Visual Presentation of Software Specifications and Designs

Gruia-Catalin Roman
Delbert Hart
Charles Calkins

WUCS-94-08

Revised August 1994

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

Printed 11/18/94

Abstract

Formal methods hold the promise for high dependability in the design of critical software. However, software engineers who employ formal methods need to communicate their design decisions to users, customers, managers, and colleagues who may not be in a position to acquire a full understanding of the formal notation being used. Visualizations derived from formal specifications and designs must be able convey the required information precisely and reliably without the use of formal notation. This paper discusses a design methodology which attempts to integrate a design methodology based upon specification and program refinement with a state-of-the-art approach to rapid visualization of executing programs. The emphasis is placed on how to convey graphically various kinds of formally-stated program properties. The illustrations are extracted from a case study involving the formal derivation of a message router. The ultimate goal is to identify issues fundamental to the use of visualization in conjunction with formal methods and to catalog techniques which achieve effective visual communication without compromising formal reasoning.

Correspondence: All communications regarding this paper should be addressed to

Dr. Catalin Roman	office: (314) 935-6190
Department of Computer Science	secretary: (314) 935-6160
Washington University	fax: (314) 935-7302
Campus Box 1045	
One Brookings Drive	
Saint Louis, MO 63130-4899	roman@cs.wustl.edu

1. Introduction

As amply demonstrated by two recently published surveys [6, 12], program visualization is an area of significant research growth. Although to date its impact on software development methods has been relatively small, our own experiments identify requirements validation, rapid prototyping, training, monitoring, and testing as areas of immediate, high-payoff potential. It is less clear, however, what role program visualization ought to play during design. Since program visualization does not come for free, one must ensure that the benefits outweigh the potential costs associated with constructing visualizations. Predefined visualizations practically eliminate the construction costs but may not be responsive to the needs of a specific design effort. Rapid visualization techniques which allow one to develop custom visualizations with minimal effort offer an attractive alternative. This idea was the key motivating factor behind the development of Pavane [7] and the use of declarative visualization as its foundation. Experiments with Pavane showed that programmers with no prior visualization experience can construct relatively sophisticated program visualizations within half a day. Despite this, we believe that much more drastic reductions in the visualization development effort must be achieved before custom visualizations are likely to impact to a significant degree the typical design process.

There is, however, one design area where visualization may prove to be not only a cost-saving technique but a basic necessity. What we have in mind is the use of formal methods in the development of high-dependability software involved in critical applications. Formal methods involve specialized expertise not widely available. Yet, the designs must still be scrutinized by customers, users, managers, and colleagues who may not be in a position to become versed in the use of formal notation. Concrete visual representations of the evolving design could become a practical communication medium between designers and the rest of the community. This is not actually a new idea. Architects make use of scale models and elevation views to explain projects, gain approval, and to explore options. In this paper we investigate ways of incorporating a similar visual presentation strategy into a formal design process.

A particularly attractive formal technique is program derivation. It entails the application of a series of correctness-preserving transformations to some formal description of the problem to be solved. In sequential programming, where the approach enjoys a long standing and prestigious tradition, the starting point for the derivation is a pair of assertions. In concurrent programming the starting point may be either a program or a set of assertions about some computation. In one case, the program is gradually transformed into another program having certain more desirable properties, e.g., it can be implemented efficiently on a particular architecture; in the other case, the initial abstract specification is gradually refined up to a point when it is sufficiently concrete as to have a trivial encoding into some target programming language.

All the visualizations presented in this paper have been produced using Pavane. They involve one or more worlds (windows) of three-dimensional geometric objects, full-color, and smooth animation. Each visualization is constructed by writing a set of rules that map program states to graphical representations. Pavane supports rapid visualization of C and Swarm [8] programs. Swarm employs tuple-based communication à la Linda [2] and has a UNITY-style proof logic [3] which may be used to carry out specification refinements or prove Swarm programs correct. In addition, state changes in a Swarm program are automatically supplied to Pavane thus obviating any need to modify the program being visualized. These considerations make Swarm the language of choice for our work on formal design methods. The Swarm logic is used to build the specifications and carry out the refinements, the Swarm programming notation is used to write abstract programs that satisfy the specifications, and Pavane is used to construct the visualizations that convey the design decisions associated with each refinement step.

The specific case study presented in this paper involves a message router. In the simplest terms, the message router is a device which accepts messages on a number of input lines and delivers them to its output lines. Each message consists of a header, one or more body packets, and a tail. The header contains the packet destination. Packet ordering within a message is preserved, packets belonging to different messages are not interleaved, and messages from the same input line going to the same output line are not reordered. A full formal derivation of a message router design employing wormhole routing in a cross-bar switch [4] already existed prior to the start of this work. The general question we try to address in this paper is how can the various stages of the design be presented visually. We used the case study to identify challenging issues and helpful techniques.

The remainder of the paper is organized as follows. Section 2 provides an informal overview of our design methodology and the application domains where it was used. Section 3 reviews very briefly: the UNITY-logic to help the reader understand the basic nature of the formal specifications we employ; the Swarm notation used to construct abstract programs; and Pavane's rule based notation which facilitates the declarative specification of three-dimensional, full-color, smoothly animated graphical representations of executing programs. Section 4 outlines the router design (stage by stage) and uses as a backdrop against which to discuss a variety of useful visualization techniques. Brief concluding remarks follow in Section 5.

2. Design Methodology Overview

Central to our research is the idea that the lessons we have learned from exercises in program derivation can reshape the process by which we design concurrent systems. Critics of formal derivation make the claim that such techniques are tedious and costly and require highly-specialized skills and training. We contend that the tedium is not intrinsic to program derivation and that costs relate to the degree of reliability one attempts to achieve. We do not dispute, however, the fact that the application of formal techniques demands special skills. For this reason we are exploring a scenario involving designers comfortable with formal specifications who navigate expeditiously through the refinement process, providing some formal justification for each step along the way but not necessarily slowing down to carry out all the proofs. One important practical objective of our work is to understand better what combination of skills is essential to the successful application of program derivation on industrial-grade problems by a group of highly-trained specialists. To accomplish this we focused our attention on refinement methods and visual communication techniques. The former are the intellectual blueprint for the way in which design is approached and managed; the latter provide the means by which design decisions expressed in rigorous formal notation may be communicated to a broad audience and project progress may be monitored. These two issues and their interplay may not capture all that is needed to make industrial application of formal derivation possible but they are central such an undertaking.

Formal derivation. Program derivation is a technique for generating correct programs from some initial formal specification by means of stepwise refinement. The initial specification is usually highly abstract. Each refinement produces an increasingly more concrete and detailed specification with the final result being a compilable program. Successive specifications along the refinement path relate to each other in terms of some *implements* relation. Generally, specifications are expressed in a logic-based notation while programs are given in terms of a programming language. As many authors have noted, the distinction between the two is merely pragmatic. A program is also a specification, one which has an efficient implementation and exhibits a low level of abstraction. Because many of the “desired program characteristics” can not be expressed formally, program derivation is aided by criteria which are entirely outside the logic and are supplied by the designer as “informal motivation” for the individual refinement steps.

Previous approaches to formal derivation of concurrent programs can be classified into two broad categories. The first category includes methods which emphasize specification refinement with the program being generated at the very last step as a trivial coding exercise, e.g., [3]. The second category includes methods that start with an initial program which is subjected to a series of transformations which are guaranteed to preserve program correctness until a program having all the desired (informally stated) characteristics is obtained, e.g., [1]. In our experience, mixed specification and program refinement (Figure 1) offers some important practical advantages. Working on formal derivation of concurrent rule-based systems [10], we found it helpful to combine specification refinement and program refinement. Specification refinement was used to generate an initial program correct in all respects except that it was non-terminating. During program refinement we applied a series of optimizations which led to proper termination, to the elimination of operations too expensive for the target implementation, and to the elimination of busy-wait cycles. Similarly, during our investigation into architecture-driven refinement [11] we found it beneficial to generate initially a program which satisfies all functional requirements but fails to meet many of the architectural constraints. Subsequent program transformations lead to a program that satisfies both. In both areas we employed a UNITY-like specification refinement and generated Swarm programs which (when needed) were transformed mechanically to the target language.

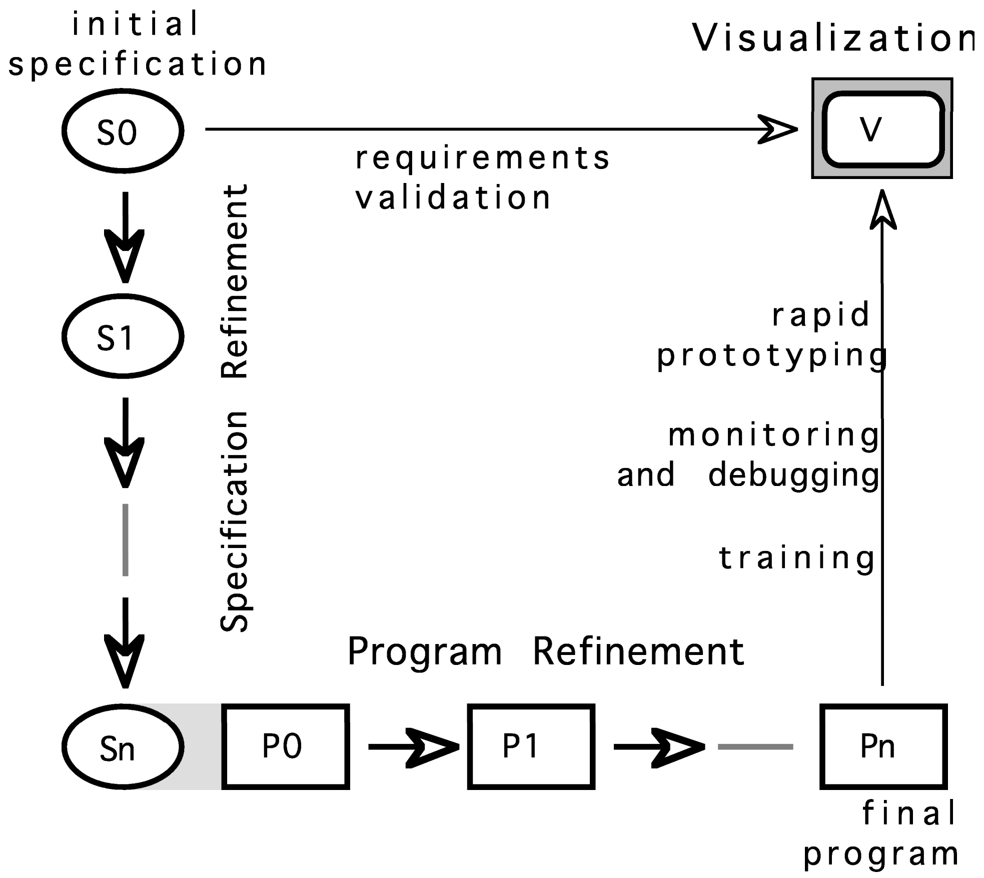


Figure 1. Design methodology overview and areas of immediate applicability for program visualization.

Visualization. Program visualization is defined as the graphical presentation, monitoring, and exploration of programs. While, in general, program visualization includes the graphical presentation of code, our interest in visualization is centered around abstract formal properties of programs. In large concurrent systems, code visualization is of limited value when trying to explain or explore the system's behavior. The complexity of the visualization overwhelms both the screen real estate and the cognitive capacity of the viewer. Furthermore, operational views of the system behavior convey the mechanics of the computation and not its rationale. The viewer may be able to observe every aspect of a program but still fail to understand its behavior. Consequently, our methodology relies on custom-built visualizations which capture formally stated assertions about the program being derived. To accomplish this, one needs access to rapid visualization capabilities and an innovative approach to visual representation of concurrent computations. The former is provided by Pavane [7, 12] while the latter is the research subject of this paper.

Pavane visualizations span multiple windows each containing a three-dimensional, full-color world of geometric objects. The scene being displayed is controlled directly by the state of the program being visualized. Each state change triggers a change of scene. The transitions from one scene to the next may involve highly synchronized and complex animations. The scenes capture abstract program properties. The animations are meant to enhance the aesthetic value of the presentation and to provide a visual commentary on the behavior of the program by focusing attention on specific objects or events or relations among them. In Pavane, the scenes and the animations are constructed by defining mappings from program states to sets of three dimensional objects from a predefined visual vocabulary, objects whose attributes can be a function of time (actually frame number). Each state transition is followed by the application of the mapping which, in turn, is followed by the display of the multiple frames required to accomplish a smooth transition from the previous to the new scene, i.e., an animation . The overall mapping is generally a composition of several simpler mappings. Each mapping is specified using a rule-based notation which is sufficiently powerful to allow for the computation of arbitrary history variables and fixpoints.

The typical program visualization process entails three steps. First, a program is constructed without any consideration being given to visualization. Since Pavane can visualize only programs, specifications are approached indirectly—by constructing a highly abstract program that satisfies the specification and visualizing the properties the program is supposed to satisfy. Next, some property of interest is selected for visualization. The property may emerge during attempts to formally verify the program or may be present in its specification. Other times, the designer may simply want to explore visually the validity of some hypothesis or understand some processing (even performance) pattern that might suggest a simpler formulation of some complex property. A tentative graphical representation is selected next and a Pavane mapping is constructed. After some experimentation and a preliminary evaluation of the suitability of the selected graphical representation, full animation is added.

Pavane's effectiveness as a rapid visualization tool has been evaluated in a variety of settings (Figure 1): scientific visualization, algorithm animation, rapid prototyping, requirements validation, program testing, debugging, and instruction. However, the question of how to integrate visualization into the design process is the most challenging task we have attempted to date.

3. Specification Methods and Notation

A distinctive feature of our strategy for integrating formal methods and visualization is the synergy among the program specification, programming notation, and visualization method. They all deal with global abstract views of concurrent programs and employ a tuple-based notation. Moreover, both the specification and the visualization methods are state-based and non-operational. The UNITY-logic involves assertions over the global state of the computation and the lowest level predicates look like tuples, predicate name and arguments. In Swarm, data and actions (called transactions) assume a tuple format and coexist in a global tuple-space call the dataspace. Declarative visualization maps program states, sets of Swarm tuples, to graphical objects which also assume a tuple format. In this section we provide a very brief overview of the three models and associated notations.

3.1. Program Properties

The basic program specification method we employ is due to Chandy and Misra [3] . It is a specialization of the temporal logic used to derive and verify UNITY programs—henceforth called UNITY logic or simply UNITY. As is the case with other logic-based models, the program behavior is given in terms of two classes of properties: safety and progress. Safety properties specify that certain actions (i.e., state transitions) are not possible, while progress properties specify that certain actions will eventually take place. In UNITY, the basic safety property is the **unless** relation. The formula p **unless** q states that if the program enters a state in which the predicate p is true and q is false, every program action will either preserve p or establish the predicate q. All other safety properties, such as **invariant**, **constant**, and **stable** are defined in terms of **unless**. The basic progress properties are **ensures** and **leads-to** (written $\mapsto$). The formula p **ensures** q states that p **unless** q holds and, in addition, there is some action which establishes q, an action the program is guaranteed to take in a bounded number of steps. Similarly, the formula $p \mapsto q$ states that if the program enters a state in which p is true, the program will eventually enter a state in which q holds, although p need not remain true until q becomes true.

For illustration purposes, let us consider a program in which data is moving from one register (say *i*)to the next (say *i+1*) in a finite length queue. Using UNITY, two aspects of the basic queue behavior can be stated simply as

$$\text{reg}(i,x)\ \textbf{unless}\ \text{reg}(i,\text{nil}) \wedge \text{reg}(i+1,x)$$

$$\text{reg}(i,x) \mapsto \text{reg}(i+1,x)$$

where *i* and *x* are universally quantified by convention; the first formula prevents the register *i* from receiving a new value before *x* is passed on to the next register in the queue and imposes an asynchronous movement of values by requiring a register to become empty (*nil* value) at the time its value is transferred to its successor; the second formula introduces a requirement that values must actually move along the queue.

UNITY specifications are simple and intuitive. There are few modeling artifacts introduced by the method, the state representation is direct, and the careful and creative use of auxiliary variables allows very sophisticated problems to have simple behavior specifications. Reasoning about global program states is more intuitive than reasoning about execution sequences. The UNITY logic may be used both as a specification language (during derivation) and as a proof logic (during verification). Although this paper avoids showing the formal specifications, familiarity with these concepts is necessary to appreciate the issues facing the designer who is attempting to confer graphical representations to such properties.

3.2. Programs

The only distinction between the UNITY and Swarm logics is in the way **unless** and **ensures** properties are proved using the program text. Since specification refinement involves neither writing nor proving programs, the target of the derivation may be either a UNITY or a Swarm program—UNITY is a proper subset of Swarm, subject to minor mechanical translation. Distinctions become important only when program refinements are involved. UNITY assumes that a program consists of a fixed finite set of variables and statements and that each statement is a conditional multiple assignment—thus the state is captured by the current values of the variables and an action corresponds to the execution of one of the statements. Swarm [8] and its logic [5, 9], has shown that the applicability of the UNITY logic can be extended to models which exhibit a very different set of characteristics: content-based access to data as in rule-based programming and Linda, dynamic creation of statements and data, and dynamic changes in the mode of execution (synchronous, asynchronous, and mixed).

Since the syntax of Swarm is very close to logical notation, many specifications have obvious representations as abstract Swarm programs, a fact which enables us to execute specifications with minimal effort. To illustrate this, we present the definition of a simple Swarm transaction *Move(i)* which forwards data along the queue discussed earlier. In Swarm, this requires one to create a parameterized transaction class called *Move* (shown below) and to place all the needed transaction instances in the dataspace (at initialization or dynamically):

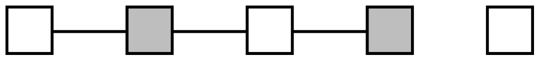
```
Move(i) ≡
          x : reg(i,x) ∧ reg(i+1,nil)
          →  reg(i,x)≤, reg(i+1,nil)≤, reg(i,nil), reg(i+1,x)
      ||  true
          →  Move(i)
```

According to the definition above, a transaction *Move(i)* consists of two subtransactions that are executed in parallel. The first one, involves a query which extracts the current value *x* for register *i* and checks if the register *i+1* is empty. If the query is successful the associated action part (following "→") is executed by first deleting from the dataspace the tuples holding the old register values (marked by a dagger "≤") and inserting tuples representing the new register states. If the query fails the first subtransaction has no effect. The query of the second subtransaction (following "||") always succeeds and recreates the transaction instance which otherwise would be implicitly deleted from the dataspace as soon as it is executed. Transactions are selected fairly and parallel execution (which extends to groups of transactions) follows a basic three-phase pattern: query evaluations precede data-tuple deletions which, in turn, precede data-tuple and transaction insertions into the dataspace.

3.3. Visualizations

In Pavane, visualizations are implemented as mappings from the state of the program to a collection of graphical objects in a four-dimensional space — the three spatial dimensions plus time. For notational convenience, we express both the state of the underlying computation and the final set of four-dimensional graphical objects as collections of tuples, called respectively the *state space* and *animation space* of the visualization. The overall mapping from state space to animation space can be decomposed into a pipeline of any number of sub-mappings, with each intermediate mapping transforming one space into the next in the pipeline; each of these intermediate spaces is also a collection of tuples. Although the Pavane rules look very similar to Swarm transactions, the visualization semantics is radically different from the transaction execution semantics. Pavane assumes that the underlying computation progresses by means of a series of atomic transitions which modify the state. After each transition, the visualization rules are re-applied to the new state and the resulting animation space is rendered.

To construct a very simple visualization of a program implementing the queue, we may want to represent an empty register as an unfilled square, a register holding some value as a filled square, and a *Move* transaction as a line. The picture below shows three empty registers and reveals an initialization error—a missing transaction.



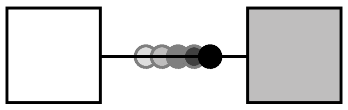
Only two trivial Pavane rules are required to construct this kind of picture

DrawRegister ≡
  i, x : reg(i,x)
  ⇒ _cube(_center:=[3*i,0,0], _size:=1, _fill:=(x≠nil))

DrawPotentialFlow ≡
  i : Move(i)
  ⇒ _line(_from:=[3*i,0,0], _to:=[3*(i+1),0,0], _chop:=0.5)

The result of a rule application is the union of the results of every successful instantiation of each rule.

Adding animation is less trivial. The increase in the complexity of the rules is determined by the sophistication of the animation being constructed. For this example, a simple animation may involve reemphasizing the flow of data by showing a little ball that retraces the data movement to attract attention to a value which just changed registers instantaneously.



Only one rule would be required in this case

DrawActualFlow ≡
  i, x : old.reg(i,x) ∧ x≠nil ∧ reg(i,nil) ∧ old.reg(i+1,nil) ∧ reg(i+1,x)
  ⇒ _sphere(_center:=ramp(0,[3*i,0,0],10,[3*(i+1),0,0]), _size:=0.2, _color:=[200,0,0])

This rule is triggered whenever the register *i* is empty but was observed to be non-empty in the preceding state. A sphere whose center moves from one register to the next over ten video frames is generated. Note that for a correct program the query part of the Pavane rule is overspecified. It was designed, however, to ensure the detection of possible program errors—if a register is emptied under the wrong conditions no sphere is displayed!

As the rules above already suggest, the overall process of visualization is data-driven; the rendering computation must wait for data from the visualization computation, which in turn waits for data from the underlying computation. It is not, however, synchronous. The underlying computation is permitted to send state changes more rapidly than the visualization computation is able to process them, and similarly the visualization can send animation spaces faster than the renderer can translate them into images. The latter often happens, simply because the process of generating an image (and, more importantly, the comprehension of that image by the viewer) is much slower than the other two computations. If synchronous execution is desired, it can be requested when the computation is started; the underlying computation is then forced to wait for the rendering computation to complete the display of an animation space. This is particularly useful when contrasting two or more different visualizations by displaying them in separate windows, since synchronization guarantees that all the images represent the same state of the underlying computation.

Pavane's graphical model provides one or more three-dimensional "worlds", each containing a collection of graphical objects. The animator defines one "window" for each world; the window definition includes the world's properties (center, scaling, background color, and so forth), the properties of the screen window which will be opened (dimensions, position, etc.), and the types of transformations that the viewer is permitted to make. Subject to any such limitations that the animator requires, the viewer can examine each world "through" its window from any point in the world's coordinate system. The ability to use multiple windows is especially convenient when the effectiveness of two or more visualizations is being compared or when multiple properties of the same program need to be examined together.

Pavane has undergone many repeated developments and redesigns. It consists of a compiler for visualization rules, a display subsystem, and a number of run-time libraries for monitoring C and Swarm programs, for executing visualization rules, and for supporting the display process. The most recent version is written in C++ under UNIX and requires X Window System® support. The only component which is vendor dependent is the display subsystem which makes use of the Silicon Graphics® Graphics Library™ (GL).

4. Visualization Techniques

The discussion in this section is informal by intent as it is meant to be equally accessible to formalists and practitioners. No UNITY assertions, Swarm programs, or Pavane rules are shown. The initial specification of the router and the key refinements leading to the final design are presented in a summary form and used primarily as a backdrop against which we organize a discussion of techniques which we found helpful in the visualization of the router and of other programs as well. Special emphasis is placed on identifying general graphical representation principles and strategies that transcend the specifics of this particular case study. We also identify several problems for which we did not find a satisfactory solution yet. Furthermore, even though the full derivation of the router predates this case study, we went to a great effort to make sure that in visualizing each refinement we used only information available at that specific point in the design process.

4.1. Initial specification: A message router

We consider a communication network that connects *N* senders of messages to *M* receivers via a message router. Each sender is connected to one of the input ports of the router, and each receiver to one of the output ports. Each message is composed of a finite number of packets that can be of three different types: *header*, *body*, and *tail*. The header, which is the first packet of the message, contains the port address of the message destination. Each header is followed by one or more body packets which contain the actual data. Finally, the tail packet marks the end of the message. The externally observable behavior of the router is defined by the following requirements:

(R1) The value of the body packets must not be modified, but, for control purposes, the router may modify the value of the header and tail packets.

(R2) Packet ordering within a message must be preserved (at the receiver).

(R3) Messages from the same source going to the same destination must not be reordered (at the receiver).

(R4) Messages from different sources going to the same destination must not be interleaved (at the receiver).

(R5) Each packet that is sent must eventually be delivered to the intended receiver.

These five properties are ultimately captured by eleven formal assertions and a number of assorted definitions. Since Pavane can only visualize programs, the first step towards building a visualization is to generate a program *P* which embodies the initial specification *S*. This needed intermediary step raises issues regarding the effort and accuracy associated with the approach. The cost of developing the program is minimal because Swarm is a very high level language allowing arbitrary queries over the state space expressed in a notation similar to that used to write logical assertions. Accuracy is a more complicated issue. It is easy to guarantee that the program is correct but it is more difficult to make sure that all behaviors permitted by the specification are actually captured by the program. This can be a serious drawback if the program is used to explore the specifications but it is less of a problem when used to communicate ideas the designer explicitly decided to present. (Possible mechanical solutions for the former case are: to make the specifications executable or to work with program refinement.)

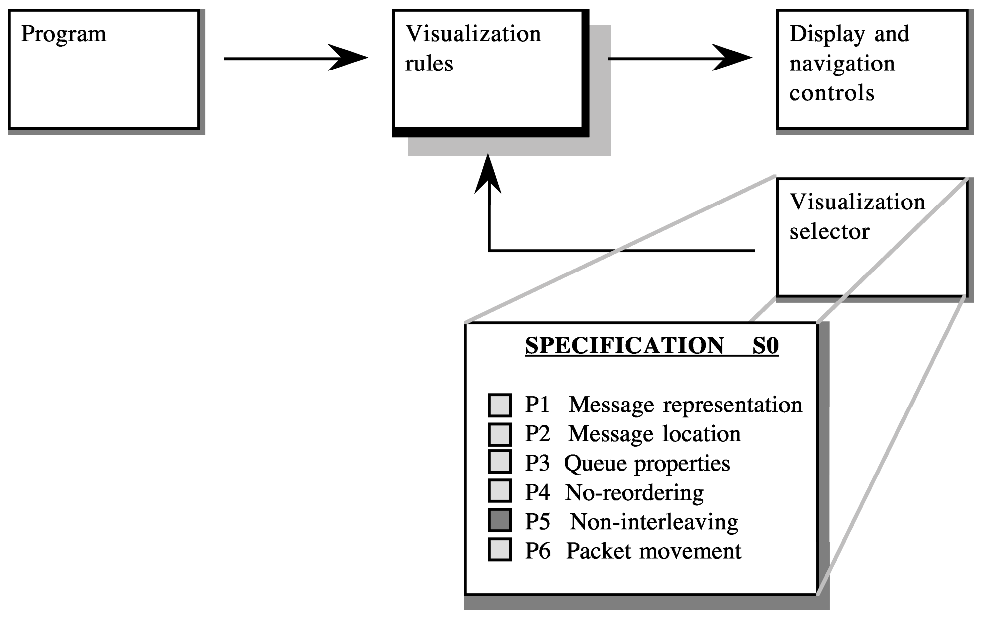


Figure 2. Basic visualization structure and controls.

Once the program is written one needs to decide on the desired visual representation and write Pavane rules that map the program state to the graphical objects on the screen (Figure 2). Smooth animation and other special effects are added to the rules once the basic representation is considered satisfactory. Although not a necessity, we find it convenient to allow the viewer to select from among a set of available visualizations during program execution—a rule selector window is constructed providing a list of visualizations, annotations explaining how they relate to the formal specifications, and buttons to turn them on and off. One visualization selector and set of rules was constructed for each specification generated during the router design. The visualization selectors listed all the relevant formal properties and provided buttons that enabled the visualization of one desired property at a time. In addition, Pavane provides a standard set of 3D navigation controls.

This section, however, is not about the mechanics of visualization but about representation. We turn our attention next to the concepts we need to visualize and the methods that help us convey them graphically. To help the reader recall and refer to specific techniques we gave them names. Very few of these names enjoy general acceptance in the field now but, as the area matures, a standard terminology must eventually emerge and we hope that some of our terms will be part of it.

Steel gray. From the very start we realized that a single visualization able to capture all the important formal properties of the router is neither feasible nor desirable. Nevertheless, all the visualizations must contribute to a single unified mental picture and must relate easily to each other, especially when the visualization changes midstream. The solution was to create one single common reference point shared by all the visualizations, a mostly neutral image that provides a contextual framework on which to highlight other properties. In the case of the router the input and output queues readily suggest a simple rectangular structure for the router with the input queues on the left to imply a starting origin and the output queues at the top to hint at a sense of progress (Image 4.1a). We use shades of gray to allow the image to recede in the background and to lower the viewer's expectation in terms of the amount of information being communicated.

Thin air. In the steel gray image the messages look like nuts on a rod. What should happen when a message travels from input to output? One subtle point about the initial specification is that it defines only the behavior observable in the environment of the router and says nothing about how the router should be realized. Showing the router as a black box is not reasonable because the specification states that each packet is present some place at all times. Our approach is to show the presence of packets in the router while making clear that their behavior and existence there does not have the same reality as on the inputs and outputs. The router (Image 4.1a) exhibits no internal structure while packets float in thin air and bounce off the walls like elastic balls. The result is not suggestive of any design while the continued existence of packets is evident.

Spot color. Some of the formal properties are concerned with the structure of the messages and others with the ordering of packets. Because these properties apply to all the messages, there is a tendency to try to color code each message. This may create a pleasing image but it fails to focus the viewer's attention on anything in particular and provides more information than necessary. Such images require additional (unnecessary) oral explanations whose sole purpose is to force the viewer to concentrate on one particular aspect of the image, i.e., to ignore the rest. Take, for instance, the property that packets belonging to different messages are not interleaved (on output). One can think of this property as a relation among all the messages but it is much more profitable to view it as a relation between one particular message and all the rest. This can be easily captured by assigning a color to one of the messages and leaving all the others gray. For reasons having to do with keeping the number of visualizations small, we opted to use spot colors also for the source and destination queues associated with the colorized message and to differentiate the header and the tail packets. Only when we want to show that ordering is preserved among messages that have the same source and destination we spot color a pair of messages (Image 4.1b).

Event compression. Most models of concurrency assume that atomic actions associated with a concurrent computation are executed serially although an actual system may schedule them in parallel if the resources are available and the observable effect is the same as some serial execution. The abstract program associated with the initial router specification, for instance, treats the movement of individual packets as separate atomic actions. The specification, however, does allow for several packets to move simultaneously. To actually capture this behavior in the program entails additional coding and performance costs. We found that one simple way to get around this added burden is to allow the program to take multiple steps which are visually captured as a single combined synchronous transition. One must exercise care not to create visual transitions that are not permitted by the specification. If individual atomic steps affect distinct graphical objects appearing in the visualization, it is relatively easy to combine them into a single visual transition thus offering the viewer the illusion of parallel or synchronous execution. It is possible, for instance, to advance by one position synchronously a whole set of adjacent packets located on the same input queue. The approach would not work, however, if the packet closest to the router advances two positions while the rest advance only one. In our case study the rule selector allows the viewer to choose between a serial execution and one involving combined presentation of multiple atomic actions.

Free walk. The last issue we faced in visualizing the initial specification was how to handle the nondeterministic behavior implicit in the specification. One option is to introduce randomness in the selection of the program statements. A second option is to involve the viewer in the choice of action to be executed next. We opted for the latter because it promotes active exploration of the program's behavior and also because Pavane already supports this kind of viewer control over the execution of Swarm programs (but not for C programs which are sequential).

These techniques were used extensively throughout the remainder of the case study and, as we continue looking at successive design refinements, we will try to showcase only those presentation methods that have not be discussed already in a previous section.

4.2. Refinement 1: Router topology

The first refinement defines the general topology of the router as a grid of *N*×*M* switches. The *N* input lines and *M* output lines are thus extended inside the router. Each switch can receive packets from its left neighbor on the row or its bottom neighbor on the column, and can route them either to its right neighbor on the row or to its upper neighbor on the column, depending on the destination of the packets. So, to move from its source row to its destination column, each packet first travels along the row (one switch at a time) until it reaches the destination column, and then just moves up the column (also one switch at a time). Each row-column intersection has a switch that holds two registers and some yet to be specified control logic.

Continuity of structure. Visually, the steel gray representation is extended into the router. The thin air behavior is replaced by packet movement from one register to the next. A packet travels along the row up to the point when the destination column is reached. Then it moves from the current row register to column register on the row above and continues to travel along the column registers. It is important to exploit the viewer's familiarity with the earlier view of the router and its environment and extend the representation to include the new structural elements. This strategy can speed up understanding and reduce the amount of verbal explanations required to interpret the imagery. Since we try to reuse the context in successive refinements, this new view can be looked at in conjunction with the next refinement (Image 4.3a).

Scenario recycling. The essence of the first refinement is the extension of properties true outside the router to the router itself. Most formal assertions associated with Refinement 1 actually have forms identical to those of their counterparts in the initial specification. Consequently, it is not surprising that the presentation scenarios introduced earlier can be used again with the refined structure. This simplifies understanding and saves some of the effort involved in the development of visualization rules. While the program being visualized changed, many of the visualization rules can be reused.

4.3. Refinement 2: Arbitration logic

The second refinement provides additional details about the behavior of each single switch, by defining the mechanism that prevents messages from being interleaved along the columns. This can be done by associating two mutually exclusive signals—*turn* and *up*—with each switch. Signal *turn* prevents messages from moving through the switch along the column when a message is currently passing through the switch from the row to the column, and the other way around for the signal *up*.

Choreography. The value of the signals is determined by the movement of the header and tail packets through each switch. This coupling between packet movement and signal changes can acquire a visual counterpart by carefully choreographing the timing of the corresponding changes in the image. A header reaching the destination column is involved in three transitions: arrival in the row register, the setting of the *turn* signal to true, and the transfer to the column register on the row above. These transitions are sequenced in this manner by the program and naturally appear as such in the animation. The tail packet, however, is required to reset the corresponding signal (*turn* or *up*) upon exiting the switch. Let's assume that the switch state is shown in terms of the paths currently open through it. Changing the switch configuration while the packet is moving or immediately after it stopped moving can be misleading by making the packet travel on an apparently non-existent line or by suggesting an ordering that does not exist. Our solution takes advantage of Pavane's fine grained synchronization capabilities by scheduling the change in the switch configuration to start at the time the tail packet is almost entering the next switch and to end at the same time as the packet movement.

Tracing. Many of the formal assertions associated with this refinement are invariants that relate the state of the switches to the distribution of packets across the router—along the path laying between the header and the tail all the switches must be properly set. We show this by simply coloring the path taken by the header of one particular message and by having the tail erase the color (Image 4.3a). This one single visualization covers a number of assertions each dealing with a distinct header/tail location pattern. The location patterns map visually to shapes of the color trace.

Emaciation. Throughout the case study we tried to keep visual representations simple through the frugal use of color. One other approach to simplification is to create more abstract representations. This is highly recommended when the viewer wants to see global patterns or relationships involving multiple objects. Such properties tend to correspond to formal assertions that are not part of the specification but can be derived from it. As an example, one may want to see the message traffic through the router. To do this we can reduce the router to a simple grid, represent messages by their headers (balls whose color indicates the destination column), and turn on the tracing mechanism for all messages in the router (Image 4.3b). The result is a trade-off between level of detail and volume of information.

4.4. Refinement 3: Fairness constraint

In the third refinement, we specify further the behavior of the switches by introducing a strong fairness constraint, i.e., the existence of a constant upper bound on the number of messages that can block a particular message from passing through each switch. We choose a design in which a message waits for at most one other message to pass through the switch before it can proceed.

Uncloaking. To render the fair behavior we need to highlight to the user the conditions under which a message may be blocked and than show that this does not happen. Colorizing an arbitrary message is not useful here. The particular message may not enter in a situation where indefinite blocking occurs while other messages may be blocked without the viewer realizing it. The solution we adopted is to colorize all packets of a message whose header is blocked at a switch and remove the colorization as soon as packet moves on towards its destination. We also highlight the blocking message to show that the blocking message is making progress.

4.5. Refinement 4: Destination address

At this point in the design, each switch on the rows makes the decision to route messages either to the next switch on the row, or to the next switch on the column, by comparing the message destination to the number of the column it is located at. This implies that each switch has to know its location. The purpose of the fourth refinement is to eliminate this knowledge by using the value of the header packets. We make the value of each header packet decrease by one each time the packet passes through a switch along the row. Since the value is initially equal to the destination column, this implies that a message will have to take a turn when the value is equal to 1.

Foretelling. Formally, the changes in the header value are tied to its current position on the row by an invariant relation. Given the current position and the header value we can foretell where the header will eventually move to the destination column. This allows us to materialize the header value as a beam of light originating in the header packet and focused on the switch where the turning on the column will occur (Image 4.5a). As the packet gets closer to the destination column, the focal distance of the beam is reduced until it goes down to zero. This kind of visual representation plays a key role in the visualization of two important classes of assertions. First, many progress properties (e.g., "*eventually the header reaches the destination column*") are proved using a variant function over a well-founded set (e.g., the remaining distance to the destination column). By showing a graphical representation of the metric one can actually see how progress is being measured. Second, **unless** properties (by their very nature) constrain the set of possible state transitions. In complex situations, it becomes helpful to depict the set of possible transitions by showing both some aspect of the current state such as the current packet position and the positions the packet can occupy next. Of course, when there are not many choices, as is the case in a cross-bar router, this use of foretelling is not necessary.

4.6. Refinement 5: Asynchronous movement

The last refinement deals with the execution control we impose on the switches. A possible choice is to have the switches running asynchronously, another choice is to have them working in a synchronous way. We chose the more realistic asynchronous behavior. Since each location can contain at most one packet, this implies that a packet is not able to move to the next location, unless it is empty. This refinement is constructed simply by adding a single unless property that requires a packet to leave an empty register behind when it departs the switch. Trying to visualize that certain behaviors are no longer permitted is a little tricky. A simple solution is to foretell the situations in which packet movement is feasible by highlighting registers which could receive a packet in the next transition.

Magnification. Because at this point it is possible to write the final program, it is conceivable that one might want to present a visualization that includes details such as dataflows or even wires connecting the registers. We opted to combine one of the previous visualizations with a detailed visualization of the processing logic for a switch and its connections to its four neighbors (Image 4.6a). The absence of animation makes understanding the image more difficult. We present an annotated description of the switch representation (Figure 3) to clarify the image. The former visualization provides the global context while the latter shows the internal workings of a small portion of the switch. The two visualizations appear in separate windows exhibiting fully synchronized behaviors. The choice of switch to be shown in a close-up is at the viewer's discretion, i.e., interactive.

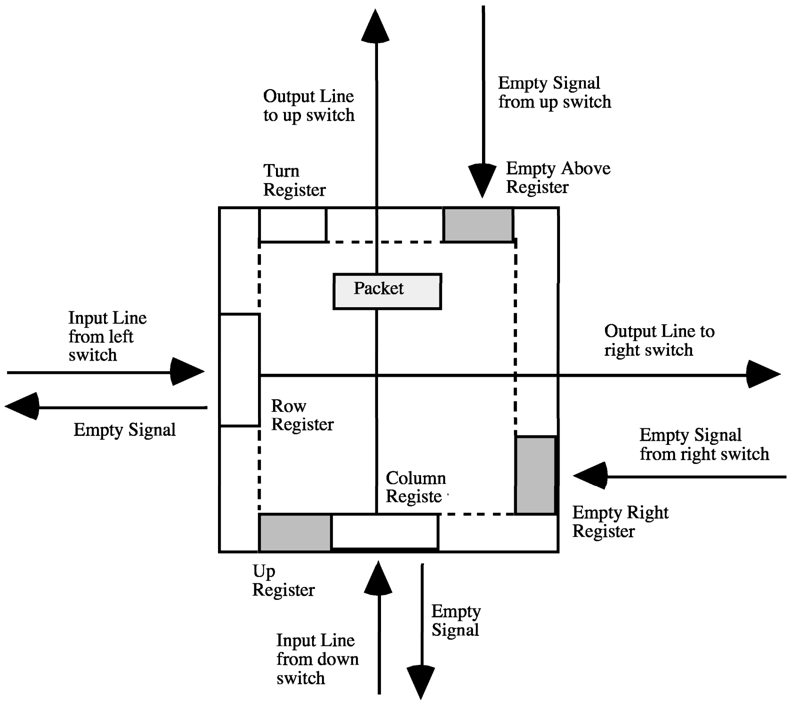


Figure 3. Annotated description of switch level visualization. (magnification).

5. Conclusions

In some earlier work [7] we proposed a *proof-based* methodology for the visualization of concurrent computations. We sought to find correspondences between formal properties of concurrent programs and appropriate visual counterparts. The case study described in this paper builds upon those early ideas but shifts the emphasis from after-the-fact program presentation to the integration of visualization into the design process. The paper showed the kinds of representations that emerge during a rigorous program design process and considered their interactions along a sequence of refinements. This type of investigation represents an important first step towards understanding how visualization can support the application of formal design methods. The case study revealed many useful elements of a systematic visualization methodology. They, in turn, contribute to defining the kinds of features that ought to be included in the next generation of program visualization systems.

6. References

[1] Back, R. J. R., and Sere, K., "Stepwise Refinement of Parallel Algorithms," *Science of Computer Programming*, vol. 13, no. 2-3, pp. 133-180, 1990.

[2] Carriero, N., and Gelernter, D., "Linda in Context," *Communications of the ACM*, vol. 32, no. 4, pp. 444-458, 1989.

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

[4] Creveuil, C., and Roman, G.-C., "Formal Specification and Design of a Message Router," Department of Computer Science, Washington University, St. Louis, Missouri, Technical Report WUSC-92-44 (To appear in ACM Transactions on Software Engineering and Methodology), 1992.

[5] Cunningham, H. C., and Roman, G.-C., "A UNITY-Style Programming Logic for a Shared Dataspace Language," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 3, pp. 365-376, 1990.

[6] Price, B. A., Baecker, R. M., and Small, I. S., "A Principled Taxonomy of Software Visualization," *Journal of Visual Languages and Computing*, vol. 4, pp. 211-266, 1993.

[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 Computating*, vol. 3, no. 1, pp. 161-193, 1992.

[8] Roman, G.-C., and Cunningham, H. C., "Mixed Programming Metaphors in a Shared Dataspace Model of Concurrency," *IEEE Transactions on Software Engineering*, vol. 16, no. 12, pp. 1361-1373, 1990.

[9] Roman, G.-C., and Cunningham, H. C., "Reasoning about Synchronic Groups," in *Research Directions in High-Level Parallel Programming Languages*, J. P. Banâtre, D. L. Métayer, Eds., Springer-Verlag, New York, NY, vol. 574, pp. 21-38, 1992.

[10] Roman, G.-C., Gamble, R. F., and Ball, W. E., "Formal Derivation of Rule-Based Programs," *IEEE Transactions on Software Engineering*, vol. 19, no. 3, pp. 227-296, 1993.

[11] Roman, G.-C., and Wilcox, C. D., "Architecture-Directed Refinement," *IEEE Transactions on Software Engineering*, vol. 20, no. 4, pp. 239-258, 1994.

[12] Roman, G. C., and Cox, K., "A Taxonomy of Program Visualization Systems," *IEEE Computer*, vol. 26, no. 12, pp. 11-24, 1993.