the assistant might say: ``Since the condition of the while statement is false, we jump to the statement after `endwhile', and do not execute the body of the loop." Such natural language explanations complement regular visualization and appear to provide very effective in-

structional help. Our assistants were often working in exactly this way, as ``explainers'' of what the program is doing, often repeating the same explanations.

The above experience motivated our current research and development work on explanations in program visualization. The preliminary results of this work are presented in this paper. We provide a brief summary of research on example explanation in programming, then we introduce our work on (what we call) explanatory visualization in a new ITEM/IP system and present the first experimental results which support our hypotheses and design decisions. The second part of the paper is devoted to a fine-grained description of the implementation of explanatory visualization in the ITEM/IP-II system.

Background: Example explanations in teaching and learning programming.
The most relevant research on using explanations of examples in teaching programming was performed by the group of Peter Pirolli. Their ideas were based on previous research into the role of examples in programming (Pirolli & Anderson 1989) and on findings about the role of self-explanations in the domain of physics problem solving (Chi at al, 1989). Investigating student self-explanation strategies in the programming domain (Pirolli & Bielaczyc, 1989), Pirolli, et al. found that the self-explanations students made while studying instructional materials correlated with the corresponding problem solving performance of those students. Students showing good performance not only generated more self-explanations than students showing poor performance, their explanations were qualitatively different. In particular, the following two effective self-explanation strategies were identified: connecting ideas in the texts with their instantiations in the examples (and vice-versa), and determining the meaning of LISP code presented in the examples. Good performers generated almost an order of magnitude more explanations connecting portions of the example solution to concepts introduced in the text.

Such findings can be used to produce better learning support systems. The first idea is to make a meta-cognitive tool that supports self-explanations by providing students with the ability to make written comments about both provided examples and their own programs. Unfortunately, such tools that have been designed recently (Recker & Pirolli, 1992; Linn, 1992) do not appear to be very effective. Many students consider such commenting activity as a waste of time.

From our point of view, the educational role of self-explanations is to provide a way for the student to relate their general knowledge about a subject with situational knowledge represented in a particular example. Using Clancey's terms (1987), self-explanations establish the links between general models and situation-specific models of an example. There is, however, another way - the instructional materials provided by the system can be extended with special example explanations that connect the problem examples with general knowledge. System-provided explanations are not as effective as self-explanations, but they can be easily provided and are very beneficial for the students, as was shown by Recker and Pirolli (1992). They designed a hypermedia-based system that contained a set of examples. These examples were annotated with explanatory elaborations (accessed via mouse clicks), that explained how programming principles were implemented within a concrete model. The idea of representing examples augmented with explorable explanations has also been implemented in other systems, such as Molehill (Singley & Carrol, 1992) and

Explainer (Redmiles, 1993). These tools were also effective, but their implementation was inspired more by experience of the authors than by cognitive considerations.

The work of Pirolli and his group gives more support to our hypotheses about our problems with understanding visualization. Weak students often cannot understand the visualization, because they do not relate the behavior of a particular construct in an example with their general knowledge about the semantics of the construct. An assistant's explanations bridge the gap between the example and general knowledge and enable the student to understand the example. Stronger students, or the students with more experience, have well established connections between general knowledge and situational knowledge that control the interpretation of examples, so they require example explanations much less frequently.

Interestingly, a similar phenomenon was reported in a related domain - algorithm visualization (Stasko, Badre & Lewis, 1993). It appears that novices and weak students benefit much less from algorithm visualization than was expected. Based on our experience and the ideas of Pirolli, we can explain this phenomenon: novices didn't relate their general knowledge of algorithms with the animation of a particular case on the screen. We think specific system-provided explanations bridging this gap can be of significant help here.

2 Explanatory Visualization: First Experience

Once our experience has shown that example explanations are helpful to resolve student problems with understanding visualization, the next step was to design a tool that can generate such explanations for the student. We have designed such a tool as a component of our recent ITEM/IP-II system that was designed to support a part of ``the computer literacy" course for 14--16 year-old students of Moscow schools. ITEM/IP-II is similar in its architecture to ITEM/IP and uses a similar mini-language called Tortoise.

One of the new features of ITEM/IP-II is what we call "explanatory visualization". In addition to standard visualizations, the visual interpreter uses a special window to explain all the steps of the executed program in the same way that human assistants sometimes do when working with ITEM/IP. Note that our explanatory visualization tool differs from the tools suggested by Pirolli and other authors. First, all existing example explanation tools are applied to examples of program design, while our tool deals with examples of program behavior. Second, all other tools use static explanations of examples, provided by a course designer beforehand; our tool is able to generate explanations for any given example, either contained in the course, or suggested by the student. In this sense, our system is similar to the SCENT (McCalla, Greer, et al, 1992) program understanding system, which can explain the role of particular lines in a recursive student program in terms of standard recursion techniques. An ability to generate explanations requires expert knowledge to be represented in the tool - in our case, knowledge about language semantics.

Last year we completed the first classroom study of the ITEM/IP-II system. One of our goals was to measure the role of explanatory visualization in program debugging. The subjects in this study, 30 students from Moscow Lyceum of Information

Technologies, were divided into two groups. The students were 15--16 years old and most of them had never had any programming experience. The subjects were presented with a course of introductory programming based on the Tortoise mini-language. A course contains 14 lessons of 40 minutes each. Six lessons were spent presenting new material and solving some problems in a blackboard classroom. Each of these lessons was followed by a lesson in the computer classroom where the subjects used the ITEM/IP-II system to solve a sequence of related problems. The students used the structure editor and the visual interpreter to prepare solutions and called the evaluation module to check a solution when they thought it was ready. The solution was checked on a sequence of tests. If the solution appeared to be correct, then the student was presented with the next problem. Otherwise the evaluation module determined the ``faulty'' test and beeped to call one of the assistants to the student's computer for a short ``interview''.

The role of the assistant was to test several kinds of tools aimed at helping the student to understand the source of the error. At the beginning of the ``interview,'' and again after trying each tool, the assistant checked to determine whether the student understood the location and the source of the bug. If the student claimed that the error was understood and explained it correctly to the assistant, then the case was recorded and the student was permitted to correct the program. Otherwise, the next tool was applied. The first tool presented to the student showed the unequal results produced by the student solution and by the model solution on the ``faulty'' test. Then the student was presented with a standard, computer-generated visual execution of his or her incorrect solution on the faulty test. Next the assistant, with the use of the computer, formally simulated a standard explanatory visualization, as it is implemented, and then gave an adaptive explanatory visualization. Finally, the assistant applied his or her own intelligence to explain the error until the student understood it.

We utilized simulated vs. automatic explanatory visualizations because we wanted to obtain some feedback on our explanation methods, and because explanatory visualization was not well-tested enough to be used in a real classroom. However, all of the assistants were carefully instructed to produce standard explanatory visualizations as they were implemented in our system. Adaptive explanatory visualizations were produced less formally than standard visualizations, but more formally than ad-hoc assistant explanations. We found one very interesting result of the experiment was that sometimes ``assistant explanations'' were just the same explanatory visualizations given in an earlier step, but adapted to the student in two ways: the better the particular student's knowledge, the less detailed the explanations were, and at the same time, the closer the explanation was to the source of an error, the more detailed it was. It was about in the middle of the experiment that we formalized the rules of adaptive visualization and started counting it as a separate case (so the effect of adaptive visualization is probably even bigger than the data show). Up to six assistants were employed to work with each of 15 students in a group. They processed 167 interviews. The following table presents the proportion of students that understood the error after the corresponding tool was applied to the ``faulty'' test.

Table 1. The results of experiment

Tool	Interviews	%
Before all tools were applied	10	6.0%
Demonstrating results	32	19.0%
Standard visual execution	64	39.0%
Simulated explanatory visualization	34	20.0%
Simulated adaptive visualization	11	6.5%
Assistant explanations	16	9.5%
Total	167	100%

The results confirm our hypotheses that visualization is really a good tool for program debugging. We also found that explanatory visualization can significantly increase the effect of visualizations, leaving only 16% of errors uncovered, and that adaptive visualization provides further improvement. Measuring the role of visualization was not the only goal of the experiment. However, we obtained encouraging results as well as some new ideas about how to improve the visualization. Of course, even formally simulated explanatory visualization is not the same as computer generated visualization. First, human assistants cannot be as rigid as a computer, even if instructed. Second, simulated visualization was presented in a spoken way, i.e. in a different modality than the computer-generated visualization. Both aspects make simulated visualization more effective than computer-generated visualization. We plan to study the role of explanatory visualization (including adaptive visualization) more formally in the next studies.

3 Adaptive Explanatory Visualization in ITEM/IP-II

The results of the experiment gave us some idea about how to improve explanatory visualization in ITEM/IP. The current version of ITEM/IP-II includes not just explanatory visualization, but also adaptive explanatory visualization. At present we have implemented automatic adaptation of generated explanations for various student knowledge levels. This kind of adaptation was investigated by us previously (Brusilovsky, 1992a) for the case of regular visualizations. With adaptive program visualization the level of detail given in the visualization of a programming construct is made dependent on the differences in the students' knowledge about this construct. Adaptive visualization appears to be a very useful feature. For explanatory visualization adaptation is even more important. Explanatory visualization produces large amounts of text, so students may get lost in this stream of explanations and miss the important piece of the explanation.

There are 13 constructs in the Tortoise mini-language having from 1 to 4 degrees of visualization. The current state of visualization is determined by the visualization status vector which consists of 13 integers. This vector is formed using the central student model in the system. Each component of the vector determines the level of visualization for one of the constructs (level 1 - concept is new or poorly studied - maximum degree of visualization; level 2 - concept is understood better -

visualization is less detailed; ...; 4 - concept is understood well enough and does not need to be visualized). For each construct a template with stop points was developed. For example, consider the operator "IF < condition > THEN - ELSE - END_IF". This operator has five potential stops (figure 1) for stepwise execution mode. At each stop the corresponding part of the operator is pointed out and an appropriate message is displayed in the explanation window.

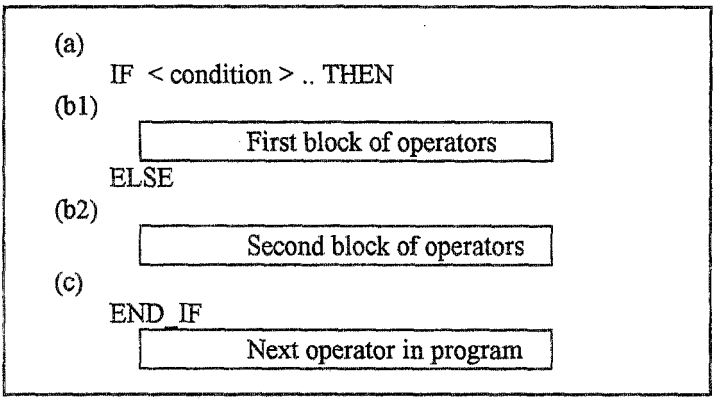

Figure 1. A construct template with visualization stops

Examples of comments for several degrees of visualization are given bellow.

1-st degree of visualization:

(a) - Operator "IF < condition > THEN - ELSE - END_IF" consists of one condition and two branches - "THEN - ELSE" and "ELSE - END_IF". First we shall check the condition: true or false. (Here a visualization of checking the condition can follow).

(b1(b2)) - Condition is true (false)- control is passed to the "THEN - ELSE" ("ELSE - END_IF") block of operators.

(c1(c2)) - The "THEN - ELSE" ("ELSE - END_IF") block of operators has been executed - go to the operator following END_IF.

2-nd degree of visualization:

(a) - Operator "IF". Checking if the condition is true or false. (Here a visualization of checking the condition can follow).

(b1(b2)) - Condition is true (false) - control is passed to the THEN (ELSE) block of operators.

No visualization and no stop at points (c1, c2).

3-rd degree of visualization:

(a) - just stop without any explanation message

(b1(b2)) - Condition is true (false) - control is passed to the THEN (ELSE) block of operators.

No visualization and no stop at points (c1, c2).

4-th degree of visualization:

operator "IF" is well understood, we shall not explain it, however the silent stop is kept at (a).

All text displayed in the visualization window is stored in a file in an easily modifiable format (figure 2). This was specifically done to provide the teacher with the ability to represent pedagogical knowledge about the required behavior of the visualizations. An example of the explanation message representation is given in figure 3. The role of the teacher is important for the explanatory visualization mechanism. Actually, it's the knowledge and the experience of the teacher that should determine when and how to use explanatory visualization. The system provides the teacher with a great deal of control over the use of visualization. The teacher can change the vector of parameters of visualization, the number of visualization degrees for any concept, the number of break points (stops), and the content of all visualization messages for every degree of visualization.

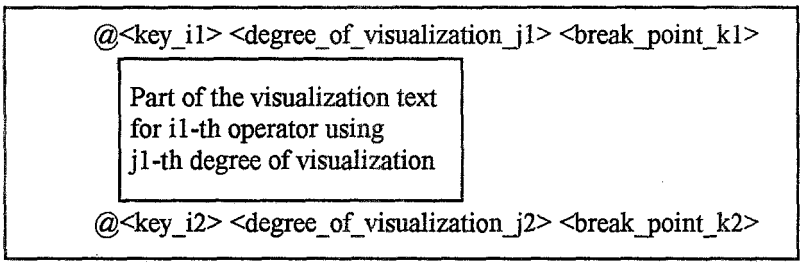

Figure 2. Representation of teacher's knowledge about visualization

```
@if 2 1
Operator "IF". Checking the condition true or false.
@if 2 2
Condition is true (false) - control is passed
to the THEN (ELSE) block of operators.
@if 2 3
NIL      . . . . .
```

Figure 3. An example of explanations for operator "IF" (translated from Russian). If the word "NIL" is stored, instead of text , then there is no stop for the given degree of visualization at this point.

4 Student Modeling for Adaptive Explanatory Visualization

The vector of visualization parameters is constructed as a dynamic projection of the central student model. According to our general approach to student modeling (Brusilovsky, 1993b), the central student model keeps the history of student interactions with system components, with the domain model concepts clearly indicated.

For the case of the visualization module, the central student model stores the number of its executions for each level of visualization separately for each construct (table 2).

Table 2. Individual parameters for student modeling

	regular non-stop execution	regular stepwise execution	executed with visualization of 1st degree	executed with visualization of 2nd degree	... ...
number of repetitions	X_1	X_2	X_3	X_4	...
weight	a_1	a_2	a_3	a_4	...

$X_1, X_2, X_3, \ldots$ - how many times a construct was executed in different conditions;
$a_1, a_2, a_3, \ldots$ - weights to calculate the cumulative amount of visualization.
For example: $a_1 = 0.1$, $a_2 = 1$, $a_3 = 5$, $a_4 = 4$, $a_5 = 3$, $a_6 = 3$, $a_7 = 2$.

When projecting the central student model onto the vector of visualization parameters, first the cumulative amount Y of the visualization the student has received for a construct is calculated.

$$Y = a_1 * X_1 + a_2 * X_2 + a_3 * X_3 + \ldots + a_i * X_i;$$

The weights (a_1, a_2, ...) are specific to each student (table 2). They are stored in a file, and may be changed by the teacher. The visualization parameters for a construct are determined from Y using thresholds R_1, R_2, R_3 and R_4 for the 1-st, 2-nd, 3-rd, and 4-th degrees. If Y falls within segment $[R_{i-1}, R_i]$, then the i-th degree of visualization is assigned.

Thus the historic content of the central student model is currently projected into the vector of visualization parameters and used to control the visualization granularity. If the student is not satisfied with the degree of visualization, the vector of visualization parameters may be modified by the student or the teacher for better comprehension. More details about student modeling for adaptive visualization can be found in (Brusilovsky, 1992a, 1993b).

5 Conclusion

We have presented some ideas about explanatory program visualization. We describe the technique of adaptive explanatory visualization developed for a mini-language and provide some experimental results for its effectiveness. We consider two directions of further work on explanatory visualization. First, a more formal study should be conducted to carefully measure the contributions of both adaptive and standard explanatory visualization. Second, a similar approach can be used to support other parts of teaching programming. A particularly interesting area to support is the production of adaptive explanatory visualization at the level of algorithms. We think

that such an explanatory visualization can solve some of the problems with algorithm visualization reported by Stasko, Badre and Lewis (1993). Eisenstadt et al. (1992) also suggest some good ideas regarding the application of knowledge about plans and algorithms to dynamic algorithm visualization. In addition, some of the existing approaches to program understanding and plan recognition can be used to evaluate the student knowledge about plans and to update the student model.

References

1. du Boulay J.B.H., O'Shea T., Monk J. (1981) "The black box inside the glass box: Presenting computing concepts to novices", International Journal on the Man-Machine Studies, v.14, 237-249.
2. Brusilovsky P. (1991) "Turingal - the language for teaching the principles of programming", Proc. EUROLOGO-91 conference, August 1991, Parma, 423-432.
3. Brusilovsky P. (1992a) Adaptive visualization in an intelligent programming environment. In: J.Gornostaev, (ed.) Proceedings of the East-West International Conference on Human-Computer Interaction, EWHCI'92, St.Petersburg, 4-8 August, 1992. - Moscow, p.46-50.
4. Brusilovsky P. (1992b) "Intelligent environment, tutor and manual for introductory programming", Educational Technology and Training International, v.29, n.1, 26-34.
5. Brusilovsky P.L. (1993a) Towards an intelligent environment for learning introductory programming. - In: E.Lemut, B.du Boulay, G.Dettori (eds.) Cognitive models and intelligent learning environments for learning programming. Springer-Verlag, p.114-124.
6. Brusilovsky P. (1993b) Student as user: Towards an adaptive interface for an intelligent learning environment. - In: P.Brna, S.Ohlson and H.Pain (Eds.) Proceedings of AI-ED'93, World Conference on Artificial Intelligence and Education, Edinburgh, 23-27 August 1993, AACE, Charlottesville, p.386-393
7. Chi M.T.H, Bassok M., Lewis M.W., Peiman P., Glaser R. (1989) Self-explanations: How students study and use examples in learning to solve problems. Cognitive Science, 13, 145-182.
8. Clancey W.J. (1988) The role of qualitative models in instruction. In J.Self (ed.) Artificial intelligence and human learning. Chapman and Hall, London, 49-68
9. Eisenstadt M., Price B.A., Domingue J. (1992) Software visualization vs. ITS: a better way forward In Brusilovsky P. and Stefanuk V. (eds.). Proc. East-West Conference on Emerging Computer Technologies in Education, Moscow, April 1992, 111-115.
10. Linn M.C. (1992) How can hypermedia tools help teaching programming. Learning and instruction, 2, 119-139.
11. McCalla G.I., Greer J.E et al (1992) Granularity hierarchies. International journal of computers and mathematics with applications, 23, 363-376.
12. McGlinn R.J., Britt M., Woolard L. (1989) APEX1, a library of dynamic programming examples. SIGCSE bulletin, v.21, n.1, 98-102.

13. Pirolli P. and Anderson J.R. (1985) The role of learning from examples in the acquisition of of recursive programming skills. Canadian Journal od Psychology, 39, 240-272.
14. Pirolli P. and Bielaczyc K. (1989) Empirical studies of self-explanations and transfer in learning to program. Proc of 11th Annual Conference of the Cognitive Science Society. Lawrence Erlbaum, Hillsdale, 450-457.
15. Recker M.M. and Pirolli P. (1992) Student strategies for learning programming from a computational environment. In: Proceedings of Second International Conference, ITS'92, LNCS N.608, Springer-Verlag, Berlin, p.499-506.
16. Redmiles D.F. (1993) Reducing the variability of programmers' performance through explained examples. In Human factors in computing systems. Proceedings of INTERCHI'93. Amsterdam, 67-73.
17. Sanders I. and Gopal H. (1991) AAPT: Algorithm animator and programming toolbox. SIGCSE bulletin, v.23, n.4, 41-47.
18. Stasko J., Badre A. and Lewis C. (1993) Do algorithm animations assist learning? An empirical study and analysis. In Human factors in computing systems. Proceedings of INTERCHI'93. Amsterdam, 61-66