Specifying Algorithm Visualizations: Interesting Events or State Mapping?*

Camil Demetrescu[1], Irene Finocchi[2], and John T. Stasko[3]

[1] Dipartimento di Informatica e Sistemistica,
Università di Roma "La Sapienza",
Via Salaria 113, 00198 Roma, Italy.
demetres@dis.uniroma1.it,
http://www.dis.uniroma1.it/~demetres/

[2] Dipartimento di Scienze dell'Informazione,
Università di Roma "La Sapienza",
Via Salaria 113, 00198 Roma, Italy.
finocchi@dsi.uniroma1.it,
http://www.dsi.uniroma1.it/~finocchi/

[3] College of Computing / GVU Center,
Georgia Institute of Technology
Atlanta, GA 30332-0280.
stasko@cc.gatech.edu,
http://www.cc.gatech.edu/~john.stasko/

Abstract.

Perhaps the most popular approach to animating algorithms consists of identifying *interesting events* in the implementation code, corresponding to relevant actions in the underlying algorithm, and turning them into graphical events by inserting calls to suitable visualization routines. Another natural approach conceives algorithm animation as a graphical interpretation of the state of the computation of a program, letting graphical objects in a visualization depend on a program's variables. In this paper we provide the first direct comparison of these two approaches, identifying scenarios where one might be preferable to the other. The discussion is based on examples realized with the systems Polka and Leonardo.

1 Introduction

One of the main issues in algorithm animation is the specification of the graphical abstractions that illustrate computations. Two problems arise in this context:

* Work supported in part by the project "Algorithms for Large Data Sets: Science and Engineering" of the Italian Ministry of University and of Scientific and Technological Research (ex MURST 40%) and by the IST Programme of the EU under contract n. IST-1999-14.186 (ALCOM-FT).

modeling graphical scenes and animation transitions, and binding the attributes and the animated behavior of graphical objects to the underlying algorithmic code. The power of a specification method is mainly related to its flexibility, generality, and capability to customize visualizations. In this setting, a common approach is to use conventional textual programming languages as specification tools. In general, a visualization specification language can be different from the language used for implementing the algorithm to be visualized, though they often coincide. An important factor that determines the connection of the visualization code with the algorithm implementation is how animation events are triggered by the underlying computation. One approach, dubbed *event-driven*, consists of identifying *interesting events* in the implementation code, corresponding to relevant actions of the algorithm, and turning them into graphical events by inserting calls to suitable animation routines, usually written in an imperative or object-oriented style. Another natural approach, dubbed *data-driven*, is to specify a *mapping* of the computation state into graphical scenes, usually declaring attributes of graphical objects to depend on variables of the underlying program. In this case, animation events are triggered by variable modifications. For a comprehensive discussion of other specification methods used in algorithm visualization, we refer the interested reader to [4,12,13,15,21].

This article provides the first direct comparison of the interesting event and state mapping specification styles. The two approaches are reviewed in more detail in Section 2. Section 3 addresses the problem of specifying a basic algorithm visualization and provides two different solutions for the Bubblesort algorithm, one event-driven and one data-driven, realized in the systems Polka [20] and Leonardo [9]. Further advanced aspects of algorithm visualization specification are considered in Section 4: the discussion is based on refinements and extensions of the Bubblesort visualization code given in Section 3. Section 5 addresses concluding remarks.

2 Two Visualization Specification Techniques

In this section we briefly review the event-driven and the data-driven visualization specification methods, listing some systems that instantiate the two approaches, and in particular the systems Polka [20] and Leonardo [9].

2.1 Event-Driven Approach

A natural approach to animating algorithms consists of annotating the algorithmic code with calls to visualization routines. The first step consists of identifying the relevant actions performed by the algorithm that are interesting for visualization purposes. Such relevant actions are usually referred to as *interesting events*. For instance, in a sorting algorithm the swap of two items can be considered an interesting event. The second step is to associate each interesting event with a suitable animation scene. In the sorting example, if we depict the values to be ordered as a sequence of sticks of different heights, the animation of a swap event might be realized by exchanging the positions of the two sticks corresponding to the values being swapped. Animation scenes can be specified by setting up suitable visualization procedures that drive the graphic system according to the actual parameters generated by the particular event. Alternatively, these visualization procedures may simply log the events in a file for a *post-mortem* visualization. Calls to the visualization routines are usually obtained by annotating the original algorithmic code in the points where the interesting events take place. This can be done either by hand or by means of specialized editors.

The event-driven approach is very intuitive and virtually any conceivable visualization can be generated in this way. Besides being simple to implement, interesting events are not necessarily low-level operations (such as comparisons or memory assignments), but can be more abstract and complex operations designed by the programmer and strictly related to the algorithm being visualized (e.g., the swap in the previous example, as well as a rotate operation in the management of AVL trees). Major drawbacks are invasiveness (even if the code is not transformed, it is augmented) and code ignorance allowance: the person who is in charge of realizing the animation has to know the source code quite well in order to identify all the interesting points.

A limited list of well-known systems based on interesting events include Balsa [3], Zeus [5], Tango [18], XTango [19], Polka [20], CAT [6], ANIM [2].

Polka. In this paper we will consider examples of visualizations based on interesting events realized with Polka. Polka is a system for visualizing programs written in C++. The system has two main foci: allowing designers to create animations with smooth, continuous movements and simplifying the overall process of developing algorithm animations. To build an algorithm animation with Polka, the developer annotates the program source with *Algorithm Operations*. These are Polka's version of Interesting Events. The developer also creates *Animation Scenes* that are procedures which perform an animation chunk and are written using the Polka graphics library. Finally, the developer specifies a mapping between algorithm operations and animation scenes. The Polka system distribution includes full source code and numerous animation examples. Versions of Polka for both the X Window System and Microsoft Windows exist. Further information can be found at the URL `http://www.cc.gatech.edu/gvu/softviz`.

2.2 Data-Driven Approach

Data-driven systems rely on the assumption that observing how variables of a program change provides clues to the actions performed by the underlying algorithm. The focus is on capturing and monitoring the data modifications rather than on processing the interesting events issued by the annotated algorithmic code. Specifically, data-driven systems realize a graphical mapping of the state of the computation (*state mapping*): an example is given by conventional debuggers, which provide a direct feedback of how variables change over time.

Specifying an animation in a data-driven system consists of providing a graphical interpretation of the *interesting data structures* of the algorithmic code.

It is up to the system to ensure that the graphical interpretation reflects at any time the state of the computation of the program being animated. In the case of conventional debuggers, the interpretation is fixed and cannot be changed by the user: typically, a direct representation of the content of variables is provided. The debugger just updates the display after each change, sometimes highlighting the latest variable that has been modified by the program to help the user maintain context. In a more general scenario, an adjacency matrix used in the code may be visualized as a graph with vertices and edges, an array of numbers as a sequence of sticks of different heights, and a heap vector as a balanced tree. As the focus is only on data structures, the same graphical interpretation, and thus the same visualization code, may be reused for any algorithm that uses a given data structure. For instance, any sorting algorithm that manages to reorganize a given array of numbers may be animated with exactly the same visualization code that displays the array as a sequence of sticks. Main advantages of the data-driven approach are a clean animation design and a high ignorance of the code: in most cases only the interpretation of “interesting variables” has to be known in order to produce a basic animation. On the other hand, focusing only on data modification may sometimes limit customization possibilities, making it difficult to realize animations that would be natural to express with interesting events. As we will see in Section 4, a *pure state mapping* approach, where there is no connection of the visualization code with the program's control flow, is intrinsically less powerful than interesting events.

Examples of systems based on state mapping are Pavane [14,16], Leonardo [9], and WAVE [10]. Toolkits such as CATAI [8], Gato [17] and LEDA [11] provide self-animating data structures, incorporating the principles of state mapping, but still supporting interesting events. Declarative visual programming languages that integrate algorithm animation capabilities have been also considered (see, for instance, Forms/3 [7]).

Leonardo. In this paper we will consider examples of state mapping visualizations realized with Leonardo. Leonardo is an integrated environment for developing, executing, and visualizing C programs. It provides two major improvements over a traditional integrated development environment. In particular, it supports a mechanism for visualizing computations graphically as they happen by attaching in a declarative style graphical representations to key variables in a program. With this technique, basic animations can usually be obtained with a few lines of additional code. It is to notice that Leonardo does not realize a pure state mapping, in the sense that it allows the user to control in an imperative style which visualization declarations are active at any time. However, differently from the interesting events, these manipulations change the set of active declarations, rather than the visualization itself, and may not have necessarily an immediate effect on the graphical scene. As a second main feature, Leonardo includes the first run-time environment that supports fully reversible execution of C programs. The system is distributed with a collection of animations of more than 60 algorithms and data structures including approximation, combinatorial optimization, computational geometry, on-line, and dynamic algorithms. Leonardo has been widely distributed on CD-ROM in computer magazines and is available for download in many software archives over the Web. It has received several technical reviews and more than 18,000 downloads during the last two years. At the time of writing, Leonardo is available only on the Macintosh platform. Further information can be found at the URL `http://www.dis.uniroma1.it/~demetres/Leonardo/`.

3 Anatomy of a Basic Visualization Specification

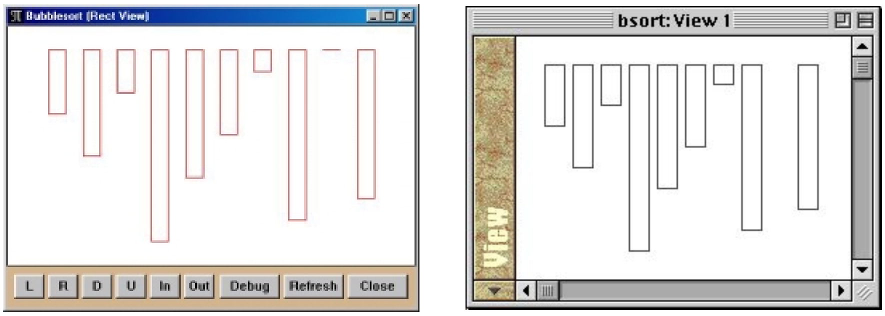
In this section we show how to specify a simple algorithm visualization using interesting events and state mapping. In particular, we focus on sorting algorithms and we show how to specify the well-known *sticks visualization*, where items to be sorted, assumed to be non-negative numbers, are visualized as rectangles of height proportional to their values. We first describe how the final visualization should look, and then we provide two solutions for the Bubblesort algorithm: one event-driven, realized with Polka, and one data-driven, realized with Leonardo. We give and discuss actual code and screenshots from both. For simplicity, we do not address issues of interaction with the visualization.

Bubblesort Code. We base our visualization examples on the following C/C++ implementation of the Bubblesort algorithm, which sorts an array `v` of `n` integer values.

```
1.   int v[]={3,5,2,9,6,4,1,8,0,7}, n=10, i, j;
2.   void main(void) {
3.       for (j=n; j>0; j--)
4.           for (i=1; i<j; i++)
5.               if (v[i-1]>v[i]) {
6.                   int temp=v[i]; v[i]=v[i-1]; v[i-1]=temp;
7.               }
8.   }
```

In this implementation, the first pass of lines 4–7 scans the first n elements, the second pass scans the first $n-1$ elements, etc. As elements are being swapped, each pass leaves the highest element found at its final proper position.

Visualization Setup. The first steps in specifying an algorithm visualization consist of deciding which pieces of information related to the algorithm's execution should be visualized and choosing a suitable graphical representation for them. In the case of sorting algorithms, an effective visual methaphor is to associate sticks of different heights to elements to be sorted. A possible simple layout places sticks vertically from left to right aligning their tops at the top of the viewport (see Figure 1). A swap operation can be animated in many ways: perhaps the simplest one is to show consecutive scenes that visualize the sticks before and after the swap.



(a) (b)

Fig. 1. Screenshots of the Bubblesort visualization in: (a) Polka; (b) Leonardo.

Polka. Visualizations are specified in Polka by annotating the program source with interesting events. Below, we show the source code for the Bubblesort program that has been annotated with interesting event calls.

```
int v[] = {3,5,2,9,6,4,1,8,0,7}, n=10, i, j;
void main(void) {
    bsort.SendAlgoEvt("Input",n,v);
    for (j=n; j>0; j--)
        for (i=1; i<j; i++)
            if (v[i] > v[i+1]) {
                int temp = v[i]; v[i] = v[i-1]; v[i-1] = temp;
                bsort.SendAlgoEvt("Exchange",i,i-1);
            }
}
```

Two events exist here. The first, "`Input`", signifies that all the array values to be sorted are set and that the animation should draw the initial configuration of the array. The event must send the size of the array and the array itself to the animation component as parameters.

We will omit the animation scene that is invoked as a response to the "`Input`" event for brevity. This scene creates and lays out the set of vertical rectangles and stores them in an array of Polka `Rectangle` objects, which is a subclass of the basic graphic primitive `AnimObject`. The scene does involve some subtle geometric calculations, however, as the designer must position all the rectangles with their tops aligned, space the rectangles out horizontally, and scale the heights of the rectangles according to the corresponding array values. Frequently, this type of geometric layout is the most difficult aspect of creating an algorithm animation.

The second event, "`Exchange`", signifies that a swap of two elements has occurred. It passes the indices of the two exchanged array elements as parameters. The corresponding animation code for this event is shown below.

```
int Rects::Exchange(int i, int j) {
 2.       Loc *loc1 = blocks[i]->Where(PART_NW);
 3.       Loc *loc2 = blocks[j]->Where(PART_NW);
 4.       Action a("MOVE",loc1,loc2,1);
 5.       Action *b = a.Reverse();
 6.       int len = blocks[i]->Program(time,&a);
 7.       time = Animate(time,len);
 8.       len  = blocks[j]->Program(time,b);
 9.       time = Animate(time,len);
10.       Rectangle *t = blocks[i]; blocks[i] = blocks[j]; blocks[j] = t;
11.       return len;
12.   }
```

First, we get the top-left (NW) corners of the two appropriate rectangles (lines 2–3), and then we create two movement Actions between them, in the two opposite directions (lines 4–5). Next, we schedule the first block's animation to occur at the current time; animate it; schedule the second block's animation; and animate it. Finally, we must swap the two objects being held int the Rectangle AnimObject array. This animation routine makes the first rectangle move in one sudden jump, then the second rectangle moves afterward, again in one jump. Note that the variables blocks and time are defined in this particular View of the animation which is a C++ class of type Rects here. A screenshot of the resulting visualization in Polka is shown in Figure 1a.

Leonardo. Visualizations are specified in Leonardo by adding to C programs declarations written in ALPHA, a simple declarative language, enclosing them with separators /** and **/. A complete ALPHA specification of the sticks visualization described above is shown below; this fragment can be simply appended to the Bubblesort code and compiled in Leonardo.

```
 1.   /**
 2.       View(Out 1);
 3.       Rectangle(Out ID,Out X,Out Y,Out L,Out H,1)
 4.           For N:InRange(N,0,n-1)
 5.           Assign X=20+20*N  Y=20  L=15  H=15*v[N]  ID=N;
 6.   **/
```

In line 2 we declare a window with identification number 1: this window is the container of the visualization. Sticks are then declared in lines 3–5, where we enumerate n rectangles (line 4), and we locate them in the local coordinate system according to the desired layout (line 5). Specified geometrical attributes include the coordinates of the left-top corner (X,Y), the width W, and the height H of the N-th rectangle, with N $\in [0, n-1]$. The last parameter in the Rectangle declaration (line 3) makes sticks appear in window 1. Like windows, rectangles have identification numbers (ID): this allows us to refer to them in any subsequent declarations.

Observe that the ALPHA code refers to variables v and n of the underlying C program, and size and position of sticks depend on them: Leonardo reevaluates automatically the ALPHA code and updates the graphical scene for each change of these variables. Since both statements `v[i]=v[i-1]` and `v[i-1]=temp` in line 6 change the array `v`, this yields two animation events per swap. A screenshot of the resulting visualization in Leonardo is shown in Figure 1b.

Even if imperative state mapping specification has also been considered (see WAVE [10]), the declarative approach has many advantages: in particular, the programmer is encouraged to think in terms of "what she wants", and not in terms of "how to obtain it" (see, e.g., Section 4.3). A price paid for this, however, may be a steeper learning curve for programmers who have never used a declarative language.

4 Customizing Visualizations

The task of specifying a visualization usually proceeds incrementally through different levels of sophistication. In Section 3 we have shown how to specify the well-known sticks visualization of the Bubblesort algorithm. As the power of a specification method is mainly related to flexibility, generality, and capability of customizing visualizations, we now consider some refinements of the basic Bubblesort animation, discussing further aspects of interesting events and state mapping.

4.1 Specifying the Granularity of Animations

We use the word *granularity* to indicate the level of detail of animation events: for instance, a sorting animation where items being swapped are moved one at a time, as in the example in Section 3, is characterized by a higher granularity (closer to the actual code that uses a temporary variable for the swap) than the one where both items are moved simultaneously (elementary steps are logically grouped and details elided).

There is a main difference in the way granularity is controlled with interesting events and state mapping. To generate an animation event with interesting events, a function has to be called: thus, increasing the number of animation events requires increasing the number of function calls, so the granularity is low by default. With state mapping, each change of a variable being mapped into some graphical object yields automatically an animation event, so the granularity is high by default: to control granularity we therefore need a mechanism to prevent variable changes from being automatically turned into animation events. In the following, we show how to modify the Bubblesort visualization code presented in Section 3 in order to reduce the granularity in the swap animation.

Polka. Making the two blocks exchange positions simultaneously, rather than sequentially, is straighforward in Polka. We simply schedule their movement Actions to commence at the same animation time, and then we animate after that. The code below, when substituted into the `Exchange` animation routine of Section 3, performs this concurrent animation.

```
    int len = blocks[i]->Program(time,&a);
    blocks[j]->Program(time,b);
    time = Animate(time,len);

```

Notice that we have reduced the number of visualization instructions in order to reduce the number of animation events.

Leonardo. Leonardo provides a simple mechanism for controlling the granularity: if the predicate `ScreenUpdateOn` is declared, then variable changes trigger automatically updates of the visualization. If `ScreenUpdateOn` is not declared, then the visualization system is idle and no animation events occur. To let each swap in our example produce just one animation event, we can temporary suspend screen updates while the swap occurs: we just "undeclare" `ScreenUpdateOn` before the swap, and redeclare it thereafter, as shown below.

```
/** Not ScreenUpdateOn; **/
int temp=v[i]; v[i]=v[i-1]; v[i-1]=temp;
/** ScreenUpdateOn; **/
```

Notice that we have increased the number of the visualization instructions in order to reduce the number of animation events.

4.2 Accessing vs. Modifying Data Structures

Sometimes we might be interested in visualizing actions of an algorithm corresponding to no variable modification: consider, for instance, events of comparison of two elements in a sorting algorithm to decide whether they need to be swapped. It is easy to animate such actions with interesting events, which can be associated to any conceivable algorithmic event. On the contrary, this seems to be a major problem with state mapping, where animation events can result only from variable changes.

Polka. Suppose that we wish to illustrate the comparison of two array elements to determine whether they need to be exchanged. To do so, we add a new interesting event named "`Compare`" to the Bubblesort source code. This event occurs just before the actual value comparison is made in the program, and it passes the two pertinent array indices as parameters. Below, we show the modified program source.

```
int v[] = {3,5,2,9,6,4,1,8,0,7}, n=10, i, j;
void main(void) {
    bsort.SendAlgoEvt("Input",n,v);
    for (j=n; j>0; j--)
        for (i=1; i<j; i++) {
            bsort.SendAlgoEvt("Compare",i,i-1);
            if (v[i-1] > v[i]) {
                int temp = v[i]; v[i] = v[i-1]; v[i-1] = temp;
                bsort.SendAlgoEvt("Exchange",i,i-1);
            }
        }
}
```

Let us suppose that we want to illustrate the comparison operation in the program by flashing the two corresponding rectangles in the animation. This is performed in Polka by modifying the fill value of the `Rectangle AnimObjects`. Originally, the rectangles have a fill value of 0.0, indicating that they are simply outlines (1.0 signifies solid color fill, and values in between correspond to intermediate fills). We create a “`FILL`” animation action of two frames that takes the rectangle from empty, to half-filled, and back to empty. We then make a new `Action` that is this simple fill change iterated four times, thus making the flashing effect more striking. We schedule this behavior into both rectangles and perform the animation.

```
int Rects::Compare(int i, int j) {
    double flash[2];
    flash[0] = 0.5;
    flash[1] = -0.5;
    Action a("FILL",2,flash,flash);
    ActionPtr b = a.Iterate(4);
    int len = blocks[i]->Program(time,b);
    len = blocks[j]->Program(time,b);
    time = Animate(time, len);
    return len;
}
```

Leonardo. A general way to illustrate a comparison event in Leonardo is to “simulate” an interesting event. To do so, we add to the Bubblesort program a new C function, `void Compare(int i,int j)`, which takes the two pertinent array indices as parameters. This function is invoked just before the comparison and highlights the elements being compared, very much like the interesting event “`Compare`” does in the Polka code.

```
1. void Compare(int i, int j) {
2.    int k=0;
3.    /** RectangleColor(ID,Out Grey,1) If (ID==i || ID==j) && k%2; **/
4.    while (k<8) k++;
5. }
```

The main idea is to create a dummy sequence of variable changes: to this aim, we declare a local variable `k`, whose value flips four times from even to odd and back to even (line 4). The declaration in line 3, whose scope is local to the function body, just states that rectangles with `ID`s equal to `i` or to `j`, corresponding to the elements being compared, must have solid gray color fill whenever `k` is odd (i.e., `k%2` is non-zero). Even if this unusual method for defining animations may seem strange at first sight, mixing declarative and imperative specification yields great flexibility in the animation design.

Notice that variable i in the Bubblesort code indicates which items are currently being compared: thus, highlighting them would be easy in a pure declarative style. However, this would be an indirect way of portraying comparison events and might not be applicable to other algorithms.

4.3 Visualizing Invariant Properties of Algorithms

An important issue in algorithm visualization is portraying invariant properties of a program, which usually provide a sound foundation to the algorithm's correctness or performances. Visualizing invariant properties can help discover implementation errors and foster a better comprehension of combinatorial, algebraic, or numerical aspects of the problem at hand.

An interesting invariant property of the Bubblesort code shown in Section 3 is that array elements with indices greater than or equal to `j` are always at their final proper positions. We note that, since eventually `j` gets equal to zero, this implies the correctness of the whole procedure. To highlight sticks corresponding to elements that are properly positioned "in place", we color them red.

Polka. We create a new interesting event titled "`InPlace`" taking one parameter, the index of the array value now at its final position. We insert this event at the end of the outer of the two main loops in the code. We also must add one final event at the very end of the algorithm.

```
int v[] = {3,5,2,9,6,4,1,8,0,7}, n=10, i, j;
void main(void) {
    bsort.SendAlgoEvt("Input",n,v);
    for (j=n; j>0; j--) {
        for (i=1; i<j; i++) {
            bsort.SendAlgoEvt("Compare",i,i-1);
            if (v[i-1] > v[i]) {
                int temp = v[i]; v[i] = v[i-1]; v[i-1] = temp;
                bsort.SendAlgoEvt("Exchange",i,i-1);
            }
        }
        bsort.SendAlgoEvt("InPlace",j-1);
    }
    bsort.SendAlgoEvt("InPlace",0);
}
```

To indicate that an array element is in place, we change it from a simple outline to a solid colored rectangle and we change its color to red. This animation routine uses the "`FILL`" `Action` much as the `Compare` animation routine did as well as a simple color change `Action`. We schedule both to occur at the same time, and a one frame animation results.

```
int Rects::InPlace(int i) {
    double f = 1.0;
    Action a("FILL",1,&f,&f);
```

```
        Action b("COLOR","red");
        blocks[i]->Program(time,&a);
        int len = labels[i]->Program(time,&b);
        time = Animate(time, len);
        return len;
    }
```

Leonardo. To achieve the same result in Leonardo, we just need to add the following declaration to the ALPHA code given in Section 3.

```
        RectangleColor(ID,Out Red,1) If ID>=j;
```

Notice that the declarative specification allows us to encode directly the invariant property described above: here we state that rectangles in window 1 having `ID` greater than or equal to `j`, which correspond to array elements with indices greater than or equal to `j`, should have red color.

4.4 Adding Smooth Animation

Smooth animation is a very useful addition to algorithm visualizations for creating continuity in the display and for capturing the user's attention. In this section we address the problem of specifying smooth animations using interesting events and state mapping. In particular, we modify the Bubblesort example to visualize swaps as smooth transitions along curved paths, rather than jerky movements. While obtaining the desired solution with interesting events is straightforward in Polka, the Leonardo implementation involves some subtle considerations.

Polka. Changing the exchange operation's movement animations from being one frame "jumps" to smooth, multiframe, curved motions is very easy with Polka. We simply change the one line of the `Exchange` animation routine that constructs the movement path. We use a different `Action` constructor, one that utilizes the predefined `CLOCKWISE` trajectory taking 20 animation frames.

```
      Action a("MOVE",loc1,loc2,CLOCKWISE);
```

Leonardo. If the predicate `SmoothAnimationOn` is declared, Leonardo provides automatic in-betweening of graphical scenes: changes of graphical objects by the same IDs in consecutive scenes are automatically linearly interpolated to generate intermediate frames. In order for this to work properly, no two graphical objects can have the same ID.

We now consider the animation effect obtained by simply adding to our Bubblesort visualization code the following declaration.

```
      SmoothAnimationOn;
```

Since a rectangle with `ID=x` has height proportional to `v[x]` in our implementation (see line 5 of the visualization code in Section 3), swaps are seen from the viewpoint of array slots, which get their content changed. Thus, the swap animation resulting from declaring `SmoothAnimationOn` is that one stick grows and one shrinks. This might be fine anyway, but we were expecting sticks to jump, not change in size as in an animated histogram.

To customize the behavior to the desired result, we can look at swaps from the viewpoint of elements that move from slot to slot: to do so, we just let `ID=v[N]` instead of `ID=N` in line 5 of the visualization specification. In this way, rectangles by the same ID may have different positions before and after a swap, but same size. To meet the requirement that no two graphical objects can have the same ID, now we have the constraint that the array must contain no duplicates. We might also need to revise previous declarations that assumed `ID=N`, e.g., predicate `RectangleColor` in Section 4.3. At this point, to move sticks along curves instead of straight lines, we can further customize the animation, adding the following declaration.

```
 8.     RectanglePosPath(ID,Out Curve,1);
```

Here we declare that any change of position of a rectangle in window 1, regardless of its ID, must be interpolated along a curve. Notice that the animation behavior of a graphical object is specified in Leonardo as an attribute of the object itself.

4.5 Other Issues

Showing Computation History. While invariant properties of programs are easily visualized with state mapping since they are defined on the current computation state, visualizing the history of the computation may be difficult, unless the current state of the computation includes some information about previous states. In fact, the solution usually adopted with both interesting events and state mapping is to record some history of previous states in a data structure, which is accessed for generating the visualization. Another possible solution with state mapping is to record the history in the mapping itself, which is progressively enriched with new declarations as the program runs.

Animating Multi-phase Algorithms. Some algorithms are based on several internal phases, each of which should be visualized in a different way (see, e.g., Ford-Fulkerson's maxflow algorithm [1]). This is achieved quite naturally with interesting events. Visualizing multi-phase algorithms with a pure state mapping, instead, may be difficult: the problem is easily solved, however, if the system allows us to let the graphical interpretation of variables depend upon the portion of code that is currently being executed. Leonardo, for instance, provides ad-hoc directives (`Not x`, `Assert x`, `Negate x`, `Substitute x With y`) that can activate, deactivate, or replace previous declarations at any point of the algorithmic code. We remark that, even if this is still state mapping, it is not "pure" in the sense that the set of active declarations defining the mapping depends on the program's control flow and is manipulated in an imperative style.

5 Conclusions

In this paper we have addressed specification aspects in algorithm visualization, providing the first direct comparison of the two most commonly used specification methods: interesting events and state mapping. We have based our discussion on specifying the well-known sticks visualization of the Bubblesort algorithm in the systems Polka and Leonardo, which instantiate the two approaches.

While interesting events are very intuitive and well-suited for specifying highly customized animations, they usually require developers to write several lines of additional code even for basic animations, and may lack in code ignorance allowance. On the other side, specifying visualizations with state mapping usually requires developers to write few lines of additional code, and little knowledge of the underlying code is needed, but this method may have a steeper learning curve and appears to be less flexible than interesting events in some customization aspects. It is our opinion that devising systems able to support both the declarative and the imperative visualization specification styles would represent an interesting research contribution, likely to be best suited for deployment in concrete applications.

References

1. R.K. Ahuja, T.L. Magnanti, and J.B. Orlin. *Network Flows: Theory, Algorithms and Applications.* Prentice Hall, Englewood Cliffs, NJ, 1993.
2. J. Bentley and B. Kernighan. A System for Algorithm Animation: Tutorial and User Manual. *Computing Systems*, 4(1):5–30, 1991.
3. M.H. Brown. *Algorithm Animation.* MIT Press, Cambridge, MA, 1988.
4. M.H. Brown. Perspectives on Algorithm Animation. In *Proceedings of the ACM SIGCHI'88 Conference on Human Factors in Computing Systems*, pages 33–38, 1988.
5. M.H. Brown. Zeus: a System for Algorithm Animation and Multi-View Editing. In *Proceedings of the 7-th IEEE Workshop on Visual Languages*, pages 4–9, 1991.
6. M.H. Brown and M. Najork. Collaborative Active Textbooks: a Web-Based Algorithm Animation System for an Electronic Classroom. In *Proceedings of the 12th IEEE International Symposium on Visual Languages (VL'96)*, pages 266–275, 1996.
7. P. Carlson, M. Burnett, and J. Cadiz. Integration of Algorithm Animation into a Visual Programming Language. In *Proc. Int. Workshop on Advanced Visual Interfaces*, 1996.
8. G. Cattaneo, U. Ferraro, G.F. Italiano, and V. Scarano. Cooperative Algorithm and Data Types Animation over the Net. In *Proc. XV IFIP World Computer Congress, Invited Lecture*, pages 63–80, 1998. To appear in Journal of Visual Languages and Computing. System home page: `http://isis.dia.unisa.it/catai/`.
9. P. Crescenzi, C. Demetrescu, I. Finocchi, and R. Petreschi. Reversible Execution and Visualization of Programs with LEONARDO. *Journal of Visual Languages and Computing*, 11(2):125–150, 2000. Leonardo is available at the URL: `http://www.dis.uniroma1.it/~demetres/Leonardo/`.

10. C. Demetrescu, I. Finocchi, and G. Liotta. Visualizing Algorithms over the Web with the Publication-driven Approach. In *Proc. of the 4-th Workshop on Algorithm Engineering (WAE'00)*, LNCS 1982, pages 147–158, 2000.
11. K. Mehlhorn and S. Naher. *LEDA: A Platform of Combinatorial and Geometric Computing.* Cambrige University Press, ISBN 0-521-56329-1, 1999.
12. B.A. Myers. Taxonomies of Visual Programming and Program Visualization. *Journal of Visual Languages and Computing*, 1:97–123, 1990.
13. B.A. Price, R.M. Baecker, and I.S. Small. A Principled Taxonomy of Software Visualization. *Journal of Visual Languages and Computing*, 4(3):211–266, 1993.
14. G.C. Roman and K.C. Cox. A Declarative Approach to Visualizing Concurrent Computations. *Computer*, 22:25–36, 1989.
15. G.C. Roman and K.C. Cox. A Taxonomy of Program Visualization Systems. *Computer*, 26:11–24, 1993.
16. G.C. Roman, K.C. Cox, C.D. Wilcox, and J.Y Plun. PAVANE: a System for Declarative Visualization of Concurrent Computations. *Journal of Visual Languages and Computing*, 3:161–193, 1992.
17. A. Schliep and W. Hochstättler. Developing Gato and CATBox with Python: Teaching Graph Algorithms through Visualization and Experimentation. In *Proceedings of Multimedia Tools for Communicating Mathematics (MTCM'00)*, 2000.
18. J.T. Stasko. TANGO: A Framework and System for Algorithm Animation. *Computer*, 23:27–39, 1990.
19. J.T. Stasko. Animating Algorithms with X-TANGO. *SIGACT News*, 23(2):67–71, 1992.
20. J.T. Stasko. A Methodology for Building Application-Specific Visualizations of Parallel Programs. *Journal of Parallel and Distributed Computing*, 18:258–264, 1993.
21. J.T. Stasko, J. Domingue, M.H. Brown, and B.A. Price. *Software Visualization: Programming as a Multimedia Experience.* MIT Press, Cambridge, MA, 1997.