Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 178 (2007) 145–152
www.elsevier.com/locate/entcs

Distributed Framework for Adaptive Explanatory Visualization

Tomasz D. Loboda[1]

Department of Information Sciences and Telecommunications
University of Pittsburgh
Pittsburgh, PA, USA

Atanas Frengov[2]

Department of Computer Science
Ramapo College of New Jersey
Mahwah, NJ, USA

Amruth N. Kumar[3]

Department of Computer Science
Ramapo College of New Jersey
Mahwah, NJ, USA

Peter Brusilovsky[4]

Department of Information Sciences and Telecommunications
University of Pittsburgh
Pittsburgh, PA, USA

Abstract

Educational tools designed to help students understand programming paradigms and learn programming languages are an important component of many academic curricula. This paper presents the architecture of a distributed event-based visualization system. We describe specialized content provision and visualization services and present two communication protocols in an attempt to explore the possibility of a standardized language.

Keywords: explanatory visualization, program visualization, visualization framework, distributed architecture

[1] Email: tol7@pitt.edu
[2] Email: afrengov@ramapo.edu
[3] Email: amruth@ramapo.edu
[4] Email: peterb@pitt.edu

doi:10.1016/j.entcs.2007.01.044

1 Introduction

Research teams working on modern algorithm and program visualization environments have been exploring a wide range of new directions in answer to the challenge of making visualization more effective, useful, and graphically rich while decreasing the effort required to maintain high quality. For example interactive questions engage students into working with visualization [5] turning them from passive observers to active learners. Explanatory narratives [2][3][7] helps students understand ideas that have been presented visually. Adaptive visualization helps to focus student attention on individual's least understood concepts [1]. The multitude of research directions has led to a situation that is typical for a research-intensive field: no team has expertise in all aspects of modern visualization and no existing tool supports all desired functionalities. In this situation, the traditional method of producing visualization environments – as monolithic software applications working on a specific platform – restricts further progress in the field. The high price of developing each desired functionality combined with the inability to integrate functionalities already developed by different teams makes it impossible for each team to explore feature-rich environments.

A possible solution is to decompose monolithic visualization environments into several reusable, communicating components. This solution was proposed by the ACM SIGCSE Working Group [6]. The architecture suggested by the group separates production of the visualization trace from its interactive rendering and possible enhancement with questions and narration. A set of XML-based protocols provides the connection between these identified components.

Our research groups at the University of Pittsburgh and Ramapo College fully support this vision. Our current NSF-supported project is focused on increasing the value of visualization with explanations and adaptation components. The distributed XML-based architecture could significantly help us to disseminate the results of our research since virtually any open visualization environment can be enhanced with explanations and personalization. To support work on the newly distributed architecture our groups have attempted to restructure our work on adaptive explanatory visualization in terms of the new architecture. This paper presents our first attempt to implement explanatory visualization in a distributed framework following the ideas of Workgroup [6]. We start by presenting the ideas and the original implementation of event-based explanatory visualization, which is the foundation of our work. Then we present our distributed architecture focusing on both components and communication protocols. The paper concludes with a discussion of current and future work.

2 Event-Based Visualization

The core concept behind the proposed distributed framework for visualization is the flow of "interesting events" that are produced by a program or an algorithm. An "interesting event" is simply an event that is important for understanding some concept and should be presented visually, for example, a variable assignment or

removal of an item from a linked list. To increase the pedagogical value of visualization, an event could be extended with explanation (narration), a question that challenges the student to predict the result, etc. The ability to represent the flow of interesting events in a standardized XML format allows to decompose monolithic visualization application into a system of several independent components such as producers, enhancers, and players within the events flow.

3 Architecture

Following the proposal of the ACM SIGCSE Working Group [6] we have implemented an architecture of a distributed model of communication between content providers and content visualizers (Figure 1). The idea behind it is straightforward. *Content Provider* (*CP*) outputs an XML stream containing code, events, and explanations (*c-XML* for *content-XML*). *Content Visualizer* (*CV*) expects an object oriented *v-XML* (*visualization-XML*). *Value-Added Service Provider* (*VASP*) translates *c-XML* into *v-XML* and enriches the stream with pertinent questions that the *CV* is capable of presenting.

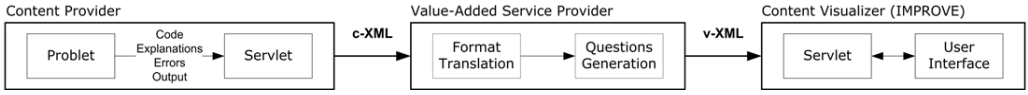

Fig. 1. In the proposed architecture content provision and presentation services are separated. The intermediate component takes care of format translation and question-generation.

The two formats involved originate from different perspectives. *c-XML* represents experiences of content provision oriented approaches. *v-XML* is shaped to best suit presentation purposes. The current work attempts to explore the possibility of having only one format. Because of that *Format Translator* (*FT*) is an optional component of the framework. At the same time, the presence of *FT* is depicting a real-life scenario where a system already capable of generating an XML-based visualization specification does not comply with the standard format. *FT* enables such a system to become a part of the distributed framework and use a non-native visualization engine of its choice (or even multiple engines at the same time).

4 Content Visualizer

From the end user point of view the implemented system (named IMPROVE) is an educational application that supplements learning of programming skills. It does so by visualizing and explaining the execution of snippets of code in a given programming language. The user interface is divided into five areas: Code, Visualization, Output, Explanations, and Navigation (Figure 2).

The system is interactive. The student can navigate the execution of the program forward or backward. Jumping to a specific line of code is also possible. If a question is encountered on the way to the user-designated line the execution stops there and the student is presented with a prompt. That prevents the student from

missing a question. As the student goes through the program execution the current line is highlighted, the output of the program is updated, and relevant visualizations are presented (currently only variables-related). Many steps will also involve presenting textual explanations. The system is capable of asking questions. Four types of these are currently supported: *one-of-many* (radio buttons), *multiple-choice* (checkboxes), *short-text-input* (one line text control), and *long-text-input* (multiple lines text control). When a question is posted the student can skip it, attempt to answer, or say "I don't know" (certain system modes restrict skipping a question).

The system can work in one of four modes: *Exploration* (free navigation; no questions asked), *Challenge* (free navigation; questions are asked, but can be skipped; feedback is presented after each answer), *Evaluation* (forward-restricted navigation; questions cannot be skipped; feedback is presented after each answer), and *Quiz* (forward-restricted navigation; questions cannot skipped; no feedback is presented after an answer). The first two modes let students interact with the material on a stress-free basis. The two last modes are designed to be used by teachers to administer quizzes (with or without feedback).

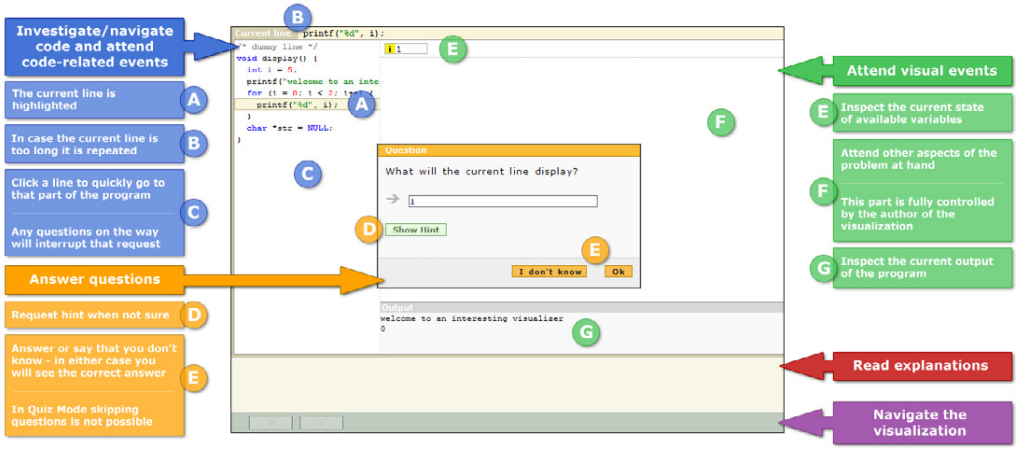

Fig. 2. The user interface of IMPROVE visualization engine. The system is working in Quiz Mode and is waiting for student to answer a question.

The visualization engine treats everything as an *action* (or event in the event-based visualization paradigm). The engine distinguishes two types of them: *non-step-based* and *step-based*. The first type is executed instantaneously (or consists of a single step). An example of such an action would be *show-object*. The second type of action takes a given amount of time to execute (or consist of a number of steps). An example could be *move-object* which would change the object's location in an animated manner (as opposed to e.g., *set-object-location*). At this time the visualization engine supports the following nine actions: *set-current-line*, *show-explanation*, *hide-all-explanations*, *print*, *define-variable*, *assign-variable*, *post-increment*, *ask-question*, and *wait*. The IMPROVE system is available at http://kt2.exp.sis.pitt.edu:8080/improve.

5 Content Provider

Problets are web-based tutors for the C/C++/Java/C# programming languages that have traditionally generated problems, their solutions, feedback, and visualization within a monolithic system. The point of this research was to test whether the problems, feedback and visualization generated by problets could be rendered by a non-native visualizer such as IMPROVE. To this end, a servlet was developed (Figure 1) to (a) convert problets from an application to a service, and (b) translate the output of problets into *c-XML*.

For each programming problem, a problet randomly generates a program and annotates each line of code with its line number. It executes the program using a custom interpreter [3], and automatically generates narration of the step-by-step execution of the program [4], indexing each line of explanation with the line number, program objects and program events that contribute to the explanation. Finally, the problet provides the learner interactive controls to visualize the step-by-step execution of the program (it uses the explanation to drive such visualization). The servlet (Figure 1) converts the explanation generated by a problet into the *c-XML* event stream that can be obtained and visualized by any distributed client. More information on problets is available at http://www.problets.org.

6 Communication Protocol

Our vision of the future involves many specialized content provision and content visualization services. These services could possibly appear and disappear on an ad-hoc basic. Such dynamic framework of responsibilities delegation has to be based on a standard communication protocol. One of our most important research agendas is to move towards defining such a protocol. In Section 6.1 we present the visualization specification expected by our visualization engine (*v-XML*). In Section 6.2 we present the content generated by our content provider (*c-XML*). For the purpose of illustration we use a one-line C language code snippet printing "Hello World!" in the console. The translation between these two formats is based on XSLT, a language for transforming XML documents into other XML documents [8]. The translation process is discussed in Section 6.3.

6.1 Visualization Specification (v-XML)

The XML format expected on the *CV* side consists of five parts presented below. It is object-based in that it first defines objects and then actions that take those objects as arguments. In *v-XML* actions drive the execution sequence.

```
<!-- 1. Lines of code -->
<code-lines>
  <item id="ln1">printf("%s", "Hello World!");</item>
</code-lines>

<!-- 2. Explanations -->
<explanations>
  <item id="expl1">
    <code>printf</code> function will display <code>Hello World!</code>.
  </item>
</explanations>
```

```
<!-- 3. Questions -->
<questions>
  <item id="q1" type="short">
    <text>
      What will be displayed in the current line?
    </text>
    <answer>Hello World!</answer>
    <hint><code>printf</code> function takes several parameters</hint>
    <feedback>
      Here, <code>printf</code> function is instructed to display
      a string. In this case this string will be <code>Hello World!</code>.
    </feedback>
  </item>
</questions>

<!-- 4. Output -->
<output>
  <item id="out1">Hello World!</item>
</output>

<!-- 5. Actions to be executed the execution sequence -->
<code-execution>
  <action type="set-current-line" line-id="ln1"/>
  <action type="show-explanation" explanation-id="expl1"/>
  <action type="ask-question" question-id="q1"/>
  <action type="print" output-id="out1"/>
  <action type="wait"/>
</code-execution>
```

6.2 Content Specification (c-XML)

The XML format generated on the *CP* side consists of four parts presented below. In *c-XML* explanations drive the execution sequence.

```
<!-- 1. Lines of code -->
<code-text>
  <line id="1" indentation="1">printf("Hello World!");</line>
</code-text>

<!-- 2. Explanations -->
<explanation>
  <event line-num="1" context="Variable" event-type="INITIALIZATION">
    <CD>printf</CD> function will display <CD>Hello World</CD>
  </event>
</explanation>

<!-- 3. Output -->
<output>
  <single-output line-number="1">
    <text>Hello World!</text>
  </single-output>
</output>

<!-- 4. Errors -->
<errors>
  <single-error line-number= "20" object-name="ptr">
    <error-name>
      Dangling Pointer: Dereferencing pointer before initializing/allocating
    </error-name>
  </single-error>
</errors>
```

6.3 Formats Translation and Questions Generation

Translation of lines of code, explanations, and program output elements is essentially XML element name-matching. Providing the questions, actions, and execution sequence is more complicated. Problets generate an explanation for every line in the order of program execution. The order of these explanations is used to infer program execution sequence. Explanations provided by the server are also processed to identify actions to be taken. `Event-type` attribute of each `explanation/event` element is used here. Each output element provided by the server contains text

printed on the console while the program executes. This text is translated into a question about output that a particular line will generate. *CV* does not use `errors` element at this moment. Therefore, *FT* discards it during the translation process. This element may be used in the future to generate questions about bugs in the provided code.

7 Conclusions and Future Work

This is a proof-of-concept system. In order to extend it to cover more programming constructs we plan to (a) identify objects and events that are of interest in various programming contexts (e.g. building a dynamically linked list) and (b) design a general, flexible, and extensible protocol to transmit this information from *CP* to *CV*.

Further, we plan to extend that protocol to adapt visualizations to the needs of the learner, i.e. focus on visualizing parts of material that are not yet well understood. We plan to represent the learner's progress in terms of concepts (e.g. *for-loop* or *variable-declaration*). Assigning each "interesting event" a set of concepts would be tasked to *CP*. *VASP* would consult the student model service to retrieve information about learner progress within each of those concepts. On the presentation side, as the learner interacts with the visualization (i.e., observes animations, reads explanations, answers questions, etc.) *CV* would record their progress through the student model service.

Another direction would be to develop an authoring tool to help teachers create visualizations easier. Direct authoring can be an alternative to creating automated content provision services. Such "hand-crafted" visualizations could then be accessed from a Web-enabled repository and presented by *CV*.

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. 0426021.

References

[1] Peter Brusilovsky and H.-D Su. Adaptive visualization component of a distributed web-based adaptive educational system. *Intelligent Tutoring Systems*, 2363:229–238, 2002.

[2] G. Dancik and Amruth N. Kumar. A tutor for counter-controlled loop concepts and its evaluation. *Frontiers in Education Conference (FIE 2003)*, 2003.

[3] Amruth N. Kumar. Model-based reasoning for domain modeling in a web-based intelligent tutoring system to help students learn to debug C++ programs. *Intelligent Tutoring Systems (ITS 2002)*, pages 792–801, 2002.

[4] Amruth N. Kumar. Explanation of step-by-step execution as feedback for problems on program analysis, and its generation in model-based problem-solving tutors. *Instruction, Cognition and Learning (TICL) Journal*, page to appear, 2005.

[5] Thomas L. Naps, J. R. Eagan, and L. L. Norton. JHAVE - an environment to actively engage students in web-based algorithm visualizations. *ACM SIGCSE bulletin*, 32(1):109–113, 2000.

[6] Thomas L. Naps, Guido Roessling, Peter Brusilovsky, J. English, D. Jark, V. Karavirta, C. Leska, M. McNally, A. Moreno, R. J. Ross, and J. Urquiza-Fuentes. Development of XML-based tools to support user interaction with algorithm visualization. *ACM SIGCSE bulletin*, 37(4):123–138, 2005.

[7] H. Shah and Amruth N. Kumar. A tutoring system for parameter passing in programming languages. *ACM SIGCSE bulletin*, 34(3):170–174, 2002.

[8] W3C. XSL Transformations (XSLT) Version 1.0. http://www.w3.org/TR/xslt, 1999.