Publication P4

Thomas Naps, Guido Rößling, Peter Brusilovsky, John English, Duane Jarc, Ville Karavirta, Charles Leska, Myles McNally, Andrés Moreno, Rockford J. Ross, and Jaime Urquiza-Fuentes. 2005. Development of XML-based tools to support user interaction with algorithm visualization. SIGCSE Bulletin, volume 37, number 4, pages 123-138.

© 2005 by authors

Development of XML-based Tools to Support User Interaction with Algorithm Visualization

Thomas Naps
U Wisconsin Oshkosh
naps@uwosh.edu

Guido Rößling
TU Darmstadt, Germany
roessling@acm.org

Peter Brusilovsky
University of Pittsburgh
peterb@mail.sis.pitt.edu

John English
University of Brighton, UK
J.English@bton.ac.uk

Duane Jarc
University of Maryland University College
jarc@nova.umuc.edu

Ville Karavirta
Helsinki University of Technology, Finland
vkaravir@cs.hut.fi

Charles Leska
Randolph-Macon College
cleska@rmc.edu

Myles McNally
Alma College
mcnally@alma.edu

Andrés Moreno
University of Joensuu, Finland
amoreno@cs.joensuu.fi

Rockford J. Ross
Montana State University
ross@coe.montana.edu

Jaime Urquiza-Fuentes
Universidad Rey Juan Carlos, Spain
jaime.urquiza@urjc.es

ABSTRACT

As a report of a working group at ITiCSE 2005, this paper represents a vision of the use of XML specifications and tools in algorithm visualization, particularly with regard to supporting user interaction. A detailed description is given of how an interesting event to be visualized is decomposed, combined with interactive questions, narratives, control flow code and metadata, and finally rendered into graphical primitive and transformation specifications. The heart of the paper is our discussion of XML specifications for content generation (the object being visualized), interactive questions, and graphical primitives and transformations, with briefer discussions of narratives and metadata. Examples are provided for each in an appendix, with fuller details to be published on an associated website that we hope will become a source of future standards in this area. In conclusion, the approach of the working group is discussed, and important remaining challenges are identified.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer & Information Science Education - *Computer Science Education*

General Terms

Algorithms

Keywords

Visualization, Animation, Pedagogy, XML

1. INTRODUCTION

This paper is the result of the working group that met at ITiCSE 2005 to discuss and develop XML-based tools for algorithm visualization. The goal of this ongoing working group is the development of a standard framework that will help promote the design, implementation, and use of educational visualization systems. This is to be accomplished by providing XML definitions of a number of elements intrinsic to visualization systems, along with a vision of how these XML definitions could evolve and be used both in existing and newly developed visualization systems.

As a start, we have provided examples of XML specifications for the following:

- objects, which correspond to high level constructs (e.g., data structures) that are most often the focus of a visualization;
- graphical primitives, such as squares, circles, lines, and so forth, from which more complex objects can be constructed and animated;
- transformations on graphical primitives, such as the scaling, rotation, and translation of graphical primitives displayed on a screen;
- narration (e.g., text, graphics, and audio) that may be attached to particular events that occur in an animation;
- questions (e.g., quiz questions) with hints and feedback that may be inserted in various places in an animation;
- metadata that can, for example, be used to describe content or log how a visualization system is being used by a student.

We have also attempted to provide guidance on

- how more complex XML definitions can be constructed, thereby composing lower-level events into a hierarchical structure that allows a student to view interesting events at a specified level of detail;
- how the XML definitions provided by the working group might be used in the many disparate visualization systems already in existence.

We want to emphasize that the XML definitions and the overall structure of the report are meant, at this point in time, to be neither comprehensive nor exclusive. It is hoped that this effort will remain dynamic and that others working in the development of visualization systems will contribute to the work started here. This would lead to the refinement and extension of the XML definitions proposed here, with the goal of soon providing a set of standards that will help further promote work in educational visualization.

1.1 Background

Educational visualization systems have been around for many years. The most common are *algorithm animators*, although various other types of visualization systems exist. These include *program execution animators*, which elucidate the execution of a source program in action, *concept animators* that illustrate (often abstract) concepts, such as the working of finite state automata, and others.

Examples of current, large-scale algorithm animation systems are *Animal* [22], *JAWAA* [1], *JHAVÉ* [18], and *MatrixPro* [12]. Other well-known algorithm animation systems from the past include *Samba* [27], *Tango* [28], and *Zeus* [5]. Unfortunately, the roads leading to the development of useful, large-scale animation systems have often led to dead ends, and some systems have withered and fallen into disuse. The primary reason for their demise was platform dependence—early systems could only run on specialized hardware and with the support of software that is now no longer in widespread use (for example, special graphics terminals that ran X-Windows under Unix).

Platform independence was an elusive target until the advent of the Web, browsers, Java, and the Java Virtual Machine. Most current, large-scale visualization software systems are written in Java (or some other language that compiles to Java bytecode), which allows them to be run on most hardware/operating system platforms.

However, many visualization systems still suffer from a few system dependencies that preclude work done by one designer to be used by others [19]. First, the internal representations of structures being animated are generally both non-standard and program-dependent. Second, it is likely that many of the internal representations used by existing systems are tightly coupled with the visualization software itself. By providing XML definitions of the essential elements of a visualization, we believe that both the aspects of program dependency and that of tight coupling of the animation objects with the visualization software can be eliminated. We hope that this will lead to easier and more widespread development of effective visualization systems, as well as a cross-fertilization that will enhance the quality of existing systems.

1.2 A Note

It should be noted that visualization software falls into many different categories. In order to better understand the focus of this paper, consider the following two:

- generic, event driven visualizations using primitive graphical operations;
- specific, model-based visualizations using simulation.

Algorithm animators generally fall into the first category. That is, the effect of algorithms operating on data structures is displayed through the use of generic low-level operations that are not tied to the algorithm itself—elements are placed on the screen, colored, rotated, scaled, and translated in a manner that is independent of the algorithm being visualized. The same objects and operations used to animate the actions of one algorithm can be used for animating other algorithms as well. The underlying software support structure for accomplishing the visualization can be used to animate topics in many, diverse subject areas.

Model-based animators, such as those animating the execution of a finite state automaton (for example, *JFLAP* [9] and the hypertextbook work at `http://www.cs.montana.edu/webworks/projects/theoryportal`), are generally model driven. The concrete input structure to the animator is usually a more conceptual entity (such as a finite state automaton specified in an XML file) which is then simulated by the visualization software. Although the actions of a finite state automaton *could* be visualized in simple cases with the same primitive objects and transformations used by algorithm animators, the capablities of the visualization software would be limited by not being aware that the object being animated *is* a finite state automaton. When the input structure is known to be a finite state automaton, all of the theory about finite state automata can be brought to bear on the visualization system. For example, this would allow the visualization system to animate the algorithm for converting a nondeterministic finite state automaton into an equivalent deterministic machine.

In both cases, XML can be used to define the underlying structures that are used by the visualization software. In the case of the finite state automaton animator, the XML input would be based on a standard DTD for finite state automata.

This paper does not provide XML definitions for model-based visualizations. These would generally require XML definitions of each model to be visualized through simulation, which, although challenging in their own right, are quite straightforward. Rather, the working group restricted its efforts to the first category of visualizations of the generic, event driven type.

2. PROPOSED ARCHITECTURE

The visualization environment described here is conceptual in nature and consists of a number of interacting components. Designing XML specifications for the data processed by these components allows for increased interoperability.

2.1 Data and processing components

We work from the premise that any visualization depicts a stream of interesting events that occur on an object (for example, a data structure). We break down the architecture of such a system into three components—an *elaborator*, a *synchronizer*, and a *graphic decorator*. As depicted in Figure 1, each component is responsible for adding certain data to the specification that is ultimately provided to the *visualizer*.

The elaborator combines an interesting event (for example, *insert 6 into a binary search tree*) with an object (that is, the tree itself) on which it occurs to produce a specification called the *interesting-event-with-object*.

The synchronizer augments the *interesting-event-with-object* by adding specifications that can be used to include pedagogical hooks—interactive questions, narrations (textual, audio, and hyper-media), and so forth—that will engage students, requiring them to be more than passive viewers of the visualization. We now have a specification called *interesting-event-with-object-and-pedagogical-information*.

The graphics decorator adds the information that will determine the visual layout (geometry, color, and so forth) of the rendering that the student will see. After the graphics decorator has done its job, we have a specification consisting of *interesting event with object, pedagogical information, and graphic information*.

Unwieldy terminology? Without a doubt! So, henceforth we will refer to the *interesting event with object, pedagogical information, and graphic information* as the *complete visualization specification*. The complete visualization specification is given to an *adapter*, which determines how a particular visualization system displays everything on the screen. Figure 1 illustrates this architecture. Note that the figure is intended to convey the group's shared understanding that our description of what is done by the elaborator, synchronizer, and graphics decorator is not meant to imply that their actions must occur in a "pipeline". Rather our nomenclature is used to emphasize a breakdown of the roles of the various information specifications that are processed. In any particular implementation of a visualization system, the actions of the elaborator, synchronizer, and graphics decorator may often be interleaved.

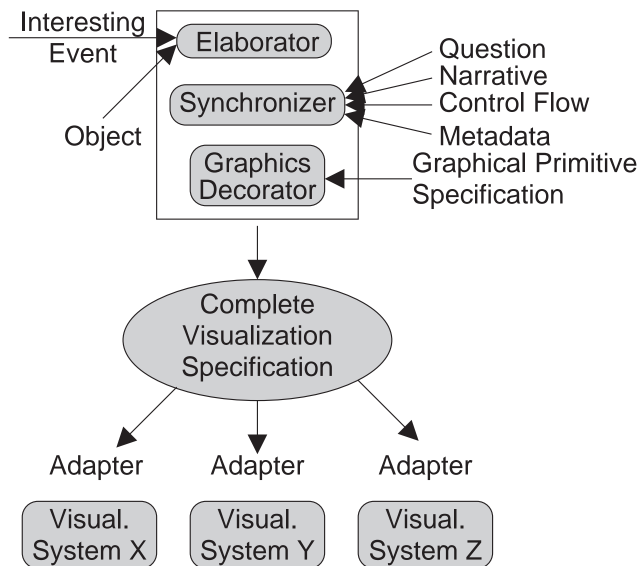

Figure 1: An overview of "The Big Picture"

A more detailed description of each of the information specifications used by these components follows.

- At the highest-level is the *interesting event* specification. Here, to allow a variety of perspectives, we purposefully avoid a detailed definition of what an interesting event is—the interested reader may consult [5, 8]. Instead we informally view it as a conceptual-level action upon an object that can potentially be visualized. In traditional computer science applications, these events may be thought of as operations on a data structure, although one can also envision objects such as circuit diagrams being displayable in this model. Events can be hierarchically organized where an upper-level event includes a series of lower-level events. Each event has a unique ID that allows an association of the event with other specifications such as objects (data structures), questions, narration, control flow, metadata, and graphical primitives. It is assumed that each event in the stream of events is produced by the execution of an algorithm at some point in time, but it can also be produced by a visualization author (possibly with the use of some tools).

- The *object* specification associates a concrete and possibly high-level data structure with an interesting event that occurs on the data structure during the algorithm's execution.

- The *question* specification can associate one or more interactive questions (for example, prediction-style questions [2, 11]) with selected events.

- The *narrative* specification can provide "explanations" for selected events. The simplest type of explanation is just a fragment of text. However, we anticipate that narratives will incorporate audio (multimedia) comments, images, and/or links to URLs.

- The *control flow* specification associates a line or fragment of code in the program (or pseudocode) with each event to be visualized.

- The *metadata* specification can provide additional information related to authoring, content, student identification, and so forth.

- The *graphical primitives* specification provides geometric, transformation, color, and font information to the graphics display.

Although this report will focus on discussing specifications for the types of data in the above list, we do not mean to imply that this list is necessarily exhaustive. We can envision (and indeed hope) that other interested parties outside the working group will provide details for additional specifications that would allow the designer to extend the value of visualization even further.

The hierarchical structure of the interesting event referred to in Figure 1 means, for features that are supported, that the system can make a decision about the level of detail to be presented in a visualization. A visualizer could be programmed to selectively ignore lower-level events in a hierarchy and provide visualization only at the level of higher-level events. It could also be configured to present only some of the included questions and narrations. A student could control the level of visualization by requesting more or less detail. One could also develop an intelligent or adaptive system that monitors the level of student knowledge and choose the level of visualization, proper questions, and narrations adaptively.

2.2 The Role of an Adapter

Although it is possible that a visualizer could directly understand all the semantics of a complete visualization specification given to it, the architecture of many existing systems will not be suited to appropriately handling all of this information. Modifying such existing visualization systems to interpret an XML-based definition of this information will no doubt require *adapters*, interfaces that transform the complete visualization specification into the internal data format used by that system. Such adapters may, in some instances, choose to ignore some of the information in the complete visualization specification, although doing so will obviously cause that system to not be able to completely display the visualization and all of its associated pedagogical hooks. In this situation, visualization system X would merely display its "best possible approximation" to a complete visualization specification that was perhaps originally intended for a different system Y.

An obvious strategy for developing such an adapter is to define an API that allows the adapter to access only the components of the complete visualization specification in which it is interested. Such an API would make it unnecessary for the writer of an adapter to traverse a large XML parse tree, ignoring nodes in which the adapter was not interested. This could greatly facilitate the adaptation of this model, or parts of it, for an existing visualization system that does not presently use XML-based specification.

2.3 Illustrative scenarios

To help clarify the architecture described in Figure 1, we describe five scenarios of an instructor and/or student using such a visualization system.

First scenario:

1. The instructor executes an algorithm (from a provided library) to produce the interesting events for a specific example. The algorithm may also produce the control flow that synchronizes the events with the line of the code (or pseudocode) associated with this event.

2. A special editor is used by the instructor to extend this animation stream with narrations and questions that form two additional specifications to be merged into the visualization by the synchronizer.

3. The whole product is packaged into the complete visualization specification, ready for the visualizer to use.

4. At runtime, the system runs the visualization for the student, allowing the student to play the visualization in both the forward and backward directions.

Second scenario:

An alternative scenario, but one that is likely to find frequent use, is the direct production of the visualization by the author.

1. Using an editor, an instructor composes a set of events from an existing library of higher-level events.

2. The elaborated events are further annotated with narrations and, if desired, questions by the synchronizer.

3. At the end of this process, the complete visualization specification is interpreted by the visualizer directly, without the need for an adapter.

Third scenario:

This is a typical scenario that is appropriate for program visualization as well as algorithm visualization.

1. The program that produces a stream of events to be visualized is actually being used by the student at runtime.

2. The algorithm can request input from the student (for example, which value to insert into a tree). This input will dictate what the student will consequently see.

3. The program produces a stream of events in the standard format that is then immediately processed by the elaborator, synchronizer, and graphics decorator to be rendered.

Typically, this "on the fly" approach does not easily allow for the creation of narrations or questions. However, a more advanced visualization program could generate at least three specifications—events, control flow, and narrations—as shown in Kumar's "problets" [14].

Fourth scenario:

This scenario describes a client-server approach, much like that used in JHAVÉ [17, 18].

1. The student, running the client, contacts the server and requests a visualization for a particular algorithm, supplying the algorithm with any input that it needs.

2. The server executes an algorithm (from a library of existing algorithms) to produce the interesting events for this example. The server then invokes its elaborator and synchronizer to send an XML specification to the client.

3. The client renders the received XML specification by applying its graphics decorator, whence the complete visualization specification is produced and displayed. The student then engages with the visualization to study the algorithm.

Notice that, in this scenario, the information-processing components in Figure 1 will likely reside on different computers.

Fifth scenario:

In this scenario, the visualization is produced by the student and submitted for automatic or instructor grading, such as in [13, 30]. We expect this scenario to become increasingly popular as teachers attempt to engage students to work with visualizations [19].

1. The student works with a data structure to solve a problem that requires generating a correct sequence of actions to achieve a desired effect. These actions are recorded as a trace of interesting events with objects.

2. The trace is immediately sent to the graphics decorator to visualize the actions, but it is also saved in XML format to be graded.

3. The instructor can then read the saved trace and replay the sequence of actions executed by a student. The instructor can explore this trace forwards and backwards.

4. Any corrections and messages that the instructor wants to deliver to the student are added to the narrative flow synchronized with the original specification.

5. The student then uses a similar tool to analyze the solution that has now been enhanced with the instructor's narrative comments.

For each of the scenarios above, appropriate APIs can allow content authors to produce a unique animation or an interactive animation simply as a piece of normal code extended with API calls to produce visualizable events. The larger the library of supported animation events, the easier the job of the author.

2.4 Semantic events and graphical primitives

The basic goal of the events specification produced by the elaborator is to provide a semantic level description that is close to the teacher's and student's perceptions of the operation. This semantic level would operate with objects and actions. The objects could range from lower-level objects, such as a simple variable in a program visualization, to a complex data structure in an algorithm visualization. Each object would have a set of "eligible actions" that could be performed upon it. For example, an array could allow its elements to be compared or swapped, have a sub-array sorted, or simply have its elements highlighted. A variable could have its value assigned, changed or used. To maintain consistency, the community should develop an ontology of objects and actions to be supported by the shared framework (and ideally to be understood by the elaborator component with no additional support).

A *content generator* (which could be an executing algorithm or a user) could generate its content on multiple levels of detail. For example, it could generate a "sub-array sorted" action and then generate a sequence of compare and swap actions. Moreover, a swap action could be decomposed into three variable assignment actions. Each content generator could choose the depth of definition of the actions it generates.

The goal of the elaborator component in Figure 1 is to interpret the conceptual events and link them to the objects they manipulate in the form that the visualization system could understand at the time the display is actually rendered. To see why this, in varying degrees of semantic detail, may be necessary, consider the following situation. A student is viewing a visualization that displays an AVL-tree, and is asked a question that requires clicking on the tree node (circle) at which a rotation would occur when the data item 42 is added to the tree. For the visualization system to determine whether the student has correctly answered this question, it would need to have considerable semantic information associated with the circles that are rendered on the display. Each of these circles would actually represent a tree node, so the complete visualization specification would need to define the context of each particular graphic circle in the AVL-tree. Without this contextual semantic information, the student's response could not be evaluated by the system. It is certainly possible that not all implementations of a visualization system will be able to take advantage of this rich semantic detail—those that do not would simply not be able to handle questions of the form described above.

We can anticipate at least three different variations on the connection between events produced by the elaborator and graphical primitives rendered in the display of the system.

1. *The lowest level of event hierarchy produced by the elaborator is directly supported by the rendering engine.* In that case, to visualize a lowest level event, the graphics decorator described in Figure 1 would have little to do; it would simply pass on the proper directions to the rendering engine. To visualize a higher-level event, the graphics decorator would pass a sequence of lower-level events to the actual renderer that the renderer would subsequently interpret.

2. *Two or more levels in the event hierarchy produced by the elaborator are supported directly.* For example, the visualizer could swap array elements (as a single basic action), or it could show the three variable assignments done in a swap. In this case, the graphics decorator in Figure 1 would augment the conceptual-level actions with graphical primitives on the level currently used for visualization. To visualize higher-level events, the rendering engine would be instructed to show an element swap. To visualize lower-level events, the renderer would be instructed to show individual variable assignments.

3. *The lowest level of event hierarchy produced by the elaborator is above the level supported by the graphical primitive specification.* Suppose, for example, that the lowest level event generated were adding an element to a queue, and the graphical specification language did not have such a primitive. In that case, the graphics decorator could specify the operation, using lower-level graphical primitives that are supported. Once defined, this definition could be used by others who wanted to use it in the future.

3. CONTENT GENERATION

3.1 Motivation

When performing an animation there is typically a central object. In this section, we will focus exclusively on algorithm animation, so the objects are data structures. An animation uses a sequence of snapshots, which capture the states of a data structure. The XML defining a snapshot may contain the operation being animated. In this section, we discuss and illustrate XML representations for several data structures. We do not endeavor to create a comprehensive collection of data structures, but limit ourselves to focusing on a few of the fundamental ones.

3.2 What does it look like?

The XML specification for *stack* and *tree* below use the following tags:

- *struct* for describing the data structure used in an animation.

- *operation* for describing an interesting event on a data structure.

- *node* for defining an element in the data structure.
- *param* for describing graphical decorations to a node.

We begin with a very simple example of an XML representation of a stack. The following XML is a snapshot resulting from pushing 2 onto a stack implemented with a linked list that already contains two values.

Listing 1: Stack Snapshot

```
<struct type="list" name="stack">
 <operation>push</operation>
 <node value="2" isOpParam="true" />
 <node value="6" />
 <node value="3" />
</struct>
```

Notice that the representation contains an <**operation**> tag. Our motivation for including the tag is that the representation has to be general enough that a variety of animation systems can interpret it. For example, a system that had built-in support for a stack and its operations might require only the operation *push* and the data value 2. The value that is to be pushed is contained in the <**node**> tag that is annotated with the attribute *isOpParam* set to *true*. However, a visualization system that only supported primitives might ignore the operation and require all the data. In that case it would use the node values listed beginning with the value previously at the top.

To illustrate a subsequent snapshot, consider the result of popping the stack above. The representation after the pop looks as follows:

Listing 2: Stack after pop

```
<struct type="list" name="stack">
 <operation returnVal="2">pop</operation>
 <node value="6" />
 <node value="3" />
</struct>
```

A visualization designer may wish to append additional information about the rendering of the object, such as color. Additions of this type take the following form for our stack example:

Listing 3: Stack with rendering information

```
<struct type="list" name="stack"
        color="red">
 <operation>push</operation>
 <node value="8" isOpParam="true">
  <param label="top" symbol="arrow" />
 </node>
 <node value="6" />
 <node value="3" />
</struct>
```

In this case, the stack elements would be rendered in red and the node containing 8 would have a label of "top" with an arrow from the text to the node.

As an example of a nonlinear data object, consider the binary search tree shown in Figure 2, associated with the code from Listing 4. To allow for trees with any number of children, the ordinal position of the child is specified. In the case of a binary tree, the left child would be designated as ordinal position 1 and the right child as position 2.

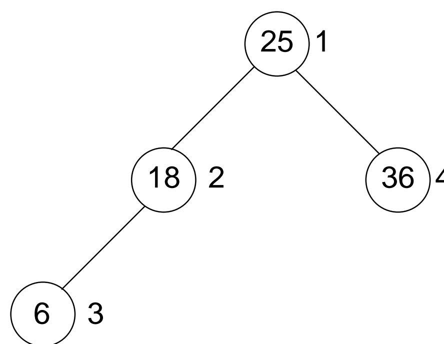

Figure 2: The tree described in Listing 4; the number to the right of each node is its id

Listing 4: XML for a tree

```
<struct type="tree" kind="binary" root="1">
 <operation>insert</operation>
 <node value="25" id="1" parent="-1"
       childOrd="1"/>
 <node value="18" id="2" parent="1"
       childOrd="1" />
 <node value="6"  id="3" parent="2"
       childOrd="1" />
 <node value="36" id="4" parent="1"
       childOrd="2" isOpParam="true"/>
</struct>
```

To illustrate the tree after an insert, the branch followed during the search of the tree is colored green, a new node containing 8 is added and an arrow is inserted to point to the inserted node. The resulting XML specification is shown as Listing 8 in the appendix.

The excerpts from an example of a graph specification in Listing 5 use the following additional tags:

- *edge* for defining an edge between nodes in the data structure.
- *nodeParam* for defining an attribute of a node, such as whether it has been visited, that a visualization designer wants to appear in an animation.
- *edgeParam* for defining an attribute of an edge, such as its weight.
- *npVal* for specifying the value of a node parameter as defined in the <**nodeParam**> tag.
- *epVal* for specifying the value of an edge parameter as defined in the <**edgeParam**> tag.

The snapshot of the more complicated structure represented by this listing is one event in the execution of Dijkstra's single source shortest path algorithm. The visualization designer wants to see as part of the animation an indication of each node's predecessor on the path and whether the node has been visited, as well as the weight associated with each edge.

To indicate that the designer wants to annotate the visualization with the predecessor node *id* and whether the node has been visited (using a v for visited and a u for unvisited), one adds the node parameters *pred* and *visited* to

the XML. Similarly, *weight* is added as an edge parameter. The <**npVal**> tag contains the actual values (in sequence) for the two node parameters and the <**epVal**> tag contains the value of the edge weight.

Listing 5: Graph XML structure

```
<struct type="graph">
 <operation>findNextNode</operation>
 <nodeParam>pred</nodeParam>
 <nodeParam dir="45" color="magenta">
   visited</nodeParam>
 <edgeParam>weight</edgeParam>
 <node id="0">
  <npVal>-1</npVal>
  <npVal>v</npVal>
 </node>
 ... <!-- see Listing 9 on page 135 -->
 <edge initNode="0" endNode="1">
   <epVal>7</epVal>
 </edge>
 ... <!-- see Listing 9 on page 135 -->
</struct>
```

For the full graph XML code, see Listing 9 on page 135.

The selected examples do not reveal the entire intended data structure DTD. Nor do our examples necessarily indicate the final form of the DTD. Each DTD is intended to be extensible and is likely to undergo modification.

4. INTERACTIVE QUESTIONS

4.1 Motivation

Interactive questions provide a mechanism for the active engagement of learners. They provide a means by which understanding can be confirmed before proceeding further. Such questions will frequently be used formatively, but there is no reason why they cannot also be used to generate marks to be used summatively. They can also provide a mechanism for adaptive learning: if a learner gets the answer to a particular question wrong, it presumably indicates a lack of understanding. If questions are annotated with metadata describing the topics of the question and the relative difficulty of the question, a visualizer can track the levels of learning attainment of individual learners with respect to individual topics. A mechanism to provide feedback based on the answers submitted by learners is also desirable for facilitating learning.

4.2 How do we achieve what we want?

Various types of question are possible. Multiple-choice questions (where a single answer is selected from a number of possible answers) are probably the most heavily used, but other possibilities include:

- Multiple select questions, which are like multiple-choice questions, but which may have several correct answers which must all be selected by the user.

- Value-entry questions, for which no predefined answers are provided. The user enters some text that is matched (with various different degrees of permissiveness) against the expected answer.

- Fill-the-gap questions, for which a list of terms is provided to be used to fill a series of gaps in a body of text.

- Point-and-click questions, for which a graphical object is identified visually by pointing and clicking on an image map.

These question types are not entirely unrelated. A fill-the-gap question might require one or more values to be entered by the user rather than being selected from a list, and a fill-the-gap question as described above can be considered as a group of related multiple-choice questions. Similarly, a point-and-click question can be regarded as a form of multiple-choice or fill-the-gap question based on a graphical form of user interaction. However, due to the differences in the intended rendering of these different question types, it seems useful to retain the distinction between them in order to leave the author with a choice as to which question type to use (and hence which rendering the visualizer will use).

The *Trakla2* system [16] uses a somewhat different approach to implementing interactive questions. It is based around a visualizer implemented as a Java applet. Each exercise is implemented as a Java class, which provides a data structure that is filled with randomized values. The learner is asked to show how the data structure is transformed as the result of a sequence of operations (for example, by inserting values into a balanced tree). The learner interacts with the applet by dragging and dropping the visual components it provides (as well as additional operations such as "delete"), and a serialized form of the resulting data structure at each stage is compared with a model solution.

The advantage of this approach is its flexibility; the primary drawback is its reliance on a specific applet, which makes it a very system-specific solution. We have deliberately avoided including drag-and-drop as a separate question type within our framework, as this is a style of interaction which is not supported by all user interfaces (for example, a browser without the aid of a supporting applet). However, the equivalent of dragging and dropping can be modeled as the selection of two endpoints. By allowing the user to select a node to move and a destination position by clicking twice on an image map, or by selecting objects by name from two lists of object names, it is possible to model the same type of question in a less interface-dependent way, and does not prevent a user interface that supports drag and drop from producing the two endpoint values as the outcome of such an operation.

Like many automatic assessment systems, *Trakla2* also supports randomized questions and answers. *Trakla2* is able to generate random input data for an algorithm, so the questions can effectively be randomly generated so that the data structure the learner is given will vary from one version of the question to another. Our specification also supports randomization, and, for example, allows the distractors (incorrect answers) presented in a multiple-choice question to be chosen at random from a pool of possible distractors associated with the question. Similarly, it is possible to specify a random choice of a question from a pool of questions, which may use metadata information to influence the choice of question according to the learner's perceived needs.

There already exist standards for interactive questions, notably the *IMS Question & Test Interoperability* (QTI) standard [23, 24, 25]. One possibility would be to adopt

the QTI standard for use within a visualization specification. However, the QTI standard is an extremely far-ranging standard which is probably far more general than is necessary for use as part of a visualization specification. In addition, the mechanism by which value-entry questions are formulated, as described above, is left unspecified in the QTI standard. In our case, a simple mechanism, such as matching the answer against a given regular expression, will suffice.

Another possibility is to define a specification which is more suited to our immediate needs. With this approach, if an author wants to be able to import questions from, or export questions to, a QTI-compliant assessment system, the simpler markup defined in our standard can be transformed to or from a QTI-compliant format using XSLT [6].

There are some additional complications. Questions are likely to need to refer to graphical objects being displayed by the visualizer, and so a tag that allows graphical objects to be referenced is needed. Exporting to a QTI-compliant form would still be possible if, for example, such tags were transformed into the name of the corresponding object at export time.

Our basic approach is therefore to define a special-purpose XML specification of questions and answers. For the sake of generality, the body of questions and answers can be written in XHTML [20] augmented with additional specialist tags such as the object reference tag described above. Questions can therefore include constructs such as hyperlinks, tables, images, and audiovisual streams. The standard includes metadata to allow student attainment to be tracked automatically in systems that support adaptive learning. Marking is supported by associating a value representing the marks for each possible answer, and the mark obtained is simply a total derived from the answers actually selected. Similarly, feedback is supported by associating some XHTML text with each answer, which can be displayed when the corresponding answer is selected.

It should be noted that not all animation systems will need to support all (or indeed any) of the features described here. Any features of our standard which a particular visualizer does not choose to support can simply be ignored.

The XML specification defines the following primitives for different question types:

- *select*—for questions for which multiple selections of answers are possible.
- *select-one*—for questions for which only one correct answer is possible.
- *value-entry*—for questions for which the user is requested to input a number, or a word, as the answer. Such questions could be corrected automatically by matching the given solution to a set of possible correct answers.
- *free-text*—for questions that ask students to write down a larger answer that is subsequently checked by a teacher.
- *upload*—for support of an automatic method of collecting answers. Any kind of file could be uploaded to a central server for later use.
- *click*—for questions that require clicking in certain part of the animation. We are aware of the difficulties of implementing this. However, alternative implementations could display a list of the possible choices (for example, nodes in a tree).
- *fill*—for fill-the-gap questions that contain several input boxes within a narration.

The following sub-elements are common to most of the elements specified above. Sub-element *item* defines one of the choices, *input* specifies the addition of an input box, used for *fill* and *value-entry*. Several parts of a question, for example its formulation and answers, and advanced entries such as feedback or hint, are defined similarly, but for different purposes. We therefore incorporate a general container element *contents*. The actual purpose of an element is specified in the *type* attribute. Element *contents* allows authors to include text, images, and, more importantly, the references to visual objects displayed in the animation. These references are marked inside the *contents* element as *object-ref*. The *object-ref* element has two main attributes, *objid* and *type*, where *objid* is the identification given by the "graphical engine", and *type* identifies the type of information we want to display (label, textual representation, or graphical representation).

Listing 6 shows excerpts from an example of a multiple-choice question. It will display three options, two random incorrect answers ("distractors") chosen from three, plus the correct answer with id *it3*. All options provide proper feedback. The full listing is included as Listing 12 in the appendix.

Listing 6: Example for a multiple-choice question

```
<select-one id="question1" solutionID="it3"
            random="3">
 <metadata>...</metadata>
 <contents type="label">...</contents>
 <contents type="hint">...</contents>
 <item id="it1">
  <contents type="answer">...</contents>
  <contents type="feedback">...</contents>
 </item>
 <item id="it2">
  <contents type="answer">...</contents>
  <contents type="feedback">...</contents>
 </item>
 <item id="it3">
  <contents type="answer">...</contents>
  <contents type="feedback">...</contents>
 </item>
 <item id="it4">
  <contents type="answer">...</contents>
  <contents type="feedback">...</contents>
 </item>
</select-one>
```

Multiple-selection questions will be implemented with the *select* element. This type of question can have multiple solutions, thus different grades for all possible answers could be assigned. The complete XML associated with the excerpts in Listing 6 appears in the appendix as Listing 12.

The user has to provide manual input for *value-entry* questions. The difference between this type and *free-text* questions is the correction of the given answers. While *value-*

entry answers may be corrected automatically, *free-text* submissions should be corrected by a teacher.

Fill-the-gap questions can specify the set of acceptable answers for each gap. One implementation approach is to show all terms specified in the gaps in random order. The user then has to pick appropriate terms from this list for each gap. Note, however, that this results not so much in a "real" fill-the-gap approach as in a mapping operation. Excerpts from an XML specification for an example question in Listing 7 refer to external metadata associated with the question. (The complete XML for this appears as Listing 15 in the appendix.)

Listing 7: Fill-the-gap example

```
<fill id="question2">
 <contents type="answer">...
  <input answer="data" />...
  <input answer="nodes" />...
  <input answer="edges" />...
 </contents>
 <contents type="hint">...</contents>
 <contents type="feedback">...</contents>
</fill>
```

A variety of additional examples of XML-based interactive questions appears in Listings 12-14 of the Appendix.

5. NARRATIVES

Narratives allow a visualizer to provide additional levels of explanation for a topic being presented. This may be textual, or it may be in a more complex format such as an audiovisual presentation. The content of such narratives is expressible as XHTML. Adaptive systems might want to be able to select between different narratives based on the skill level of the learner. Narratives should be closely tied to the metadata used to describe skill levels.

In a non-adaptive system where a visualisation is associated with multiple narratives, it is necessary to provide a mechanism by which the system can identify the "default" narrative to be presented. The visualizer might choose to render the default narrative or to present the learner with a menu of available narratives to choose from. Alternatively, the default narrative might be presented initially under all circumstances and the remaining narratives made available for the user to choose from by selecting from a list of titles.

The narration element shown in Listing 16 in the appendix explains that two elements will be swapped because they are unordered. Based on the metadata description of the assumed user's skill level, the information may be hidden by the system.

6. METADATA

Objects created following the XML specifications for data structures, questions, narrations, and animations lack higher order information, the metadata. Higher-order information includes data about the author, the intended audience, the required tool to visualize the object, and a number of other fields that can help to categorize the object.

Basic metadata can help in tracking the document, for example who created it and which version it is. More advanced uses for the metadata have been proposed. For example, a course can be adapted to the needs of a teacher [4] or a student [29] using the metadata corresponding to the object.

Another important reason for using metadata is to integrate the described content with current Learning Management Systems (LMS). This way, teachers can adopt already-developed animations and use them in their own courses.

We propose to let authors define metadata in two ways. First, they can include the metadata in the actual XML document that they want to categorize. For that purpose, they can use our simplified metadata specification, which contains important fields related to algorithm animation and education in general. The second approach, and the more complete, is to adopt the Learning Object Metadata specification [15]. In this case, the metadata should be stored as a separate object and referred to by the original object. Brase [3] details how to use LOM metadata in a learning context. Finally, a mixed approach is also possible. Documents can contain both types of metadata, and extend each other. General metadata should be included in the LOM document rather than in the document itself to conform to the standard.

Apart from the usual fields in metadata (for example, title and author), it is important to add educational data (for example, skill, Bloom category, and learning style) that can help teachers and agents to make decisions about whether the object suits their intentions.

The two complete examples in the appendix show how metadata fits into XML documents. Listing 17 declares basic information about author and skill level. A new namespace is created for the metadata fields, as they refer to a specification different from the one used in the document.

Finally, the declaration in Listing 18 is sufficient to refer to an external file containing the metadata. It also specifies the language of the object described by the metadata. (Note that the line breaks in the URLs in both listings in the appendix are due to formatting reasons and do *not* belong in the actual file.)

7. GRAPHICAL PRIMITIVES AND TRANSFORMATIONS

One of the contributions of this report is the start of a formulation of a set of XML definitions (DTDs and/or schemas) for the kinds of graphical "primitives" and transformations that might be used by a visualization system.

Previous work on algorithm animation has resulted in many different languages for describing graphical primitives and transformations. Some of the most well known languages are *JAWAA* [1], *AnimalScript* [21], and *Samba* [26]. Another general specification for graphical primitives is *Scalable Vector Graphics* [7], which has become popular in the last few years. Our specification, described in the next two sections, aims at combining the good features of the previous work.

7.1 Graphical Primitives

Graphical primitives are basic graphical components that can be composed to represent arbitrarily complex objects (for example, a tree data structure) and direct animation (for example, inserting a node into a tree data structure). The following have been defined:

- point, polyline, line;
- arc, ellipse, circle and circle-segment;
- square, triangle, rectangle;

- text.

Some of these objects are extensions of other objects. For example, a line is a special instance of a polyline. For ease of use, we have included these "special case" objects among the graphical "primitives."

All graphical primitives—in the following, abbreviated to "primitives"—have some shared attributes: *base-object, id, hidden, depth*, and *style*. The *base-object* attribute of a primitive is used to associate the primitive with the more complex object (for example, a tree node) to whose rendering it may belong. Every primitive can have an *id* field to be used for identification when transforming it. Primitives can be either hidden or visible through use of the attribute *hidden*, which, by default, is false. The *depth* attribute specifies the display depth of the object, needed to determine what happens if objects overlap.

Every graphical primitive can contain an element specifying the style. Additionally, primitives can reference predefined styles through the *style* attribute. For examples of styles, refer to section 7.2.

Listings 19 and 20 in the appendix provide XML examples for two representative graphical primitives. Listing 19 defines a polyline object with three nodes. Listing 20 illustrates the support for internationalization for text objects. Text can be displayed in any language by using multiple *contents* elements as illustrated. A complete list, including the definition for each primitive object, can be found at [10].

7.2 Styles

Consider cascading style sheets (CSS), which are used in conjunction with HTML documents to enforce a standard "look and feel" to an entire document without requiring the author to attend to the stylistic details on each page. In our specification, we want to have something similar to relieve the author/system of the duty of applying a particular style specification to every primitive.

Each graphical primitive is specified by several subelements or attributes. The visual appearance is further specified by a style, which can specify stylistic properties, including:

- whether it has a forward or backward arrow
- the primitive's color and fill-color (not applicable to all primitives)
- the font family, size, and style (bold, italic, or bold italic)
- the primitive's stroke type.

Not all properties are applicable to all primitive objects.

Styles can also have an identifier (*id*), that can be referenced by multiple objects. Similar to cascading style-sheets used in conjunction with HTML, styles can also be shared among objects by referring to the same style id. Local attributes can also be overwritten.

Listing 21 in the appendix specifies a style. When applied to a graphical primitive, for example *line*, this style specifies that the line should be solid (as opposed, say, to dashed), colored red, three pixels wide, and have an arrow tip in the "forward" direction. For the line object, the fill color and font size are not applicable, although they might be for other objects that reference this style definition.

To provide the power of CSS, which allows style sheets to be extended, the *style* element can specify a style to be extended, and it allows elements to be overwritten, as shown in Listing 22 in the appendix.

7.3 Coordinates

To allow for flexible and easy positioning of objects in a visualization, the DTD for primitive objects supports the specification of coordinates, either in absolute coordinates or as offsets relative to a given location. A location can be based on the *bounding box* of an object, the node of a polygon/polyline object, the last used location, or the baseline of a text component (not taking into account under- or overstrokes). Figure 3 and Listing 23 in the appendix illustrate these different approaches.

In this example, the coordinates refer to the polygon in Figure 3 (points 1 and 2) and to the text "aPoly" (point 4). Point 3 is drawn relative to point 2.

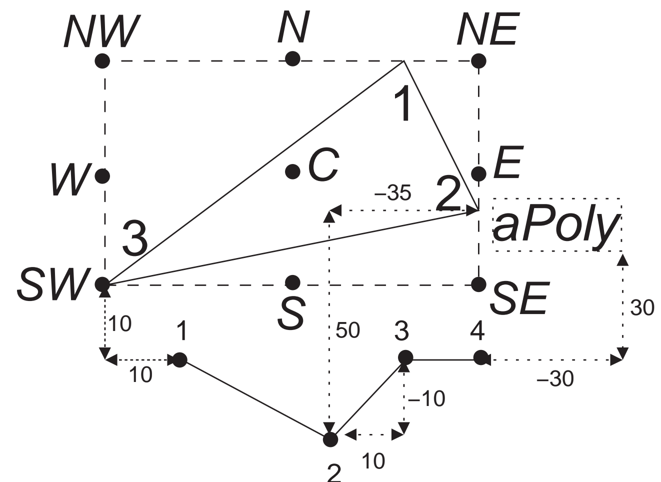

Figure 3: Example of relative coordinates supported by the DTD.

7.4 Animation Steps

The transformations described in the previous section can be combined as steps in which all the transformations comprising a step are done simultaneously. A *step* thus acts a phase of an animation that displays some single interesting event. A *frame* is similar to a frame of a movie—it represents one component in a series that is involved in visualizing a step. A step can thus be made up of many frames that, for example, illustrate the effect of a step by smoothly transitioning the visualization from the current state to the next.

A step can also contain a title and a description of the step in multiple languages. This narrative can be included in the document, or it can reference a narration from an external source. Listing 24 in the appendix illustrates this.

7.5 Transformations

The DTD for graphical primitives also covers the following transformations that change the appearance or position of a single primitive or a set of primitives:

- show/hide

- move (along a set of coordinates or following an arc or polyline object)
- rotate
- scale
- change-style
- change-property
- group/ungroup to combine a set of objects under one ID
- swap-id to swap the IDs used to reference two objects.

All transformations can possess a timing specification that can define the starting time of the transformation, its duration, or both. There are the following ways to specify timing:

- no timing definition: the transformation takes place at once and without a duration.
- delay only: a delay between the start of the current step and the start of the operation is given. The delay can be specified as a time (based on seconds or milliseconds), or on a number of preceeding animation frames. The transformation has no duration and therefore immediately completes.
- duration only: the transformation starts at the beginning of the step and takes a certain amount of time. This can also be specified by frames or milliseconds.
- duration and delay: combines the two options above.

The other tag common to all the transformations is the *object-ref* tag that refers to the objects being transformed or used to provide a given effect.

To give an example of what the actual XML for the transformations looks like, Listing 25 in the appendix shows how to specify a rotation and translation (*move*) of one object (for example, a square) along another object (for example, a line).

7.6 Shapes

The DTD also supports defining reusable shapes. So, for example, to specify the shape *house* that consists of a rectangle and a triangle, the XML in Listing 26 (see appendix) can be used.

8. AN UNFOLDING RESEARCH PROGRAM

This paper represents a snapshot of a research program that had its genesis at the ITiCSE 2005 conference in Lisbon, Portugal. The program's goal was (and is) to define and implement XML standards and associated tools that foster advances in algorithm visualization systems, particularly in the area of user interaction. To this end, a group of eleven individuals committed to being part of a working group during the conference, with the expectation that a sizable subgroup of them would then return to their home institutions and continue work. This subgroup, perhaps augmented with additional individuals, anticipates meeting again as a working group during ITiCSE 2006.

The original hope was that the result of the 2005 group labors would be a complete design document. This document would then have been used to inform implementation efforts during the following year. As can easily be seen, the present document falls short of that, and should instead be seen as a framework for future research and development. Before the working group began, it seemed that there was broad agreement on many issues. However, the challenges of defining standards and architectures while maintaining interoperability and compatibility revealed a number of evolving understandings rather than a unified vision. To the extent possible, within the context of this paper, these various understandings have been smoothed out into an apparent single vision. This belies significant areas that still require much work and thought, such as the following:

- What is the true nature of the Elaborator? How does it go about its job of decomposing an interesting event into a hierarchy of subevents? How can it be made extensible to new classes of objects and operations?
- To what level of granularity should events be decomposed? Is the nature of the object itself modified during this process?
- When in the process are the graphical primitives added? Can it be that interesting events arrive in the system with (some) graphical primitives already attached to them? Or should input consist only of what we have called semantic information?
- Should the relationship between the Elaborator, Synchronizer, and the Graphics Decorator be thought of as a data-flow pipeline, or are these activities that can be interleaved? If it is a pipeline, could an existing visualization system tap the stream before the graphics decorator does its work, receiving only semantic information about the object being visualized?

So, beyond the basic challenge of fully developing the XML specifications begun in this paper, there are serious conceptual issues remaining to be tackled. The group will tackle these issues before it reconvenes in 2006.

9. CONCLUSIONS

In this paper, we have developed a vision of how XML standards and tools could be used to enhance algorithm visualization. Concrete examples of potential XML use in visualization systems were given, and an overall framework for visualization systems was developed. The XML specifications themselves—for objects, interactive questions, narratives, graphical primitives and the like—can be adapted for use in existing visualization systems. The framework envisioned defines a direction for future research and development, and raises a number of interesting issues in visualization system design.

The website [10] is associated with the larger research project referenced in the last section. Please consult it for the forthcoming DTD specifications for the examples given in the paper, and (as research continues) additional XML-based tools which support user interaction with algorithm visualization.

10. REFERENCES

[1] Akingbade, A., Finley, T., Jackson, D., Patel, P., and Rodger, S. H. JAWAA: Easy Web-Based Animation from CS 0 to Advanced CS Courses. In *Proceedings of the* 34*th* *ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2003), Reno, Nevada* (2003), ACM Press, New York, pp. 162–166.

[2] Anderson, J. M., and Naps, T. L. A Context for the Assessment of Algorithm Visualization System as Pedagogical Tools. *First International Program Visualization Workshop, Porvoo, Finland. University of Joensuu Press* (July 2001), 121–130.

[3] Brase, J. *Usage of Metadata*. PhD thesis, University of Hannover, Hannover, Germany, 2005.

[4] Brase, J., Painter, M., and Nejdl, W. Completing LOM - How Additional Axioms Increase the Utility of Learning Object Metadata. In *3rd IEEE International Conference on Advanced Learning Technologies (IEEE ICALT 2003), poster session* (2003), IEEE Press, p. (poster session). `http://www.kbs.uni-hannover.de/Arbeiten/Publikationen/2003/icalt03_lon%g.pdf`.

[5] Brown, M. H. Zeus: A System for Algorithm Animation and Multi-View Editing. *Proceedings of the 1991 IEEE Workshop on Visual Languages, Kobe, Japan* (Oct. 1991), 4–9.

[6] Clark, J. XSL Transformations (XSLT), version 1.0, 1999. `http://www.w3.org/TR/xslt/`.

[7] Ferraiolo, J. Scalable Vector Graphics (SVG) 1.0 specification. `http://www.w3.org/TR/SVG`, 2001.

[8] Henríquez, L. M. G. Software Visualization: An Overview. *Informatik / Informatique, Special Issue on Visualization of Software* (Apr. 2001), 4–7.

[9] Hung, T., and Rodger, S. H. Increasing Visualization and Interaction in the Automata Theory Course. In *Proceedings of the* 31*st* *ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2000), Austin, Texas* (2000), ACM Press, New York, pp. 6–10.

[10] ITiCSE 2005 AV XML Working Group. Algorithm Visualization XML Specifications, 2005. `http://www.algoanim.net/xmlspec`.

[11] Jarc, D., Feldman, M. B., and Heller, R. S. Assessing the Benefits of Interactive Prediction Using Web-based Algorithm Animation Courseware. *Proceedings of the* 31*st* *ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2000), Austin, Texas* (Mar. 2000), 377–381.

[12] Karavirta, V., Korhonen, A., Malmi, L., and Stålnacke, K. MatrixPro - A Tool for Ex Tempore Demonstration of Data Structures and Algorithms. In *Proceedings of the Third Program Visualization Workshop, University of Warwick, UK* (July 2004), pp. 27–33.

[13] Krebs, M., Lauer, T., Ottmann, T., and Trahasch, S. Student-Built Algorithm Visualizations for Assessment: Flexible Generation, Feedback, and Grading. In *Proceedings of the* 10*th* *Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE 2005), Monte de Caparica, Portugal* (2005), ACM Press, New York, NY, pp. 281–285.

[14] Kumar, A. Problets - The Home Page, 2005. `http://phobos.ramapo.edu/~amruth/grants/problets/home.html`.

[15] Learning Technology Standards Committee of the IEEE. Learning Object Metadata (LOM). `http://ltsc.ieee.org/doc/wg12/LOM-WD3.htm`.

[16] Malmi, L., Karavirta, V., Korhonen, A., Nikander, J., Seppälä, O., and Silvasti, P. Visual Algorithm Simulation Exercise System with Automatic Assessment: TRAKLA2. *Informatics in Education Volume 3 3*, 2 (2004), 267–288. `http://www.vtex.lt/informatics_in_education/htm/INFE048.htm`.

[17] Naps, T. JHAVÉ – Addressing the Need to Support Algorithm Visualization with Tools for Active Engagement. *IEEE Computer Graphics and Applications*, 6 (Dec. 2005), (to appear).

[18] Naps, T., Eagan, J., and Norton, L. JHAVÉ: An Environment to Actively Engage Students in Web-based Algorithm Visualizations. In *Proceedings of the* 31*st* *ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2000), Austin, Texas* (Mar. 2000), ACM Press, New York, pp. 109–113.

[19] Naps, T. L., Rößling, G., Almstrum, V., Dann, W., Fleischer, R., Hundhausen, C., Korhonen, A., Malmi, L., McNally, M., Rodger, S., and Velázquez-Iturbide, J. Á. Exploring the Role of Visualization and Engagement in Computer Science Education. *ACM SIGCSE Bulletin 35*, 2 (June 2003), 131–152.

[20] Pemberton, S. e. a. XHTML 1.0 The Extensible HyperText Markup Language, Aug. 2002. `http://www.w3.org/TR/xhtml1/`.

[21] Rößling, G., and Freisleben, B. AnimalScript: An Extensible Scripting Language for Algorithm Animation. In *Proceedings of the* 32*nd* *ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2001), Charlotte, North Carolina* (Feb. 2001), ACM Press, New York, pp. 70–74.

[22] Rößling, G., and Freisleben, B. Animal: A System for Supporting Multiple Roles in Algorithm Animation. *Journal of Visual Languages and Computing 13*, 2 (2002), 341–354.

[23] Smythe, C., Shepherd, E., Brewer, L., and Lay, S. IMS Question & Test Interoperability: An Overview, Final Specification, version 1.2. IMS, Feb. 2002. `http://www.imsglobal.org/question/qtiv1p2/imsqti_oviewv1p2.html`.

[24] Smythe, C., Shepherd, E., Brewer, L., and Lay, S. IMS Question & Test Interoperability: ASI Information Model, Final Specification, version 1.2. IMS, 2002. `http://www.imsglobal.org/question/qtiv1p2/imsqti_asi_infov1p2.html`.

[25] Smythe, C., Shepherd, E., Brewer, L., and Lay, S. IMS Question & Test Interoperability: ASI XML Binding Specification, version 1.2, 2002. `http://www.imsglobal.org/question/qtbind03.html`.

[26] Stasko, J. Using Student-built Algorithm Animations as Learning Aids. In *Proceedings of the* 28*th* *ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '97), San Jose, California* (Feb. 1997), ACM Press, New York, pp. 25–29.

[27] Stasko, J. Samba Algorithm Animation System. http://www.cc.gatech.edu/gvu/softviz/algoanim/samba.html, 1998.

[28] Stasko, J. Smooth Continuous Animation for Portraying Algorithms and Processes. In *Software Visualization*, J. Stasko, J. Domingue, M. H. Brown, and B. A. Price, Eds. MIT Press, 1998, ch. 8, pp. 103–118.

[29] Sun, S., Joy, M., and Griffiths, N. The Use of Learning Objects and Learning Styles in a Multi-Agent Education System. In *Proceedings of the World Conference on Educational Multimedia, Hypermedia and Telecommunications* (2005), AACE, Charlottesville, VA, pp. 3403–3410.

[30] Warendorf, K., and Tan, C. ADIS - An animated data structure, intelligent tutoring system or putting an interactive tutor on the WWW. In *Proceedings of Workshop "Intelligent Educational Systems on the World Wide Web" at AI-ED'97, 8th World Conference on Artificial Intelligence in Education, Kobe, Japan* (1997), P. Brusilovsky, K. Nakabayashi, and S. Ritter, Eds., ISIR, pp. 54–60. http://www.contrib.andrew.cmu.edu/~plb/AIED97_workshop/Warendorf/Warend%orf.html.

APPENDIX

Listing 8: Tree after insert operation

```
<struct type="tree" kind="binary" root="1">
 <operation>insert</operation>
 <!-- Color green added here -->
 <node value="25" id="1" parent="-1"
       childOrd="1" color="green" />
 <node value="18" id="2" parent="1"
       childOrd="1" color="green" />
 <node value="6"  id="3" parent="2"
       childOrd="1" color="green" />
 <node value="36" id="4" parent="1"
       childOrd="2" />
 <!-- New node ... -->
 <node value="8"  id="5" parent="3"
       childOrd="2" isOpParam="true">
  <!-- With arrow pointing to it -->
  <param label="inserted" symbol="arrow"
         dir="225" xoff="-50" yoff="50" />
 </node>
</struct>
```

Listing 9: Example XML for a Graph structure

```
<struct type="graph">
 <operation>findNextNode</operation>
 <nodeParam>pred</nodeParam>
 <nodeParam dir="45" color="magenta">
  visited
 </nodeParam>
 <edgeParam>weight</edgeParam>
 <node id="0">
  <npVal>-1</npVal>
  <npVal>v</npVal>
 </node>
 <node id="1">
  <npVal>0</npVal>
  <npVal>v</npVal>
 </node>
 <node id="2">
  <npVal>-1</npVal>
  <npVal>u</npVal>
 </node>
 <node id="3">
  <npVal>-1</npVal>
  <npVal >u</npVal>
 </node>
 <edge initNode="0" endNode="1">
   <epVal>7</epVal>
 </edge>
 <edge initNode="0" endNode="2">
  <epVal>5</epVal>
 </edge>
 <edge initNode="2" endNode="3">
  <epVal>2</epVal>
 </edge>
 <edge initNode="1" endNode="3">
  <epVal>4</epVal>
 </edge>
</struct>
```

Listing 10: Multiple selection question example

```
<select id="question2" random="2">
  <metadata href="elsewhere"></metadata>
  <contents type="label">In which of these
    lists, a swap operation will be performed
    in the next step? Suppose that the last
    element processed is the third one.
  </contents>

  <contents type="hint">A swap operation will
    be performed if two next elements are not
    ordered correctly.</contents>
  <item id="it1" grade="1">
    <contents type="answer">
       <object-ref objid="badlist1"/>
    </contents>
    <contents type="feedback">Wrong answer,
      this list is already ordered</contents>
  </item>

  <item id="it2" grade="2">
    <contents type="answer">
      <object-ref objid="badlist2"/>
    </contents>
    <contents type="feedback">Not bad, although
      there are unordered items, the next
      two are ordered.</contents>
  </item>

  <item id="it3" grade="4">
    <contents type="answer">
      <object-ref objid="goodlist1"/>
    </contents>
    <contents type="feedback">Great. Elements
```

```
    <object-ref objid="elem4"/> and
    <object-ref objid="elem5"/> are unordered
     and will be swapped.</contents>
  </item>
</select>
```

Listing 11: Multiple selection with id list

```
<select id="question2" random="2"
        solutionID="it3 it4">
  <metadata href="elsewhere"></metadata>
  <contents type="label">In which of these
    lists will a swap operation be performed
    in the next step? Suppose that the last
    element processed is the third one.
  </contents>

  <contents type="hint">A swap operation will
    be performed if two next elements are not
    ordered correctly.</contents>
  <item id="it1">
    <contents type="answer">
      <object-ref objid="badlist1"/>
    </contents>
    <contents type="feedback">Wrong answer,
      this list is already ordered</contents>
  </item>

  <item id="it2">
    <contents type="answer">
      <object-ref objid="badlist2"/>
    </contents>
    <contents type="feedback">Not bad, although
      there are unordered items, the next
      two are ordered.</contents>
  </item>

  <item id="it3">
    <contents type="answer">
      <object-ref objid="goodlist1"/>
    </contents>
    <contents type="feedback">Great! Elements
     <object-ref objid="elem4-1"/> and
     <object-ref objid="elem5-1"/> are unordered
      and will be swapped.</contents>
  </item>

  <item id="it4">
    <contents type="answer">
      <object-ref objid="goodlist2"/>
    </contents>
    <contents type="feedback">Great! Elements
     <object-ref objid="elem4-2"/> and
     <object-ref objid="elem5-2"/> are unordered
      and will be swapped.</contents>
  </item>
</select>
```

Listing 12: Complete XML for Multiple Choice Question example

```
<select-one id="question1" solutionID="it3"
            random="3">
 <metadata>
  <skill type="required">
   <concept>list </concept>
   <level binary="Known"
          quantitative="0.5"/>
  </skill>
  <skill type="required">
   <concept>sorting algorithms</concept>
   <level binary="Known"
          quantitative="0.7"
          qualitative="intermediate"/>
  <skill>
 </metadata>

 <contents type="label">Which of the
  following lists will be the one generated
  by the algorithm in the next step?
 </contents>
 <contents type="hint">Think about if a swap
  operation should be executed...
 </contents>

 <item id="it1">
  <contents type="answer">This one:
   <object-ref objid="goodlist"/></contents>
  <contents type="feedback">Great, the
   algorithm will execute a swap operation
   on the next step</contents>
 </item>

 <item id="it2">
  <contents type="answer">This one:
   <object-ref objid="badlist1"/></contents>
  <contents type="feedback">Wrong answer,
   two first elements were already ordered.
  </contents>
 </item>

 <item id="it3">
  <contents type="answer">This one:
   <object-ref objid="badlist2"/></contents>
  <contents type="feedback">Wrong answer,
   the algorithm can not order the last
   four elements in one step.</contents>
 </item>

 <item id="it4">
  <contents type="answer">This one:
   <object-ref objid="badlist3"/></contents>
  <contents type="feedback">Wrong answer,
   this is the original, and unordered,
   list.</contents>
 </item>
</select-one>
```

Listing 13: Example questions for value input

```
<value-entry id="question2">
  <contents type="label">Write an expression
    that after being evaluated gives as a result
    the integer value of 5.</contents>
  <input/>
  <contents type="hint">Think about arithmetic
     expressions</contents>
  <contents type="feedback">Wrong answer, refine
    your expression</contents>
</value-entry>
```

Listing 14: Free text answer example

```
<free-text id="question2">
 <metadata>
  <skill>
   <concept>tree</concept>
   <level binary="Known"
          qualitative="intermediate"/>
  </skill>
 </metadata>
 <contents type="label">Give the definition of
  a tree structure:</contents>
  <textarea/>
  <contents type="hint">A tree looks like:
    <object-ref objid="treeexample">
 </contents>
</free-text>
```

Listing 15: Complete XML for Fill in the gaps example

```
<fill id="question2">
  <contents type="answer">A tree is a
    <input answer="data"/> structure which
    is made of <input answer="nodes" />
    connected by <input answer="edges" />
  </contents>
  <contents type="hint">Remember that a list
    is a data structure made of elements,
    where the order is important</contents>
  <contents type="feedback">...</contents>
</fill>
```

Listing 16: Example for an English narrative

```
<narrative id="doc1" default="true"
       title="Swapping unordered items">
  <metadata>...</metadata>
  <contents lang="en">
    As elements <object-ref objid="obj3" />
    and <object-ref objid="obj4" /> are
    unordered, they must be swapped.
  </contents>
</narrative>
```

Listing 17: Declaring Basic Metadata

```
<metadata xmlns="http://www.algoanim.net/
                  xmlspec/metadata">
 <author>
   <name>Bill</name>
   <email>bill@iticse2005.org</email>
 </author>
 <title>Insertion in AVL trees</title>
 <type>Animation</type>
 <skill type="required">
   <concept>basic-avl-trees</concept>
   <level binary="Known"
          qualitative="intermediate"/>
 </skill>
</metadata>
```

Listing 18: Refering to External Metadata

```
<metadata  xmlns="http://www.algoanim.net/
           xmlspec/metadata#">
   <external url="http://www.algoanim.net/
           xmlspec/examples/metadata.xml"/>
   <language>en-US</language>
</metadata>
```

Listing 19: Relative coordinates example

```
<polyline>
 <coordinate x="20" y="10" />
 <coordinate>
  <offset x="10" y="30"
   base-object="pg1" node="3" />
  <!-- refers to polygon "pg1", as shown
        in Figure 3 on page 132-->
 </coordinate>
 <coordinate x="10" y="20" />
 <closed value="true" />
</polyline>
```

Listing 20: Internationalization in text primitives

```
<text>
 <coordinate x="30" y="50" />
 <alignment value="left" />
 <boxed value="true" />
 <contents lang="en">
  Text of the object
 </contents>
 <contents lang="fi">
  Objektin teksti
 </contents>
 <contents lang="de">
  Der Text des Objekts
 </contents>
</text>
```

Listing 21: Style example

```
<style id="style1">
 <arrow forward="true" backward="false"/>
 <color name="red"/>
 <fill-color name="green"/>
 <stroke type="solid" width="3"/>
 <font family="Monospaced" size="8"/>
</style>
```

Listing 22: Overriding style settings

```
<style id="style2" uses="style1">
 <color red="0" green="240" blue="120"/>
</style>
```

Listing 23: Coordinates example

```
<!-- point 1 -->
<coordinate>
 <offset x="10" y="10"
         base-object="pg1" anchor="SW" />
</coordinate>
<!-- point 2 -->
<coordinate>
 <offset x="-35" y="50"
         base-object="pg1" node="2" />
</coordinate>
<!-- point 3 -->
<coordinate>
 <offset x="10" y="-10" />
</coordinate>
<!-- point 4 -->
<coordinate>
 <offset x="-30" y="30"
         baseline-of="text1" mode="end" />
</coordinate>
```

Listing 24: Animation step example

```
<step>
 <narrative ref="external_narration.html"/>
 <show> ... </show>
 <move> ... </move>
  ⋮
 <move> ... </move>
 <rotate> ... </rotate>
</step>
```

Listing 25: Transformations example

```
<rotate degree="90" type="simple">
 <object-ref id="obj1" />
 <timing><delay s="2" /></timing>
 <!-- rotation center, can be any node -->
 <coordinate x="10" y="20" />
</rotate>

<move type="move">
 <object-ref id="objGrp1" />
 <timing>
  <duration frames="15" />
  <delay ms="200" />
 </timing>
 <along-object id="pg1" />
</move>
```

Listing 26: Defining a new *shape*

```
<!-- First define the shape -->
<define-shape name="house">
 <square>
  <coordinate x="10" y="20" />
  <length>10</length>
 </square>
 <triangle>
  <coordinate x="10" y="20" />
  <coordinate x="15" y="15" />
  <coordinate x="20" y="20" />
 </triangle>
</define-shape>

 ...

<!-- Later use the house shape -->
<shape uses="house">
 <coordinate x="30" y="50" />
 <shape-scale value="1.5" />
</shape>
```