Report Number: WUCS-94-9

1994-01-01

# An Evaluation of the Pavane Visualization System

Kenneth C. Cox and Gruia-Catalin Roman

The Pavane program visualization system is an implementation of the declarative paradigm of visualization. After a brief report on the status of the Pavane implementation, we present the results of an evaluation of the usability of Pavane. This evaluation is based on the use of Pavane by its developers to construct program visualizations, on its use in a classroom setting as a tool for examining executing programs, and on its application to some simple scientific visualizations.


# An Evaluation of the Pavane Visualization System

Kenneth C. Cox
Gruia-Catalin Roman

**WUCS-94-09**

April 1994

Submitted to the *1994 IEEE Workshop on Visual Languages.*

# An Evaluation of the Pavane Visualization System

Kenneth C. Cox
kcc@cs.wustl.edu

Gruia-Catalin Roman
roman@cs.wustl.edu

Department of Computer Science
Washington University
St. Louis, MO 63130-4899, USA

## Abstract

The Pavane program visualization system is an implementation of the declarative paradigm of visualization. After a brief report on the status of the Pavane implementation, we present the results of an evaluation of the usabil ity of Pavane. This evaluation is based on the use of Pavane by its developers to construct program visualizations, on its use in a classroom setting as a tool for examining executing programs, and on its application to some simple scientific visualizations.

## 1. Introduction

This paper is a progress report on Pavane, a program visualization system developed at Washington University in St. Louis. Pavane is based on the declarative visualization paradigm [6]. In this paradigm, visualization is treated as a mapping from some domain of interest to a graphical range. All of the various categories of visualization and visualization systems can be considered as examples of this paradigm. For example, in scientific visualization the domain is data obtained from instruments, computer simulations, or both; in the visual display of programs, the domain is the program text; and in program visualization the domain is an executing computation. In each case the purpose of the visualization is to map information from the domain into a collection of graphics. (The paradigm may be extended to the production of general multimedia presentations. In this case, the range of the mapping would include graphics, text, sound, and so forth.)

Most visualization systems do not explicitly use mappings. There are several reasons for this. In many visualization applications (*e.g.*, mapping the data from a weather simulation into a three-dimensional depiction of cloud masses, or transforming the text of a program into a "boxes-and-lines" image for graphical editing) the desired mapping is fixed. In these cases, a hard-coded implementation of the mapping is more efficient than a system which provides general capabilities. However, even among systems designed to provide general mappings — whether program visualization systems such as Zeus [1], or software design tools such as Garden [5] — the mapping is not usually explicit. Pavane was developed partly as an experiment to see if the declarative paradigm could be specifically used as the basis for visualizations.

This paper presents the results of our evaluation of Pavane. After a brief overview and status report (Section 2), we describe the results of our evaluation. This evaluation was in three main parts. The first (Section 3) was performed by the system developers and consisted of using the system to construct visualizations of a large number of algorithms, both sequential and concurrent. The second stage (Section 4) used Pavane as an adjunct to several classes. The Pavane users were students with a computer science background, and the applications involved visualization of a variety of computations. The third phase of the evaluation (Section 5) opened the system to other applications, including medical and similar scientific visualizations. Although Pavane was not specifically designed for scientific visualization, the declarative approach proved to be general enough to accomodate them and produce useful results. We include some examples of the educational and scientific applications on the last page of this paper, in the form of grayscale images from the visualizations.

Pavane was evaluated on the basis of several criteria. Among these were the comprehensiveness of the declarative model (i.e., whether or not the desired visualizations could be constructed in the declarative paradigm); the clarity and power of the rule-based notation used to specify visualizations; and the usefulness and accesibility of the system. The evaluation led to several changes in the Pavane implementation. However, no changes to the basic concepts of the declarative model were required. We believe this to be a strong indication of the effectiveness and power of the declarative approach.

## 2. System Overview

### 2.1. Visualization Model

The declarative approach is a general paradigm which does not specify the domain or range of the visualization mapping or the manner in which the mapping is specified. Our first task in designing Pavane was thus to specify these aspects of the system.

In the case of Pavane, we selected the state of a computation (called the *underlying computation*) as the domain.

This choice was made for several reasons, the most important being that we wished to use Pavane to explore the visualization of concurrent algorithms. One way to model such algorithms is as global states which are modified by the computation; the state-based visualization model used by Pavane was thus compatible with this type of computational model. In addition, we wished to explore an *analytic* methodology for developing program visualizations [6]. This methodology uses the correctness properties of the program to guide the selection and representation of key properties of the algorithm. Use of the state-based model gave us access to a large number of existing proofs.

We chose to represent the domain (computational state) as a collection of tuples. This was largely inspired by the notation used in the Swarm language [9], which we planned to use for the development of many of our concurrent programs. The tuple-based notation is also flexible and compatible with the state-based computational model. For consistency reasons, we also chose to represent the range of the computation by a collection of tuples. Each tuple in this *animation space* represents a single graphical object in the final image. The components of the tuples represent the object's attributes, and we permit these components to have time-varying values — for example, the *center* of a sphere can change from coordinate $c_1$ to coordinate $c_2$. This allows the production of smooth animation and other visual effects. Such effects are often used to draw the viewer's attention to particular aspects of the computation, and we call such visualizations *explanatory*.

Our final design decision was to choose the means whereby the mapping is specified. We selected a rule-based approach similar to that used in production systems. A mapping is represented by a collection of rules. Each rule specifies a relationship between two sets of tuples called the *input* and *output* spaces of the rule. A rule has the form:

$$\text{name} \equiv \text{variables} : \text{query} \Rightarrow \text{production}$$

The *name* serves to identify the rule. The *variables* are identifiers. The *query* is a predicate which may involve tests for the presence or absence of tuples in the rule's input space and may make use of the *variables*. The *production* is a list of tuples in the rule's output space. The informal semantics of such a rule are, "For every instantiation of the *variables* such that the *query* is true, the tuples in the *production* will be present in the output space." The output space produced by a mapping (set of rules) is just the set of all tuples produced by the rules in the mapping. Conceptually, each time the state of the underlying computation changes the mapping is re-applied to generate a new image which is then displayed.

Our initial work with this system quickly pinpointed several shortcomings and led to corresponding extensions of the basic model. Three major modifications were made to the rule-based notation. First, we permit the overall mapping from the computation's state to the animation space to be decomposed into an arbitrarily-long series of sub-mappings, where the input space of each sub-mapping is the output space of the previous one. This allows the computation of intermediate results at each stage for use by the next — for example, the first sub-mapping might perform a projection to select some data from the state, the second might use this selected data to compute some geometric coordinates, and the final sub-mapping would then use these computed results to generate the image.

The second modification was the addition of a history capability to the mapping. This was accomplished by allowing each sub-mapping to examine not only the current state of its input space but also the previous state of its input and output spaces. The final modification was the addition of a fixpoint rule-application capability. A fixpoint mapping contains one or more fixpoint rules (as identified by the keyword "fixpoint" before their name). When a fixpoint mapping is evaluated, the fixpoint rules are applied repeatedly until the output space of the mapping remains unchanged (reaches a fixpoint). This capability allows each sub-mapping to perform unbounded serial computation.

As we have shown elsewhere [3], both the history capability (memory) and the fixpoint application (unbounded computation) are essential to the construction of program visualizations. This result applies, of course, to all systems, not just to the rule-based approach used in Pavane. We have also shown that Pavane mappings with history and fixpoint capabilities are computationally equivalent to a Turing machine, and thus Pavane can produce any computable mapping.

### 2.2. The Pavane Implementation

The Pavane implementation has undergone many changes since its earliest, Prolog-based version [7]. We describe the system as it currently exists, without attempting to provide its development history.

A typical Pavane visualization consists of three separate UNIX™ processes as shown in Figure 1. These three processes run concurrently, producing "live" visualizations (as distinct from systems which save information about the computation in files and produce the images at a later time). The first process is the underlying computation, which in the current system may be written in Swarm, C, or C++. At each state change in this computation, information about its state is collected and transmitted to the mapping computation which applies the Pavane mapping. The results of the mapping (the animation space) are transmitted to the rendering computation which translates the animation-space tuples into three-dimensional images.

The development of a Pavane visualization requires the creation of the executables for the three processes. The

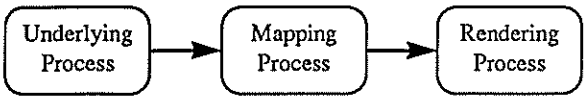


Figure 1. Process structure of a Pavane visualization.

system provides five software components to assist in the production of these executables. These are the compiler *sc*, the three libraries *SwarmLib*, *CtoVis* and *VisLib*, and the *Display* program.

*sc* is used to translate Swarm programs and Pavane mappings into compilable C++ code. *SwarmLib* is linked with the C++ code produced by *sc* from Swarm code to create an executable underlying computation in Swarm. *CtoVis* is linked with C or C++ code to create an executable underlying computation in C or C++. *VisLib* is linked with the C++ code produced by *sc* from Pavane mappings to create an executable mapping computation. *Display* is the rendering computation. All of these components are implemented in C++. The Display program uses the Silicon Graphics *gl* library and runs only on the SGI line of graphics workstations. The relationships between the Pavane system components and the processes of the visualization are illustrated in Figure 2.

A complete description of the implementation of each of these components is not possible in this paper. However, the issue of state-collection should be mentioned. For underlying computations written in Swarm, the collection of the state is automated. Swarm's computational model precisely defines "state" and "state change", and the execution engine (part of the *SwarmLib* library) simply sends the state to the mapping computation after each transition.

The situation is not as simple with C or C++ code, where the portion of the state that is of interest and the meaning of a "state change" cannot be automatically determined. We have adopted an annotative approach to defining these parameters. The animator adds two types of function calls to the code of the underlying computation. These calls access the *CtoVis* library.

The first type of function call is used to define the state. This call defines a mapping between a C program variable and a Pavane tuple. For example, the function call

```
VisualMonitor("sortarray", (void *)sortdata,
    VISMONITOR_PAVANE_ARRAY, nelems,
    VISMONITOR_PAVANE_INTEGER,
        VISMONITOR_C_LONG);
```

indicates that the C array *sortdata* is to be mapped into a Pavane tuple named *sortarray*. The remaining elements of the function call describe the format of the data; in this case it is an array of *nelems* elements, where each element is a C `long` that is to be mapped to a Pavane integer variable. This monitoring facility currently supports C `char`, `short`, `int`, `long`, `float`, `double`, and string (*i.e.*, `char *`) types and arrays and structures built from these types. These annotations cause the *CtoVis* library to store the information about the C variable and the desired mapping to a Pavane tuple.

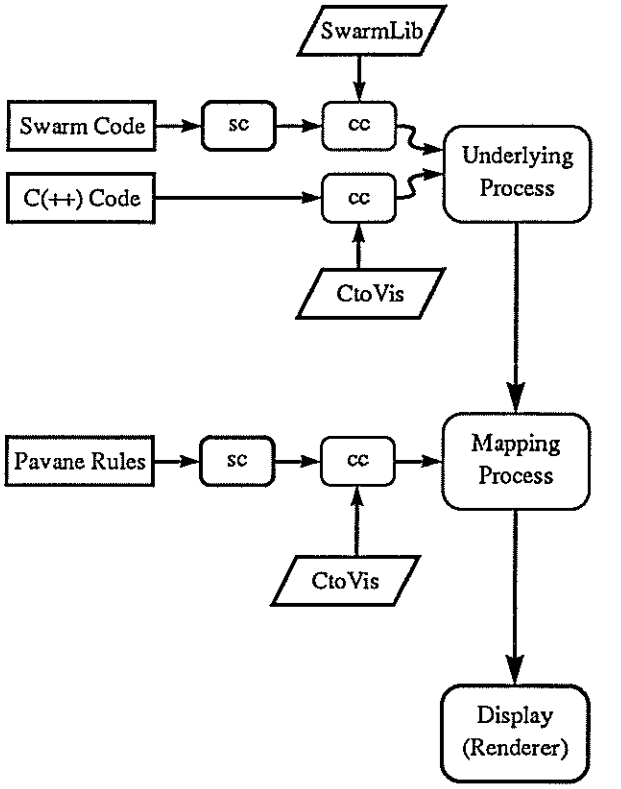


Figure 2. Use of Pavane to construct visualizations.

The second type of annotation involves function calls that indicate when state changes — that is, processing considered to be a single atomic transformation — have occurred. When these function calls are executed, the `CtoVis` library transforms each of the monitored variables into the corresponding Pavane tuple and transmits it to the mapping computation. To continue the above example, if *nelems* is 3 and the following code is executed:

```
sortdata[0] = 6; sortdata[1] = 3; sortdata[2] = 8;
VisualUpdate();
```

the *VisualUpdate()* call will cause the transformation of *sortdata* into the Pavane tuple *sortarray([6,3,8])*. This tuple is then transmitted to the mapping.

This simple method of specifying the state and state changes in C programs could be extended in many ways; for example, mechanisms for automatically monitoring the program could be investigated. However, it was sufficient for the Pavane evaluation. The remaining sections describe the results of this evaluation.

## 3. Evaluation: Algorithm Visualization

To avoid bias in the selection of programs to be visualized, a text (Chandy and Misra's *Parallel Program Design: A Foundation* [2]) containing a large number of algorithms was selected as a source. One of the main reasons for selecting this text was that it also provided correctness proofs for the algorithms, which assisted in our exploration of the analytical methodology for developing visualizations. A few other algorithms were also included in the evaluation, either because they "traditionally" must be included in any visualization system (*e.g.*, sorting algorithms) or because of particular interest to the developers of the system. Implementations of these algorithms were constructed in either the Swarm or C programming languages, using the monitoring libraries described previously, and visualizations were developed in Pavane.

Table 1 presents the results of this phase of the evaulation. This table lists, for each of several algorithms, the number of Pavane rules used in the visualization(s) of the algorithm. The time required to develop each visualization varied considerably, depending on both the size of the visualization and the experience of the animator; most took under two hours (not including the time required to program the algorithm).

| Algorithm | Number of Rules |
|---|---|
| Quicksort | 2-12 |
| Bubble sort | 1-4 |
| Shell sort | 1-3 |
| Batcher's sort | 2-4 |
| Region labeling | 1-2 |
| Bagger (bin-packing) | 3-10 |
| Shortest paths | 2-11 |
| Matrix saddle point | 2 |
| Earliest meeting time | 2-6 |
| Graph reachability | 4-6 |
| Termination detection | 4-10 |
| Diffusing computation | 4-9 |
| Centralized snapshot | 2-4 |
| Distributed snapshot | 3-6 |
| Distributed priority | 1-3 |
| Dining philosophers | 2-4 |
| "Humorous" philosophers | 14 |

Table 1. Algorithm visualizations and rule-counts

In most cases several visualizations were constructed for each algorithm, and the table accordingly presents a range of values for the number of rules. The lower number typically represents a simple, direct mapping from the algorithm state to the image. The higher number corresponds to a more complex visualization, often involving explanatory animations of the transitions between states.

As Table 1 indicates, in most cases only a few rules were sufficient to produce fairly complex visualizations. The largest numbers appearing in this table are for a "humorous" visualization of the dining philosopher's problem (14 rules) and for the quicksort (12 rules). The "humorous" visualization of the dining philosophers algorithm was constructed by taking the problem description literally — the philosophers' plates, forks, and a big platter of "spaghetti" in the middle of the table were shown, and events such as "eating" and "washing forks" were animated. This resulted in a very complex image, with many objects and animation effects.

In the quicksort, we wished to produce a representation of the partition tree induced by the pivot operations. This tree is not part of the quicksort algorithm's state (which consists only of the partially-sorted array of numbers and the ranges that remain to be sorted). The information can be produced from the state using a history-sensitive mapping; six of the twelve rules are concerned with this part of the mapping. The remaining rules draw and animate the three-dimensional tree. An image from this visualization is included in a recent paper [8].

The algorithm-visualization phase of the evaluation of Pavane produced a number of valuable results. Several improvements to Pavane's rule-based model and notation were suggested and incorporated into the system. This phase also produced a number of techniques for developing visualizations, including straightforward methods of converting a non-animated visualization into an animated one. Most importantly, our analytical methodology for constructing visualizations (using the program correctness properties to guide the development process) was shown to be feasible. When this phase was complete, we made Pavane available to other users in order to further evaluate its utility.

## 4. Evaluation: Educational Visualization

Pavane has been used as an adjunct to three classes taught in the Computer Science department of Washington University in St. Louis. Two of these classes were undergraduate software engineering workshops, while the third was a graduate-level course on parallel and concurrent algorithms.

The first application of Pavane was in the Fall 1991 Software Engineering Workshop. This class used Swarm and Pavane in an experiment in the prototyping and visualization of specifications. An airport radar handling system was modeled. This system received input in the form of a grid of pixels, which were processed and analyzed to detect and track planes. In the design phase of the project, the radar handler functionality was broken down into

packages, each handling a distinct group of operations. These packages were then specified as an object-oriented system.

The students translated this system specification into a Swarm program and produced two visualizations. The first showed the operations of a single package. The scope of this visualization was the status of the various objects and messages. The visualization, shown in Plate 1, somewhat resembles those generated by the Object-Oriented Programming System [4], in that it depicts the component software objects and the interactions between these objects. The package is depicted as a large rectangle, with package functions represented by small labeled rectangles extending from the package. Messages are shown as rectangles labeled with the message type, with messages "in transit" arranged on the left side of the image. The receipt of a message and the generation of a response are shown by an explanatory animation in which the message rectangles move to and from the package functions. The second visualization, not shown here, depicted an "operator's console" for the radar handler.

The timing of this experiment was inopportune, in that it began when Pavane was implemented in Prolog [7]. The performance of this interpreted version of Swarm and Pavane was far from satisfactory (indeed, it was largely as a result of this experiment that the move to C++ was initiated). Despite this, the students were quite enthusiastic about the project. All agreed that the ability to actually see their specifications in operation was a major attraction, even if the implementation left something to be desired.

The second application of Pavane was in the Fall 1992 Software Engineering Workshop. This use was similar to the previous one, in that specifications were constructed, transformed into programs, and visualized. However, rather than visualizing the behavior of an object-oriented system, the behavior of an application constructed in such a system — in this case, an elevator control system — was examined. Pavane was in a much more mature state at this time; the initial C implementation was complete, and the CtoVis library was available so the students could use C++ to construct their underlying computations.

In contrast to the highly-abstract visualization of the Fall 1991 project, this visualization depicts the actions of the elevator in detail (Plate 2). Two Pavane windows are used, one to show the external status of the elevator (the floor it is on, the positions of the doors, and the status of the various call lights and buttons) and the other to depict the control panel inside the elevator. The latter led to an immediate complaint from the students, who wanted to be able to control the elevator by clicking on the buttons. Unfortunately Pavane does not (yet) support such interactions, so the students had to use the keyboard to send commands to the underlying computation. Other than this complaint, the students were very satisfied with the system and spent considerable time developing the detailed depiction shown in the figure. (This time was largely "play"; the students completed the initial mapping, which had neither the level of detail nor the animations of the final version, in less than eight hours.)

Visualization was again shown to be useful in formal design work. Several errors in the original specification (including one error which sent the elevator through the bottom of the shaft) were detected using the visualization. The flexibility of the model was also confirmed; the students said that the ability to quickly modify the visualization by changing the rules was especially useful.

The most recent use of Pavane in a classroom setting was in the Fall 1993 Concurrent Algorithms course. In this course, students were given a number of programming projects which were coded in C or C++. The students then visualized the operation of the algorithms using Pavane. A variety of algorithms were selected, ranging from simple parallel merge sorts and matrix transpositions to fairly complex physical simulations. Plate 3 shows one of the simpler examples, a Game of Life program, while Plate 4 shows a relatively complicated automated order-filling program (the operations of the program are depicted using a "robot" which selects items from shelves). Once again, the students spent a lot of time playing with the system and getting attractive or amusing visual effects. The time to develop a visualization (according to student estimates) ranged from three to six hours.

Table 1 shows the number of rules used in the visualizations performed in this phase of the evaluation . Again, for relatively simple problems (such as showing the activities of an object-oriented system, or presenting a merge sort) only a very few rules are required. The larger numbers in this table are due to two factors. The first factor is simply that some of these visualizations are quite intricate, involving multiple windows, many objects, and explanatory animations. The other factor was a lack of familiarity with Pavane's capabilities — in many cases the students used two or three rules to construct a mapping that could have been implemented with a single rule.

| Algorithm | Number of Rules |
|---|---|
| Object-oriented behavior | 6 |
| Elevator control system | 18 |
| Merge sort | 2-4 |
| Data-cycling ring | 2 |
| Matrix transposition | 5-10 |
| Game of Life | 5 |
| Flight simulator | 26 |
| ATM transactions | 26 |
| Automated order-filling | 19 |

Table 2. Educational visualizations and rule-counts

The three educational uses of Pavane clearly indicate the usefulness of the system in this context, with the convenience of the rule-based notation and the ability to rapidly construct prototypes identified by the students as attractive aspects of Pavane. In addition, the first two experiments verified the feasibility of specification visualization and the appropriateness of Pavane for this task. The latter result was not unexpected, as specification visualization is simply a type of program visualization, which is what Pavane is designed to do. The next section examines the application of Pavane to an entirely different type of problem.

## 5. Evaluation: Scientific Visualization

Scientific visualization is the transformation of large data sets representing physical processes into images. The data may result from monitoring devices or may originate from computer simulations. The motivation for scientific visualization is essentially the same as for program visualization: By observing large data volumes as a suitably-abstracted image, the viewer may develop some insight into the underlying process.

Pavane was not originally developed for scientific visualization applications. However, Pavane's state-based declarative model allows it to be applied in this area. The key is to simply treat a collection of time-series data as representing successive states of some computation. Rule-based visualizations can then be constructed in the usual manner to transform these states into images.

Pavane has some obvious limitations insofar as scientific visualization is concerned. One of the most significant was identified in advance of the study. Pavane's rule-based approach is more appropriate for symbolic manipulations than for brute-force number-crunching. We thus expected that its performance on the large data sets typical of scientific visualizations would not be acceptable, but experimental verification was required.

In the Fall of 1992 a letter was sent to various departments of Washington University in St. Louis soliciting researchers who wanted to apply visualization to their current projects. This resulted in the development of a number of visualizations. Since the researchers generally lacked the time to learn enough about Pavane to construct visualizations, their role in this process was limited to providing data sets, suggesting means for representing the data, and observing the results. The visualizations themselves were written by the developers of Pavane.

One aspect of Pavane which all the outside researchers appreciated was the fast development cycle made possible by the rule-based notation. It was possible to try several different means of visualizing the data in the course of an hour, allowing the researchers to explore various presentations of the data.

The results of one such exploration are shown in Plates 5 and 6. These visualizations are from a series produced to examine novel methods of displaying time-series medical data, with the goal of finding new visual representations that bring out patterns of interest. Both visualizations plot two time-varying parameters. Plate 5 is a standard way of plotting this type of data. This visualization was one of our major tests of Pavane's behavior with large data sets; approximately 2000 values are represented by this image. A noticeable degradation in performance was encountered, with updates between images taking two to three seconds. This confirmed our expectations about Pavane's number-processing abilities.

The second visualization (Plate 6) plots the two parameters along two axes — heart rate (HR) on the horizontal axis and chest volume (CV) on the vertical — as a series of rectangles which "recede" from the viewer along the third axis. An opaque box lying along this third axis has dimensions equal to the expected range of variation of these parameters. The box conceals this "normal" portion of the data from view, allowing easy recognition of outlying values as well as perceptions of patterns in their temporal behavior.

Our study involved a number of other visualizations, as summarized in Table 3. (In this table, "Standard medical display" refers to the visualization shown in Plate 5, while the "Novel medical display" is the one in Plate 6.) One notable aspect of this table is the very low rule counts. This result is due to the fact that scientific visualizations usually make use of very direct mappings, since the data represents some physical system.

| Algorithm | Number of Rules |
|---|---|
| Atomic forces | 3 |
| Materials stress | 2 |
| Economic optimization | 1 |
| Magnetic domains | 2 |
| Standard medical display | 6 |
| Novel medical display | 4 |

Table 3. Scientific visualizations and rule-counts

## 6. Conclusions

Our evaluation of Pavane, particularly the results obtained from the educational and scientific applications, leads us to make the following observations and conclusions:

- The declarative model and the rule-based form of the model used by Pavane are suited to the visualization of serial and concurrent algorithms.
- Pavane's rule-based notation allows the expression of complex visualizations in a compact form.
- Pavane's "learning curve" is quite short. In the edu-

cational applications, students were normally able to produce their first simple Pavane visualization within an hour, and after several more hours were making use of complex mappings and explanatory animations.

- The "compile-view-edit" cycle of Pavane is short enough (a few minutes) to allow an iterative approach to the development of visualizations. This encourages the exploration of alternate representations of data.
- This ability to easily explore alternate visualizations is appreciated by users and encourages the addition of (sometimes amusing) detail and explanatory animations to visualizations.
- Pavane is not suitable for the presentation of the very large data sets encountered in many scientific applications. However, the abilty to explore visualizations is still significantly useful in this context. In some cases this is because the user does not have a specific presentation of the data in mind; but more significantly, it can be used to search for presentations that bring out the key aspects of the data.
- Pavane and Swarm can be used together to visualize specifications. Coupled with the use of program derivation, this technique may someday provide a valuable tool for the development of large software systems.
- The analytical methodology is a valid means of developing visualizations that illustrate key properties of a computation's behavior. The methodology is of particular use in the concurrent domain, where operational depictions of programs often miss the truly fundamental properties of the algorithm.

Taken together, these results provide a very positive picture of Pavane and, more generally, of the declarative approach to visualization.

## Acknowledgments

The authors wish to thank all the participants in our study for their assistance. This work was supported in part by the National Science Foundation under Grants No. CCR-9015677 and CCR-9217751.

## References

[1] Brown, M. H., "Zeus: A System for Algorithm Animation and MultiView Editing," *Proceedings 1991 IEEE Workshop on Visual Languages*, pp. 4-9.

[2] Chandy, K. M. and Misra, J., *Parallel Program Design: A Foundation*, Addison-Wesley, New York, NY, 1988.

[3] Cox, K. C. and Roman, G.-C., "A Characterization of the Computational Power of Rule-based Visualization," *Journal of Visual Languages and Computing* 5(1), March 1994, pp. 5-27.

[4] Cunningham, W. and Beck, K., "A Diagram for Object-Oriented Programs," *SIGPLAN Notices* 21(11), 1986, pp. 361-367.

[5] Reiss, S. P., "Working in the Garden Environment for Conceptual Programming," *IEEE Software* 4(6), November 1987, pp. 16-27.

[6] Roman, G.-C. and Cox, K. C., "A Declarative Approach to Visualizing Concurrent Computations," *IEEE Computer* 22(10), October 1989, pp. 25-36.

[7] Roman, G.-C., Cox, K. C., Wilcox, C. D. and Plun, J. Y., "Pavane: a System for Declarative Visualization of Concurrent Computations," *Journal of Visual Languages and Computing* 3(2), June 1992, pp. 161-193.

[8] Roman, G.-C. and Cox, K. C., "A Taxonomy of Program Visualization Systems," *IEEE Computer* 26(12), December 1993, pp. 11-24.

[9] Roman, G.-C. and Cunningham, H. C., "Mixed Programming Metaphors in a Shared-Dataspace Model of Concurrency," *IEEE Transactions on Software Engineering* 16(12), December 1990, pp. 1363-1373.

## Note on the Plates

The images on the following page were produced using Pavane running on a Silicon Graphics Persional Iris. Although these static grayscale images cannot capture the full effect of Pavane's three-dimensional animated color imagery, we hope that they will provide some feel for the types of effects that can be easily created with just a few Pavane rules.

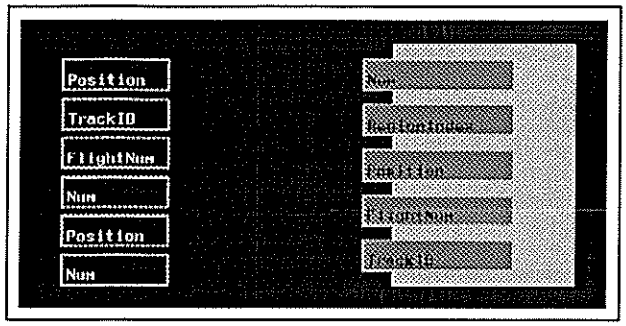


Plate 1. Package behavior. 6 Pavane rules.

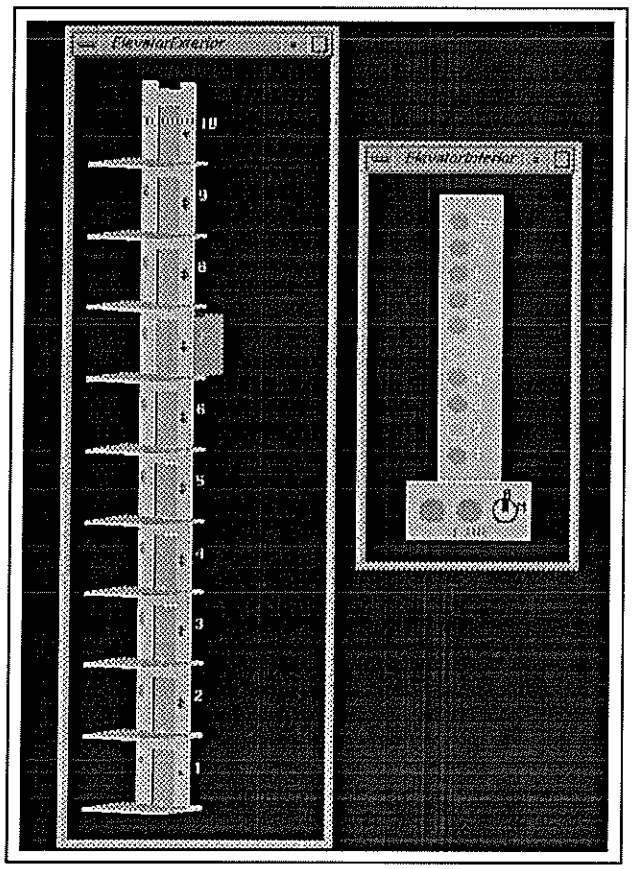

Plate 2. Elevator control system. 18 Pavane rules.

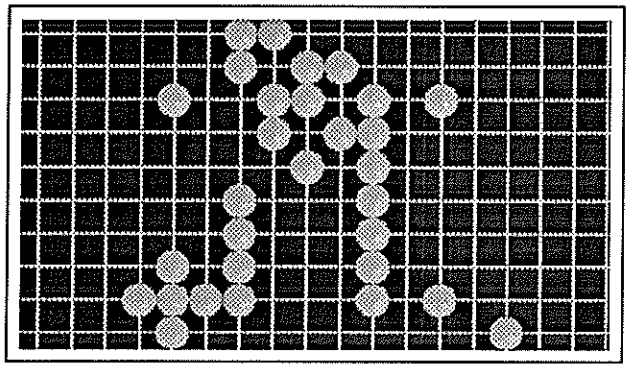

Plate 3. Game of Life visualization. 5 Pavane rules.

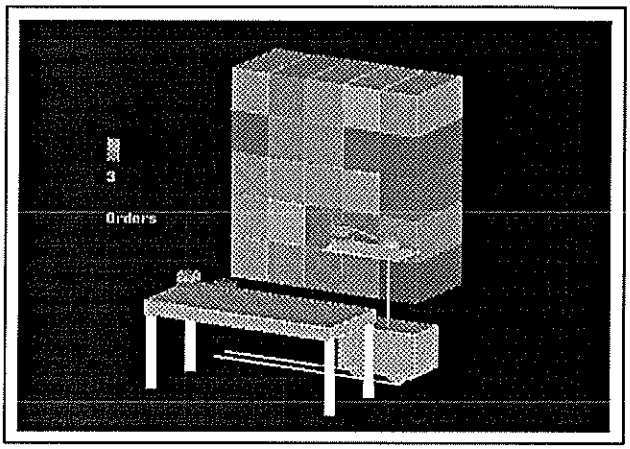


Plate 4. Order-filling system. 19 Pavane rules.

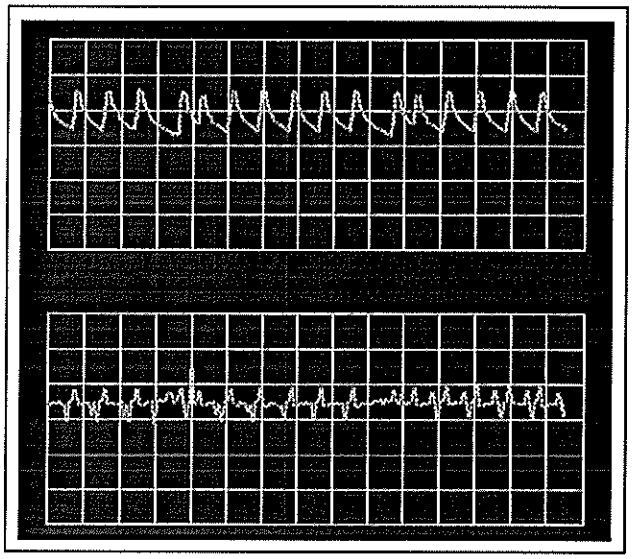

Plate 5. Standard medical display. 6 Pavane rules.

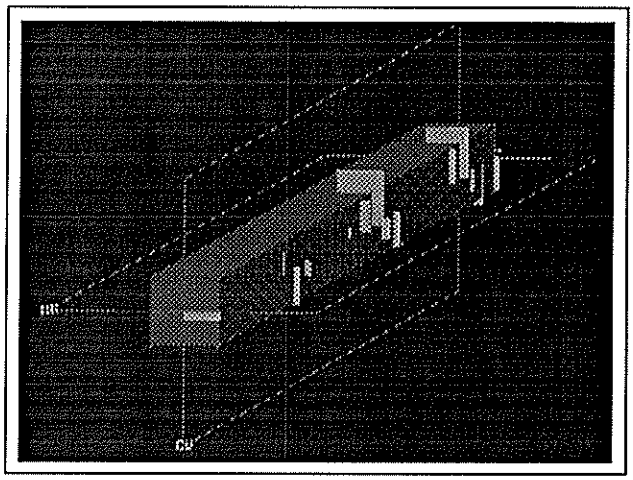

Plate 6. Novel medical display. 4 Pavane rules.