Toward Empirically-Based Software Visualization Languages

Sarah Douglas[†], Christopher Hundhausen[†], and Donna McKeown*
[†]Computer & Information Science Dept.
*Psychology Dept.
University of Oregon
Eugene, OR 97403
{douglas, chundhau}@cs.uoregon.edu, donna@dynamic.uoregon.edu

Abstract

Underlying any single-user software visualization (SV) system is a visualization language onto which its users must map the computations they would like to visualize with the system. We hypothesize that the usability of such systems turns on their ability to provide an underlying visualization language that accords with the ways in which their users conceptualize the computations to be visualized. To explore the question of how to design visualization languages grounded in human conceptualization, we present an empirical study that made use of a research method called visualization storyboarding to investigate the human conceptualization of the bubblesort algorithm. Using an analytical framework based on entities, attributes, and transformations, we derive a semantic-level visualization language for bubblesort, in terms of which all visualizations observed in our study can be expressed. Our empirically-based visualization language provides a means for predicting the usability of the visualization language defined by Lens [8], a prototypical single-user SV system. We draw from a follow-up usability study of Lens to substantiate our predictions.

1 Introduction

Predicated on the intuitive idea that a mapping between an executing program and computer graphics can give one insight into the program's dynamic behavior, computer-based software visualization (SV) systems provide techniques for facilitating four central activities involving that mapping: its (1) design, (2) specification, (3) observation, and (4) interactive exploration. Early SV systems such as Balsa [1] and Tango [12] defined system user models in which different actors performed those activities. In the Balsa model, for example, *client programmers*—algorithmaticians and animators—were responsible for designing and specifying the algorithm-to-graphics mapping, whereas the *end users*—script authors and script viewers—actually observed the mappings. Although, in theory, the same person could have assumed both roles, in practice they were played by different people, and at different times.

That division of labor can be largely attributed to the sharp difference in the levels of expertise that the Balsa system demanded of client programmers and end users. Because they needed to possess a detailed knowledge of a low-level graphics system, client programmers required a much higher level of expertise than end-users, who required no knowledge whatsoever of either the program or the techniques used to animate it, in order to work with the Balsa system. Such a conceptual model, which encourages SV's central activities to be conducted by different people, and at different points in time, has been the dominant SV system paradigm for nearly a decade.

Motivated by a desire to bridge the gap between client programmer and end user—that is, to empower those without low-level graphics programming skills to rapidly prototype their own visualizations—SV researchers have begun to explore ways of realizing an alternative SV model, in which SV's central activities can be undertaken by the same person. Recently, Mukherjea and Stasko [8] developed the first system to adopt such an alternative conceptual model. Their *Lens* system defines an interactive environment into which an algorithm written in C may be loaded. Using a combination of dialog box fill-in and direct manipulation, Lens users can interactively specify their own mapping between the program (which runs as a separate process under the control of the dbx debugger) and graphics. At any point in the process, users can choose to run the program and observe the visualization they have defined.

Notice that, from the perspective of the system user, at least three activities are involved in the process of visualization programming under the single-user SV paradigm:

(1) the conceptualization of the computation and its representation, both in terms of pseudocode or a specific programming language, and in terms of a mental visualization;
(2) the mapping of that conceptualization to the *visualization language* defined by the system; and
(3) the manipulation of the SV system's user interface to program the visualization.

Thus, any single-user SV system must define a visualization language onto which computations to be visualized with the system must be mapped.

The central hypothesis of our work is that the usability of computer-based SV systems—and, in particular, those that assume the single user model—turns on their ability to offer a visualization language that accords with the ways in which humans conceptualize the computations to be visualized with the systems. The goal of our research is two-fold: (1) to explore methods for designing SV languages grounded in empirical studies; and (2) to substantiate and refine our hypothesis by integrating empirically-based SV languages into single-user SV systems and conducting further empirical studies.

In this paper, we present our foray into (1). We begin, in Section 2, by describing an empirical study that makes use of a research technique called *visualization storyboarding* to explore human conceptualization of the bubblesort algorithm. In Section 3, we present an analytical framework that allows us to formulate a *semantic-level* visualization language for bubblesort based on our study. In Section 4, we demonstrate the utility of the approach by showing how the language can be used to diagnose language-based usability problems with SV systems, using a follow-up study we conducted with the Lens software for supporting evidence. Section 5 surveys related work. We conclude, in Section 6, by identifying two directions for future research.

2 An empirical study of the human conceptualization of bubblesort

In this section, we introduce a research method called visualization storyboarding, and we describe an exploratory study that made use of the method to investigate the ways in which humans visualized the bubblesort algorithm. See [4] for a more thorough treatment of the study.

2.1 Visualization storyboarding

Based on the storyboarding technique introduced by the animation industry in the 1930s, *visualization storyboarding* takes advantage of the familiarity and dynamism of art supplies and human conversation to create a situation in which humans can describe computer programs naturally. Two person teams are provided with a full spectrum of colored construction paper, scissors, and different colored pens; they are then asked to work together to describe a computer program to someone who has never seen it. The resulting *storyboard* of the algorithm's execution—a dynamic presentation consisting of animated construction paper cut-outs, drawings, conversation, and gestures—is videotaped, becoming the empirical data to be analyzed at a later time.

Visualization storyboarding applies two qualitative research techniques—*constructive interaction* and *conversation analysis*—to the situation of software visualization design. By using two participants, constructive interaction creates a situation of collaborative problem solving in which each participant must inform the other, in an explicit verbal and visual record, about problems, causes, and solutions each has encountered. To interpret participants' interaction, we review the videotapes of participant sessions using conversation analysis techniques developed by Douglas [3].

2.2 Procedure

We videotaped three same-sex pairs of participants (two consisting of women, one consisting of men) during 45 to 70 minute sessions. All participants were graduate students in computer science at the University of Oregon, and all reported that they understood the bubblesort algorithm. Nonetheless, we provided participants with C code for their reference.

After providing our pairs of participants with art supplies, we gave them the following task: "If you were to explain the bubble sort algorithm to someone who had never seen it, how would you do it? Feel free to use any of these resources (construction paper, scissors, etc.) to assist you in your explanation." If they decided to design a visualization for a sample data set, we recommended that they only step through a few of the iterations for simplicity's sake.

2.3 Observations

All pairs of participants used the art supplies materials to create homemade animations of bubblesort operating on a sample data set; they augmented their construction paper animations extensively with their own gestures and comments. Figures 1, 2, and 3 depict our participants' storyboards. Note that, because they are static in nature, these figures cannot do full justice to our participants' dynamic presentations, which must be seen on videotape to be appreciated.

One pair, whom we shall call the Number Pair, used numbers (e.g. 5, 4, 3) to indicate the magnitude of each array element, and used color to illustrate significant state changes (see Figure 1). They initially shaded all elements in the array red to indicate that the elements were unsorted. Each pass of the algorithm's outer loop

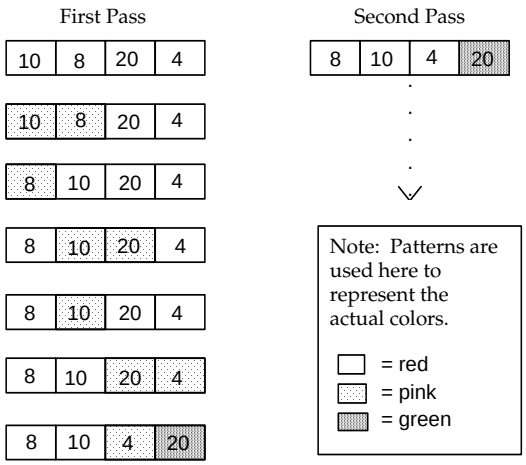

Figure 1. The visualization trace created by the Number Pair to demonstrate bubble sort

was depicted as a column of rows. Successive horizontal rows of array elements within a column corresponded to successive passes of the algorithm's inner loop. Within each horizontal row, the two array elements compared during that pass were shaded pink to indicate that they had been compared. At the end of a pass of the algorithm's outer loop, the pair colored the element that had reached its rightful place in the array green, and began a new column for the next pass of the algorithm's outer loop.

Another pair, whom we shall call the Color Pair, used color to indicate magnitude. The legend they constructed (see the triangles at the top of Figure 2) defined a canonical (spectral) order for a five-element array. By illustrating the array after each successive pass of the algorithm's outer loop, they demonstrated how the bubblesort gradually placed an unsorted array into that order. Like

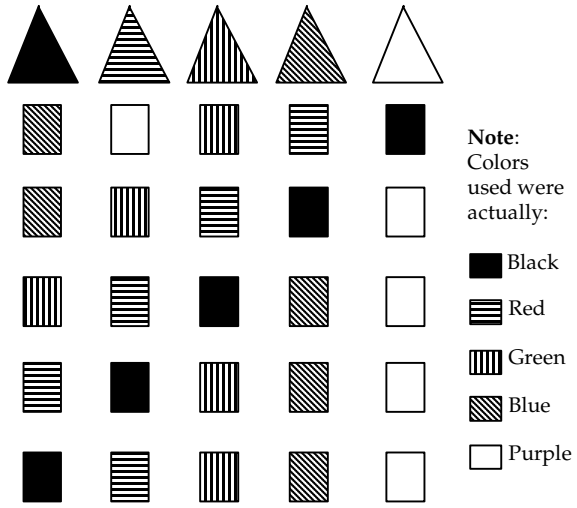

Figure 2. The visualization trace created by the Color Pair to demonstrate bubble sort

the first pair, this pair had built, by the end of the sort, a history array of the sort; however, they depicted each pass of the algorithm's outer loop with row instead of a column.

The third pair, whom we shall call the Football Pair, also used color to denote magnitude (see Figure 3). However, color played only a secondary role in their visualization, which drew extensively from concepts of American football to depict various aspects of the algorithm.

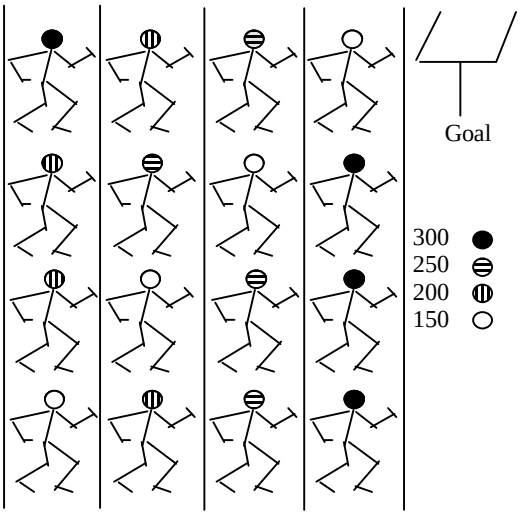

Figure 3. The visualization trace created by the Football Pair to demonstrate bubblesort

Sorting elements were represented by football players whose varying weights were represented by color, as indicated by the legend. At the beginning of each pass of the sort, the football was given to the first player in line. The ball carrier and the player next in line symbolized the two elements being compared at any given time. If the ball carrier weighed more than the next in line, then the ball carrier knocked that player over, thereby exchanging places with that player in the array. If, however, the ball carrier weighed less than the next player in line, the ball carrier fumbled the ball to the next player, who then became the ball carrier. This process of ball advancement continued until the ball carrier reached the end of the line, whereupon a new pass of the sort began.

3 Formulation of a semantic-level SV language for bubblesort

In this section, we motivate an analytical framework that provides a means for characterizing visualization semantics. We then use the framework to derive a semantic-level SV language, in terms of which all of the bubblesort visualizations observed in our study can be expressed.

3.1 Analytical framework

In deciphering the languages employed by our participants in their visualization storyboards, we find it useful to distinguish between two linguistic levels: *lexical* and *semantic*. We use the *lexical level* to refer to the graphical vocabularies of their visualizations. In contrast, we use the *semantic level* to refer to the meanings of elements of those graphical vocabularies.

At a lexical level, our participants' visualization languages varied on three main fronts. First, they used varying *graphical entities*. For example, the Number Pair used numbers, the Color Pair used triangles and squares, and the Football Pair used football players. Second, the visualizations conveyed various kinds of information by using different *attributes* of those entities. For example, the Color Pair used *color* to represent sorting element magnitude; the Football Pair used the *position* of a football to denote the two elements currently being compared. Finally, the visualizations employed different *transformations* among entities and attributes to convey various kinds of information. For instance, the Number Pair altered number square color to indicate comparisons, whereas the Football Pair changed the location of the football to indicate comparisons.

Despite the lexical differences among our participants' visualization languages, notice that they all encoded the bubblesort at a similar level of abstraction—one that conveys bubblesort's overall functionality in terms of a similar kernel of fundamental abstractions, including *sorting element*, *array of sorting elements*, *comparison* and *exchange*.

Given the three dimensions along which our participants' visualization languages differed lexically, and given that all of their visualizations portrayed bubblesort's abstract functionality using a similar semantics, we propose a framework for analyzing their visualizations that characterizes them in terms of the (lexical level) entities, attributes, and transformations by which they represented bubblesort's (semantic level) abstract functionality. As we shall see below, by using such a framework, we shall see below, by using such a framework, we uncover a similar semantics for the visualization languages, allowing us to unify the languages at a semantic level.

3.2 Deriving a semantic-level SV language for bubblesort

Table 1 summarizes the mapping between the lexical entities and attributes created by each pair, and their underlying semantics. For example, the Number Pair used numbers in a row of contiguous squares to designate an array. Notice that some semantic-level entities present in a typical semantic-level pseudocode description of bubblesort, such as temporary storage and subscripts, were not represented in our participants' visualizations. Participants perhaps believed that such entities would serve to confound the abstract functionality that they were striving to capture. Conversely, the color legend, used by both the Color Pair and the Football Pair, cannot be derived from the algorithm's semantics. Instead, it is an artifact of the visualization itself, most likely arising out of a perceived need to explicate the ordering relationship among graphical entities.

Table 2 summarizes the mapping between transformations created by each pair, and their underlying semantics. For instance, the Number Pair used color to indicate that two elements were being compared (the elements turned pink), and to indicate a difference between sorted elements (which turned green) and unsorted elements (red).

Derived directly from the analysis presented in Tables 1 and 2, the SV language presented in Table 3 unifies our participants' SV languages at a *semantic* level. While we might have chosen to express that language using any number of different formalisms, our analytical framework lends itself naturally to expression in terms of abstract data types (ADTs), which capture the semantics underlying our participants' visualization languages, but which intentionally hide the lexical details of their graphical representation. Indeed, as we have seen, such lexical

Semantics	Number Pair Lexicon	Color Pair Lexicon	Football Pair Lexicon
Sorting element	Square	Square	Stick Figure
Magnitude of element	Number symbol	Color	Color (as weight)
Array of elements	Contiguous row of squares	Non-contiguous row of squares	Contiguous row of figures
Inner loop pass history	Rows of sorting elements	—	—
Outer loop pass history	Columns of rows	Rows of sorting elements	Rows of sorting elements
Legend explicating ordering on sort elements	—	Triangles with color spectrum	Column of color/player weight pairs

Table 1. Mappings between lexical entities and attributes of the human visualizations and their semantics

Semantics	Number Pair Lexicon	Color Pair Lexicon	Football Pair Lexicon
Initialize magnitudes and counter	—	—	—
DO outer loop	Start new column of rows	Create new row of squares	Create new row of football players
DO inner loop	Create new row of squares	—	—
a) Reference elements to be compared	Color elements pink	—	Location of football
b) *Compare* elements (same, <, >)	—	—	Intuitions about how player size relates to running, tackling, and fumbling
c) *Exchange* elements	Exchange numbers	Exchange colors	Ball carrier advances by tackling next football player in line (thereby exchanging positions with that player)
d) Don't *exchange* elements	—	—	Fumble football to next player in line
Terminate outer loop	Color square in correct order green	—	—
Terminate Sorting	Ordering of natural numbers, all squares green	Color squares match legend	Players ordered by weight

Table 2. Mappings between lexical transformations of the human visualizations and their semantics

ADT Name	Operations	Semantics
Sorting_element	*instantiate*(*magnitude*)	Create graphical appearance for element based on magnitude
	compare_highlight()	Graphically indicate a comparison involving this element
	inorder_highlight()	Graphically indicate that this element is now in its rightful place in the sort array
	move(*new_index*)	Move this element to location defined by *new_index*
Sort_array	*instantiate* (*array of integers*)	Create graphical appearance for sort array (facilitates data initialization)
	start_outer_loop()	Graphically indicate the beginning of a pass of the algorithm's outer loop
	start_inner_loop()	Graphically indicate the beginning of a pass of the algorithm's inner loop
	compare(*elt1, elt2*)	Graphically indicate a comparison between *elt1* and *elt2* by calling on *compare_highlight*() operations of *elt1* and *elt2*
	exchange(*elt1, elt2*)	Graphically indicate an exchange involving *elt1* and *elt2* by calling on the *move*(*new_index*) operations of *elt1* and *elt2*
	no_exchange(*elt1, elt2*)	Graphically indicate that there was no exchange involving *elt1* and *elt2*
	terminate_outer_loop()	Graphically indicate the end of a pass of the outer loop, possibly calling upon *inorder_highlight*
Legend	*instantiate*(*array of integers*)	Graphically explicate ordering on sort elements

Table 3. Semantic-level SV language for bubblesort in terms of ADTs

details vary considerably depending on the people doing the visualization.

In Figure 4, we use our ADT language in conjunction with a procedural Pascal-like language to express all of our participants' visualizations as a common program. Having provided, within our study, the lexical details of how each operation manifests itself graphically, our participants can thus be viewed as the *implementors* of the ADTs used in that program.

4 Using semantic-level analysis to evaluate existing single-user SV software

In this section, we demonstrate how the semantic-level analysis introduced in the previous section can assist in the evaluation of computer-based SV systems. Section 4.1

```
VAR
  a  : ARRAY OF INTEGER;
  le  : legend;
  sa : sort_array;
i, j, temp: INTEGER;

BEGIN (* Bubblesort *)
  read data into a;
  sa.instantiate(a);
  le.instantiate(a);
  FOR j := n – 2 TO 0 DO BEGIN
    sa.start_outer_loop();
    FOR i := 0 TO J DO BEGIN
      sa.start_inner_loop();
      sa.compare(i, i+1)
       IF a[i] > a[i+1] THEN BEGIN
        sa.exchange(i, i+1);
         temp := a[i];
         a[i] := a[i+1];
         a[i+1] := temp
       END (* IF *)
       ELSE
         sa.no_exchange(i, i+1);
    END (*inner FOR *)
  END (* outer FOR *)
END; (* Bubblesort *)
```

Figure 4. Expressing participants' visualizations by annotating a procedural language with calls to the ADTs defined in Table 3

uses our framework to analyze the bubblesort SV language provided by Lens, a single-user SV system developed by Mukherjea and Stasko [8]. In Section 4.2, we substantiate our analysis by drawing from an actual usability study we conducted on Lens.

4.1 Analyzing the Lens SV language for bubblesort

Figure 5 presents the final frame of the bubblesort visualization that one would typically define in Lens. Array elements are depicted as sticks whose height corresponds to magnitude. Two elements are flashed to indicate a comparison, and smoothly change places to indicate an exchange. An ascending row of sticks indicates that the array has been sorted.

In Tables 4 and 5, we illustrate the manner in which the semantic-level entities, attributes, and transformations identified in our empirical study map to the Lens lexicon. As those tables indicate, the visualization language defined by Lens accords quite well, at a semantic level, with the visualization languages defined by our participants; indeed, it provides support for depicting the

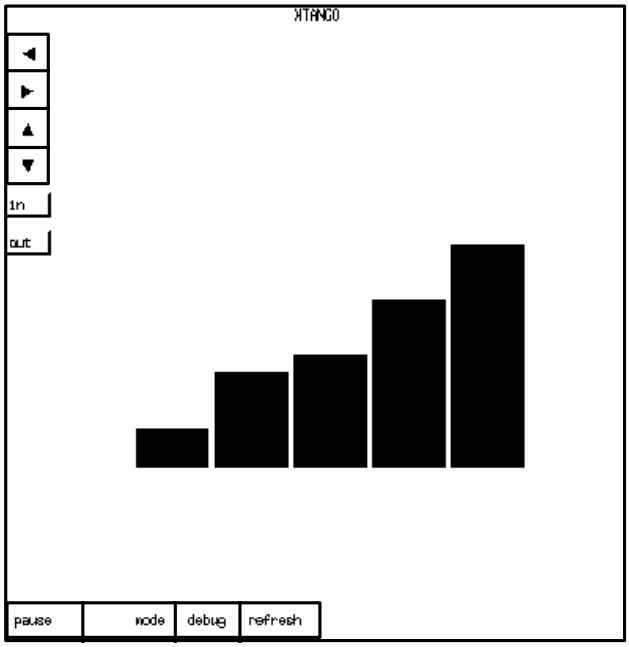

Figure 5. The final frame of the standard bubblesort visualization in Lens

Semantics	Lens Lexicon
Sorting element	Stick
Magnitude of element	Stick height or width
Array of elements	Non-contiguous row of sticks
Inner loop pass history	—
Outer loop pass history	—

Table 4. Mappings between semantic-level entities and attributes identified in our empirical study and the Lens lexicon

Semantics	Lens Lexicon
Initialize magnitudes (Done once per loop in participant visualizations)	Create initial row of black sticks
DO outer loop	—
DO inner loop	—
a) Reference elements to be compared	Flash elements, or change fill color
b) *Compare* elements (same, <, >)	—
c) *Exchange* elements	Exchange sticks
d) Don't *exchange* elements	—
Terminate outer loop	—
Terminate Sorting	Sticks ordered by increasing height or width

Table 5. Mappings from semantic-level pseudocode transformations to Lens visualization

bubblesort in terms of many of the same semantic-level abstractions as exist in our semantic-level SV language.

Notice, however, that the Lens semantics does not fully cover the semantics of the visualizations we encountered in our study. In particular, we can identify two shortcomings. First, the Lens language supports no means for maintaining a history of bubblesort's inner and outer loops; instead, all passes of the sort must be performed on the same set of elements. Here, it is important to point out that the Lens interface does provide the option of starting and stopping the visualization at any point; thus, intermediate snapshots of the array can be momentarily viewed, although they cannot be preserved for later inspection. Second, the Lens language provides no means for defining a legend for explicating the ordering on sorting elements.

Based our semantic-level analysis of the Lens SV language, we can make two hypotheses regarding its usability:

Hypothesis 1: Lens users will not have problems with the semantics behind the lexical elements in terms of which the abstract functionality of bubblesort can be depicted in Lens. Indeed, all of the semantic entities available in the Lens language are also in our semantic-level SV language.

Hypothesis 2: Lens users will encounter two kinds of usability problems with the Lens language stemming from its failure to support two kinds of semantics present in our semantic-level SV language: *array history* and *legends*. Since the Lens language does not support the former, we might expect our participants to complain about an inability to distinguish among intermediate passes of the algorithm's loops. Since the Lens language does not support the latter, we might expect participants to have problems interpreting the target ordering of the array.

In the following section, we scrutinize these hypotheses by considering a usability study we conducted on Lens.

4.2 Using a follow-up usability study on Lens to substantiate the analysis

Overview of study. Two pairs of students who participated in the study presented in Section 2 went on to program a visualization for bubblesort using Lens. Participants were given a three page description of procedural instructions for implementing a predefined visualization of bubblesort corresponding to the one described in Section 4.1; we then asked them to use the Lens software to program the visualization. As was the case with our visualization storyboarding experiments, both sessions were videotaped and later analyzed using conversational analysis techniques developed by Douglas [3].

Observations relevant to Hypothesis 1. As hypothesized, we observed that the overall usability of the Lens language was quite good. By the time they had watched the entire Lens visualization, both pairs of participants clearly indicated that they grasped the mappings between elements of the Lens lexicon and their underlying semantics. Since there existed a good match between the underlying semantics of the Lens visualization, and the semantics of the visualizations defined by our two pairs of participants, the fact that the Lens visualization used different lexical elements did not hinder our participants' comprehension.

Observations relevant to Hypothesis 2. Participants did not explicitly complain about the fact that Lens visualization did not represent sort history. Thus, we have no reason to believe that the failure of the Lens language to provide a direct means for representing sort history was the cause of a usability problem.

We do, however, have convincing evidence that the lack of a legend in the Lens visualization caused a usabilty problem. Indeed, both pairs of study participants were initially confused about the mapping between element magnitude and stick height in the Lens visualization. As the visualization progressed, they eventually came to understand that mapping. Nonetheless, our semantic-level analysis suggests that, if the Lens language had supported the definition of an order legend, participants might have grasped the visualization more quickly.

5 Related work

The research described in this paper purports to use empirical studies as a basis for SV language design and evaluation. Two lines of related work have employed empirical studies with humans within the context of SV, while a third line of related work has specifically addressed the problem of SV language design.

A number of researchers have used empirical studies to evaluate the efficacy of software visualization as a pedagogical tool [7,10]. Our work differs fundamentally from this work both in the research method (quantitative factors analysis versus qualitative visualization storyboarding), and in the research goal (evaluation of SV efficacy versus SV language design).

Ford [5] employs a research method similar to ours to document human visualizations of C++ code fragments; the visualizations are classified using Cox and Roman's [2] abstraction classification system. In contrast to our study, Ford's study considers the visualization of general imperative and object-oriented programming concepts—for example, variables, pointers, loops, conditionals, arrays, and classes; visualization conceptualization is not considered at the level of algorithm semantics. Fur-

ther, Ford does not use the study's results as a basis for an SV language. Instead, Ford's interest has been in devising a set of empirically-based visual program abstractions for aiding programmer comprehension of conventional text-based program views [6].

Drawing from an analysis of human pen-and-paper descriptions, Radiya and Radiya [9] introduce a model for graphically describing algorithms. However, whereas they are interested in using such descriptions as basis for a visual programming language, we are interested in applying our research to SV languages.

To design the Lens SV language, Mukherjea and Stasko [8] studied visualizations of 42 algorithms drawn from several problem domains, including sorting, searching, graph theory, and graphics. Programmed by over 25 people from 4 different institutions, all of the visualizations they studied had been implemented in XTango [11], an animation toolkit based on Stasko's path-transition paradigm [12]. Mukherjea's and Stasko's approach differs from ours in that it uses visualizations already programmed in a particular SV language as a basis for design; we are interested in grounding SV languages in visualizations described independently of computer-based technology and languages.

6 Conclusions and future work

In this paper, we have shown how visualization storyboarding studies, together with a semantic-level analytical framework, can be used to derive an empirically-based, semantic-level SV language for bubblesort. As we have demonstrated, such a language can prove useful in diagnosing language-based usability problems in existing single-user SV software.

However, we suspect that our approach holds even greater promise as a tool for computer-based SV system design. We envision that semantic-level languages like the one presented here could serve as a foundation for highly-usable single-user SV systems. Future research should both identify the semantic primitives of a wider range of computations, and confirm the approach's potential as a design tool by actually building single-user SV software on top of SV languages grounded in the approach.

Acknowledgment

We gratefully acknowledge John Stasko and his colleagues for allowing us to use a prototype version of Lens for our usability study.

References

[1] Brown, M. *Algorithm Animation*. Cambridge, MA: The MIT Press, 1987.

[2] K.C. Cox and G-C. Roman. Abstraction in algorithm animation. *In Proc. 1992 IEEE Workshop on Visual Languages* (Seattle, WA), pp. 18–23, 1992.

[3] S.A. Douglas. Conversational analysis and human-computer interaction design. P.Thomas (Ed.) *The Social and Interactional Dimensions of Human-Computer Interfaces* Cambridge University Press, (in press).

[4] S. Douglas, D. McKeown, and C. Hundhausen. Exploring human visualization of algorithms. TR CIS-TR-94-27, Deptartment of Computer and Information Science, University of Oregon, Eugene, 1993.

[5] L. Ford. How programmers visualize programs. Research report 271, Department of Computer Science, University of Exeter, Exeter, U.K., 1993.

[6] L. Ford and D. Tallis. Interacting visual abstractions of programs. In *Proc. 1993 IEEE Workshop on Visual Languages* (Bergen, Norway), pp. 93–97, 1993.

[7] A.W. Lawrence, A.N. Badre, and J.T. Stasko. Empirically evaluating the use of animations to teach algorithms. In *Proc. 1994 IEEE Symposium on Visual Languages* (St. Louis, MO), pp. 48–54, 1994.

[8] S. Mukherjea and J.T. Stasko. Toward visual debugging: Integrating algorithm animation capabilities within a source-level debugger. *ACM Trans. on Computer-Human Interaction 1(3)*, pp. 215–244, 1994.

[9] A. Radiya and V. Radiya. A model of human approach to describing algorithms using diagrams. In *Proc. 1992 IEEE Workshop on Visual Languages* (Seattle, WA), pp. 261–263, 1992.

[10] J.T. Stasko, J., Badre, A., and C. Lewis. Do algorithm animations assist learning? An empirical study and analysis. In *Proc. INTERCHI '93 Conference on Human Factors in Computing Systems* (Amsterdam, The Netherlands), pp. 61–66, 1993.

[11] J.T. Stasko. Animating algorithms with XTANGO. *SICACT News 23(2)*, pp. 67–71, 1992.

[12] J.T. Stasko. Tango: A framework and system for algorithm animation. *Computer 23(9)*, pp. 27-39, 1990.