Visualizing Concurrent Computations

Kenneth C. Cox  Gruia-Catalin Roman
Department of Computer Science
Washington University
Saint Louis, MO 63130, USA

Abstract
This paper describes the conceptual model and specification method for a visualization environment concerned with exploring, monitoring, and presenting concurrent computations. The model is declarative in that visualization is treated as a mapping from program states to a three-dimensional geometric world. The state-to-geometry mapping is defined as a composition of several simpler mappings. This paper shows how this decomposition was molded by two methodological objectives: (1) the desire to visually capture abstract formal properties of programs and (2) the need to support complex animations of atomic computational events. A termination detection algorithm is used to illustrate the specification method and to demonstrate its conceptual elegance and flexibility.

1. Introduction
During the past decade visualization has become an indispensable scientific tool. Researchers in many fields use images to cope with large and dynamic data volumes gathered through instruments or created by simulations. A single image is able to convey vast amounts of information about the physical phenomenon under consideration. The abstractive power inherent in visual representation and the innate human ability to rapidly process very large volumes of visual information are the keys to the success of scientific visualization [6].

The same arguments can be made on the behalf of program visualization which, within the scope of this paper, is defined as the graphical presentation, monitoring, and exploration of concurrent computations. These activities, intrinsic to the software engineering process, also involve large volumes of information of a highly dynamic nature. Moreover, both the amount of information that must be examined and the rate of change associated with this information increase with the degree of concurrency exhibited by the system under consideration.

It is our contention, however, that concurrent program visualization is faced with intellectual challenges that are much greater than those encountered in scientific visualization. In the scientific domain, one visualizes data which is associated directly with some physical phenomenon which, in turn, dictates a particular visual representation. In concurrency, the phenomenon to be explored by visual methods is the computation itself. Most often, there is no output data and the operational or structural details are too low-level and localized to be useful in reasoning about the computation. Since understanding a sequential algorithm requires something more than the ability to read each line of code it executes, it should come as no surprise that understanding a concurrent computation requires something more than just seeing what each component does.

Our research is concerned with the development of a methodology, rooted in a strong formal foundation, for selecting proper visualizations of concurrent computations. The particular approach we are currently exploring could be characterized in general terms as a proof-based visualization—the visualization focuses on abstract formal properties of the computation and not on structural or operational properties of the program. We justify our selection of this approach by observing that in the concurrent domain operational thinking is often rendered ineffective by the very large number of possible interleavings of events. Assertional reasoning [2] has emerged as an effective tool for acquiring valuable insight about the workings of such programs—today, a concurrent program without a proof is hardly given any serious consideration. Since program understanding is also a primary objective of visualization, the connection between proofs and visualization seems in principle to be a natural one. We showed previously [8] that there is also a way to exploit this relation at a practical level.

This paper is concerned with the conceptual model and specification methods employed by a program visualization system called Pavane. Its goal is to facilitate exploration and validation of methodologies for visualizing concurrent computations. Section 2 gives the Pavane model and the method whereby visualizations are specified. Section 3 specifies a complete visualization for a simple concurrent computation. Section 4 provides a brief summary and conclusions.
2. Specifying visualizations

In a system like BALSA [1], visualizations are built by identifying in the program code places where significant events occur and by annotating the code with calls to procedures that manage the visualization. In effect, these procedures provide a low-level constructive method for mapping computational events into complex sequences of visual events. By contrast, visualizations in Pavane are specified abstractly, as mappings from computational states to images on the screen. This declarative approach is attractive because visualizations are specified and modified easily and because visualization is uncoupled from the program code. Pavane is the first system to exploit the full power of the declarative visualization paradigm. Prior systems, e.g. PROVIDE [7] and PVS [5] used only simple mappings from variables to icon attributes. Difficulties associated with formally and elegantly specifying the state of programs written in traditional block-structured languages may explain why declarative methods have failed to play a significant role in visualization efforts to date.

The Pavane model assumes that the computation whose behavior is being visualized (called the underlying computation) can be characterized by a state, and that this state undergoes a series of atomic transformations during computation. We use the Swarm [9] model to formulate and simulate the computation. In Swarm, the state is represented by a set of tuples called the dataspace; the state is modified by atomic transactions which cause the insertion and deletion of tuples. Swarm has two attractive features for our work: all information about the computation is readily-accessible through the dataspace, and Swarm has a formal proof system [3] which is essential to our methodological investigations.

One of our primary requirements for Pavane was the ability to capture highly abstract properties of the visualized computation. An attempt to transform the state to an image using only a single mapping would be unnecessarily restrictive and might prevent expression of the potentially complex mappings required. Pavane therefore uses a compositional approach, in which the mapping from state to visual representation is given as a composition of several simpler mappings.

All mappings used in Pavane are defined over named sets of tuples called spaces. The tuple notation provides a single uniform representation for data. Each space has an associated collection of tuple types; each tuple type may occur in only one of the spaces. Two spaces are used in all visualizations. The state space represents the computation state, and the animation space represents the final visual output. The overall mapping is from the state space to the animation space. An additional mapping, called the rendering mapping, transforms the animation space tuples into sequences of images.

Conceptually, each transformation of the dataspace triggers an instantaneous recomputation of the entire mapping, leading to appropriate changes to the image. In order to accomplish this, a visualization system which is not fast enough to keep up with the computation must slow down the computation using a flow-control protocol. This is possible on experimental platforms such as Pavane; a production system would need to use high-performance algorithms. The Pavane model is well-suited to highly efficient parallel implementations.

Each mapping has an input space and an output space. Operationally, the mapping generates a new version of its output space from the current and previous versions of its input space and from the previous version of its output space (see Figure 1). A mapping which examines both the current and previous instances of its input space is referred to as differential; note that this permits the detection of events. A mapping which examines the previous instance of its output space is referred to as history-sensitive. This term has been selected to evoke the common use in program proofs of information about the history of the computation. A mapping's current and previous input space and its previous output space are collectively referred to as the query spaces of the mapping; its current output space is its production space.

Mappings are composed by identifying the output space of one mapping as the input space for the next. We restrict the composition so the final configuration is a pipeline in which the input space of the first mapping is the state space and the output space of the final mapping is the animation space.

We specify each mapping as a collection of rules. Each rule may be seen as a logical relationship between
the input and output spaces given in terms of two predicates \( Q(v) \) and \( P(v) \):

\[
\forall v : Q(v) \Rightarrow P(v)
\]

These predicates may include tests for the presence of tuples in spaces. When such tests appear in \( Q(v) \), they must represent tests for tuples in the query spaces of the mapping. \( P(v) \) is restricted to the form of a simple test for a list of tuples in the production space. Considered as a predicate, the rule as a whole thus indicates that for all instantiations of the variables \( v \) such that the predicate \( Q(v) \) is true (for some given query spaces), the predicate \( P(v) \) must also be true of the production space.

A mapping is defined as a finite set of rules. The production space of a mapping for some given query spaces is by definition the smallest set satisfying all the relationships required by the set of rules. Operationally, one may treat \( Q(v) \) as a query over the query spaces and \( P(v) \) as a list of tuples contributed by the particular rule to the production of the output space. The result of applying a mapping (a collection of rules) is simply the union of all sets of tuples produced by the rules of the mapping.

For notational convenience Pavane drops the universal quantification, allowing the visualization rules to be simply stated as

\[
v : Q(v) \Rightarrow P(v)
\]

The space to which a tuple belongs does not need to be specified in the rule; because of the restriction that tuple names are disjoint across the several spaces, the space of a tuple can be determined from the name. History-sensitive mappings require additional notation to distinguish the current and previous instances of the input space. We choose to prefix tuples from the previous input space configuration with the qualifier old.

Figure 1 depicts the specific visualization mapping decomposition which we use for our investigations. The state space is the state of the underlying computation; the proof space is an abstraction of the program state obtained by eliminating irrelevant details and by accumulating historical data; the object space defines a three-dimensional geometric world; finally, the animation space represents objects which may be painted on the screen. The proof mapping (producing the proof space) and object mapping are history-sensitive. The animation mapping is differential; it detects visual events and translates them into sequences of images.

3. An example visualization

In this section we apply the Pavane model and our proof-based methodology to a simple concurrent program to construct a visualization. The specific Pavane model we use is that illustrated in Figure 1; in this model, the overall visualization has been decomposed into three stages, termed the proof, object, and animation mappings. The reasoning which led to our use of this particular division is also explained in this section.

We will use the Diffusing Computations Problem to illustrate the Pavane model. This problem involves a collection of processes which communicate by sending messages over channels, and an external environment process which communicates with some of the processes. Each process is active or inactive; only active processes can send messages. An active process may become inactive at any time; an inactive process becomes active only on the receipt of a message. Computation begins when the environment sends messages to one or more processes and ends when all processes are inactive and the environment is not sending any more messages. The problem is for the environment to detect the termination of the computation, i.e., when all processes are inactive and no further messages can be received by any process.

One solution, proposed by Dijkstra and Scholten [4], is to have each process acknowledge every message it receives. A process that is inactive records the identity of the process (henceforth called its parent process) whose message caused a transition from the inactive to the active state. Acknowledgement of that particular message is delayed until the process returns to the inactive state; further, a process delays its return to the inactive state until it has received acknowledgments for every message it has sent. An active process immediately acknowledges all messages it receives. The environment detects termination when it receives acknowledgments for every message it sent.

3.1. Representation of the computation

Space limitations prevent us from presenting the full formulation of the diffusing computations problem in Swarm; the interested reader is invited to contact the authors. We present only the representation of the state.

The environment is represented by two tuples. The first tuple is of the form \( \text{Env}() \); this tuple is present whenever the environment process is active. The second tuple is of the form \( \text{environ}(m,a) \); the components of this tuple represent the number of messages \( m \) that the environment has yet to send and the number of acknowledgments \( a \) that the environment has yet to receive. (In the Swarm formulation, \( \text{Env} \) is actually a transaction and represents an active agent which can modify the dataspace; \( \text{environ} \) is a data tuple and is a passive entity. This distinction is not germane to our exposition.)

Processes are identified by numbers in the range 1 to \( N \) (0 is used for the environment when needed). Each process has an associated tuple \( \text{process}(i, s, a, p) \). This tuple identifies the process \( i \), the process status \( s \) (either active or inactive), the number of acknowledgments \( a \) that the process must receive before it may become inactive, and the process' parent \( p \). Three transactions, named \( \text{SendMsg}(i), \text{GetMsg}(i), \text{PassAck}(i) \), are also used to represent the various alternative actions that the process \( i \) may take (e.g., \( \text{SendMsg} \) models the sending of a message by an active process).

Finally, the channels are modeled by tuples \( \text{msg}(s, d, n) \) and \( \text{ack}(s, d, n) \). These tuples respectively
indicate the number $n$ of messages and acknowledgments currently "in transit" from source process $s$ to destination process $d$. The number of messages is sufficient information for the purpose of modeling the diffusing computations problem.

3.2. Proof mapping

As a methodological foundation for visualization, we have made the assumption that those properties of the program which are important in verifying its correctness are also properties which should be visualized. We have constructed a number of visualizations using this methodology and have found it to be quite successful as a method for constructing meaningful visualizations. Accordingly, in our model we refer to the first stage of visualization as the proof mapping. Its function is to extract from the program state only those aspects relevant to visualization. The output space of this mapping is called the proof space (see Figure 1).

State extraction is obviously of great importance in program visualization. Our choice of Swarm as a computational model greatly simplifies the problem of extraction—all components of the state, both data and control, are contained in the Swarm dataspace and easily accessible. When visualizing computations which are not expressed in Swarm, a mechanism must be provided for monitoring the program state and for communicating it to Pavane. In this context, the proof mapping can be considered as a specification of what information about the state must be extracted and how the information should be presented to later stages of the visualization.

Several considerations led us to make the proof mapping a history-sensitive mapping. One factor which led to this decision was our postulated proof-based methodology for constructing visualizations. It is often the case that proofs rely on the use of auxiliary variables which maintain information about the computation. Therefore, the proof rules need to access the old values of the auxiliary variables—in other words, the previous proof space—in order to compute the new ones.

We use as a starting point for our visualization of diffusing computations one of the key invariants required in the correctness proof: the parent information in those process tuples whose status is active defines a directed tree rooted at the environment. Note that at this point, the term "tree" refers to a graph-theoretic entity and not to a geometric representation. However, the correspondence between a graph-theoretic tree and its depiction is a natural one and serves as the basis for our visualization.

Not all tuples and transactions appearing in the diffusing computations program are essential to capturing this invariant. In the proof mapping, we isolate those aspects of the computation which are of interest. In this case, we wish to collect the information needed to describe the connectivity of the tree (edge definitions). In addition to extracting the parent-child relations from the state space, we also use the proof space to compute the distance of each node from the tree's root; this information is implicitly contained within the tree structure, but to generate a graphical representation of the tree we need to have explicit values. This illustrates another use of the proof space: the calculation of values required to support later stages of the visualization.

The connectivity and distances from the root are represented in the proof space by vertex tuples containing for each node in the graph the node's id, the id of the node's parent, and the distance of the node from the root (the environment). To generate the distance information, we make use of a known property of the underlying computation: a process can become active in a single atomic action only if its parent was already active. Therefore, a node can be added to the tree only if its parent was present in the tree in the previous step; we can compute the node distance by querying the previous proof space to determine the distance of its parent.

Two rules accomplish the proof mapping. The first puts a node representing the environment into the tree:

\[
\text{Env}(\text{id}) \Rightarrow \text{vertex}(0, 0, 0)
\]

Technically the environment's parent is undefined; by convention we will use 0 for the parent. The second rule creates tuples representing active processes. The query part of the rule examines the previous proof space to determine the distance of the node's parent from the root:

\[
\text{id, ack, p, pp, dist: process(id, active, ack, p), vertex(p, pp, dist) } \Rightarrow \text{vertex(id, p, dist + 1)}
\]

3.3. Object mapping

The next stage is the transformation of the properties contained in the proof space into a three-dimensional geometric representation. The object mapping accomplishes this task, producing a collection of tuples called the object space. The inputs to the object mapping are the current proof space and the previous instance of the object space. The tuples of the object space represent abstract geometric objects. They define an abstract three-dimensional world of objects which lacks a concrete visual representation, because the abstract objects are not necessarily in one-to-one correspondence with the graphical objects that make up the final image. Each tuple in the object space may eventually be mapped to several graphical objects of the final image, or a single graphical object may be described by parameters drawn from several tuples of the object space. This flexibility is an important part of our model, as it allows the constructor of the visualization to work with concepts that are most suitable for the task, without premature commitment to a particular visual representation.

Returning to our example, we construct the tree in three dimensions. (We are now using "tree" to refer to a geometric construction.) Assume that we have some layout of the underlying communications graph in the plane—this is not to say that the graph is planar, merely that we have some arrangement of its vertices in the
plane. The X- and Y-coordinates are obtained from the corresponding coordinates in this graph, while the Z-coordinate corresponds to the distance of the node from the root (environment). Pavane provides mechanisms for the definition and use of functions. Assuming we have defined functions nodex, nodey, and nodex which produce the coordinates, the tree is constructed using two types of components: nodes and links between nodes. The following rules generate these two types of abstract objects, without yet indicating how they might be represented (three-dimensional coordinates are represented by triples enclosed in brackets):

- vertex(0, 0, 0)  
  => node_object([nodex(0), nodey(0), nodez(0)])

- id, pid, dist, ppid, pdist:
  vertex(id, pid, dist), vertex(pid, ppid, pdist)

- node_object([nodex(id), nodey(id), nodez(dist)]),
  link_object([nodex(pid), nodey(pid), nodez(pdist)])

All that we currently specify is that the nodes are abstract objects associated with a particular point and the links are objects characterized by two points. The first rule creates the root node at the coordinate of the environment. The second generates a node for each non-root node (i.e., those which have a parent node) and a link between the node's coordinate and that of the node's parent.

3.4. Animation mapping

The third stage of the visualization mapping involves the translation of abstract objects produced by the object mapping into a form suitable for rendering. We call the objects which can be rendered primitive graphical objects. Pavane's current repertoire of primitive graphical objects includes points, lines, rectangles, polygons, circles, and spheres; this vocabulary can be extended by augmenting the capabilities of the rendering engine. The primitive graphical objects are produced by the animation mapping which produces an animation space in which each tuple represents exactly one primitive graphical object.

As suggested by the name, the animation mapping also serves to smooth the transition between successive representations of the state, as well as to provide "special effects" (such as flashing). The smoothing of transitions is aesthetically very important; without it, a visualization has a jerky appearance and is difficult to follow, as parts of the image simply appear and disappear in response to changes in the state.

Animation is not possible unless one is able to recognize events. This does not mean, however, that computational events must be detected. Our model does not preclude the use of animation, but moves the concern with special effects into the visual realm: we treat these effects as visual events. For example, an exchange of the positions of two objects on the screen might be the result of a change in the state of the underlying computation. The animator can detect this visual event (the exchange of objects) and render it as a sequence of simpler visual events involving movement along some prescribed trajectories. In this manner we are able to preserve the formality of state-based mapping without sacrificing appearance.

The tuples of the animation space represent graphical objects in a 4-D space, the three spatial dimensions plus time. The tuple type corresponds to the object type (line, sphere, etc.). The tuple components give to the various attributes of the object; for example, a line object has attributes lifetime, from, to, and color. Attributes can be time-dependent, with time being measured in terms of display frames starting at 0. Functions are provided to generate time-dependent values; for example, ramp generates a linear interpolation. Mechanisms for composing the effects of the functions are also provided.

For notational convenience, animation space tuples have the form

- type( attribute = value, attribute = value, ...)

where each attribute is an object attribute and each value is a value of the appropriate type. It is not necessary to specify values for all the attributes of the object; defaults are used for unspecified values. For example, the tuple line(from = [0,0,0], to = [5,5,5]) represents a line whose endpoints are [0,0,0] and [5,5,5]; the line will be colored white (the default color value).

In the diffusing computations example, we can represent each node by a sphere and each link by a line. Two rules which accomplish this are:

- coord:
  node_object(coord)
  => sphere(center = coord, radius = sphererad)

- ncoord, pcoord:
  link_object(ncoord, pcoord)
  => line(from = ncoord, to = pcoord)

Note that we permit unification in our rules; thus, the variables in the above rules are unified with coordinates. In addition, note the use of the constant sphererad; Pavane permits constants to be defined and used in the same way as functions.

Figure 2 illustrates a possible sequence of images that the above rules could produce; each image corresponds to a state of the underlying computation. These images are shown from a point on the X-Y plane of the coordinate system; the positive Z-coordinate is directed "upward" in these images. Note that with these rules, this sequence of frames is exactly what the viewer would see; lines and spheres would appear and disappear, resulting in a somewhat unaesthetic visualization.

We can improve the appearance of the visualization by animating the addition and removal of the spheres and lines. This is accomplished by recognizing three separate cases: the addition of an object, the persistence of an object, and the removal of an object. We need separate rules for each of these cases.

Consider the addition of a node to the tree, resulting in the addition of a sphere and (usually) a line to the image.
Figure 2. A possible sequence of images resulting from application of the visualization rules given in this paper. Time progresses as indicated by the numbers. Steps in which there is no change in the image (e.g., those corresponding to the sending of a message) have been omitted. The first image shows the environment, the second shows that a process has become active with the environment as its parent, and so on. The last two images show the environment detecting termination and becoming inactive.

Assume we want to animate this by having the line grow from the parent sphere until it reaches the position of the new sphere, and the sphere then expanding from a point to its full radius. This is accomplished by the rules:

\[
\text{ncoord, pcoord : } \quad \text{link-object( ncoord, pcoord ),}
\]
\[
\text{old-link-object( ncoord, pcoord )}
\]
\[
\implies \text{line( lifetime = [1, 10], from = pcoord, to = ramp(1, pcoord, 5, ncoord) )}
\]
\[
\text{coord : } \quad \text{node-object( coord ), old-node-object( coord )}
\]
\[
\implies \text{sphere( lifetime = [5, 10], center = coord, radius = ramp(5, 0, 10, sphererad) )}
\]

The first rule detects when a line is added and generates a line which will be present from frame 1 to frame 10 (the lifetime attribute). The from endpoint of the line will be at the center of the parent (location pcoord); the to endpoint will move from the center of the parent to the center of the child between frames 1 and 5, and will remain at the center of the child thereafter. The second rule generates a sphere which is present from frame 5 to frame 10; the sphere’s radius changes from 0 at frame 5 to sphererad at frame 10. Together, the rules achieve the effect of a line growing from the parent to the child, then the sphere growing from that point. The frame numbers were chosen to produce a pleasing effect, and could be represented by named constants like sphererad. Our ability to rapidly change the visualization by changing such constants is quite helpful in development.

We omit the rules for the persistence and the removal of lines and spheres, as they are quite similar to those for the addition. The rule for persistence does not use any time-dependent functions.

One frame from a visualization using these rules is shown in Figure 3. This visualization uses several additional rules that add a representation of the process graph “above” the termination-detection tree; this graph represents (using color) the states of the processes and the message activities on the channels. A seven-minute VHS tape entitled Diffusing Computations is available.

4. Conclusions

This paper presented a new model for visualizing concurrent computations. The key features of the model are its declarative nature, its rule-based notation, its ties to program verification, and the emphasis on three-dimensional visualizations. Our experimentation with Pavane provides evidence that declarative visualization is a viable and attractive paradigm. In particular, this paper shows that a declarative approach, based on mapping states to images, is able to handle elegantly both history-dependent attributes and special effects essential to effective visual communication. Finally, we showed that decomposition of the visualization mapping is useful not only as a complexity control tool but also as a means of formulating particular visualization methodologies.
Acknowledgments

This paper is based on work supported by the DARPA/NASA National Fellowship Program in Parallel Processing under agreement 26947M, and by the National Science Foundation under Grant No. CCR-9015677. The Government has certain rights in this material. The authors acknowledge Timothy J. Boemker’s collaboration in the production of the Diffusing Computations video tape, and thank Jerome Y. Plun and C. Donald Wilcox for their constructive criticisms and review of this paper.

References