Washington University in St. Louis
Washington University Open Scholarship

All Computer Science and Engineering Research
Computer Science and Engineering

Report Number: WUCS-89-44

1989-10-01

A Declarative Approach to Visualizing Concurrent Computations

Gruia-Catalin Roman and Kenneth C. Cox

Follow this and additional works at: https://openscholarship.wustl.edu/cse_research

Recommended Citation
Roman, Gruia-Catalin and Cox, Kenneth C., "A Declarative Approach to Visualizing Concurrent Computations" Report Number: WUCS-89-44 (1989). *All Computer Science and Engineering Research.*
https://openscholarship.wustl.edu/cse_research/754

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160.

A Declarative Approach to Visualizing Concurrent Computations

Gruia-Catalin Roman and Kenneth C. Cox

WUCS-89-44

October 1989

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

A Declarative Approach to Visualizing Concurrent Computations

Gruia-Catalin Roman and Kenneth C. Cox
Washington University in St. Louis

Program verification promises to provide a formal foundation for visualizing concurrent computations, a technical endeavor currently dominated by empiricism and aesthetics.

The importance of visualization as a communication tool has long been acknowledged. Recently, however, a growing consensus has emerged regarding its potential for promoting the understanding of complex behaviors exhibited by physical phenomena and computations. We explore visualization — graphically representing objects and processes — as a means for understanding programs consisting of large numbers of concurrent processes. Indirectly, we hope to establish a new technical foundation for research into the monitoring and debugging of large-scale concurrent programs.

The extremely high volume of information produced during the execution of a concurrent program greatly exceeds human abilities to assimilate it in textual form. This is in part due to the sequential processing of textual information. The human visual system is more suited to processing information in the form of images. Humans can process large quantities of image information in parallel, detecting and tracking complex visual patterns with incredible speed. Nevertheless, as the number of processes grows, the viewer's ability to understand the resulting image can be rapidly saturated unless the displayed information's level of abstraction is increased. For this reason, abstraction plays an important role in visualization. By providing flexible abstractions, a visualization system can help the programmer select displays that are easily specified and understood.

We doubt, however, that powerful abstractions of concurrent computations can be built using the operational methods currently employed in program animation. Although helpful in understanding the behavior of sequential programs, operational reasoning fails when faced with a multitude of concurrent actions that may be interleaved in arbitrary ways. These conditions require a greater reliance on formal verification. We will explore a methodology that attempts to derive visual representations of concurrent computations from their proofs. Correctness proofs involve either reasoning about computational sequences (for example, CSP[1]) or about program states (for example, Unity[2]). We consider the latter approach to be better suited as a foundation for visualization because program states map more readily to images. Moreover, we expect to be able to render invariant properties of the program state as stable visual patterns and to render progress properties as evolving visual patterns.

The departure from operational reasoning also has important ramifications for the ease with which visualizations can be

0018-9162/89/1000-0025$01.00 © 1989 IEEE

specified. Instead of treating visualization as a mechanism invoked whenever changes in the display state are desired, visualization may be defined as an abstraction, that is, a mapping from computational states to the states of graphical objects rendered by a display device. We will distinguish between the two approaches by characterizing the former as imperative and the latter as declarative. As currently employed, the imperative approach involves actual modifications to the program code: Procedure calls inserted at appropriate points in the code trigger desired changes in the image being displayed. The declarative approach, however, promises to eliminate the need to alter program code and to accommodate runtime changes in the abstraction level of the information being displayed.

Although the declarative approach can be used with any concurrent language, we have selected the shared dataspace paradigm as the underlying model of concurrent computation. Shared dataspace languages communicate among concurrent processes via a common content-addressable data structure (typically a set of tuples) called a dataspace. Content-based addressing of data is common in artificial intelligence, and expert system languages such as OPS5[3] can be classified as shared dataspace languages. Linda[4] became the first concurrent shared-dataspace language to be implemented and to receive commercial attention. Swarm,[5,6] a language developed and used by our research group as a vehicle for studying programming and visualization methodologies, became the first concurrent shared-dataspace language to have an assertional-style proof logic. Swarm is particularly appealing for declarative visualization because its dataspace fully specifies the current program state (control and data) and has a simple uniform-state representation (a set of tuples).

Although we use Swarm as the basis for our discussion, this article is not about a particular language or visualization system. (Actually, since efforts to implement Swarm and its visualization testbed on a hypercube-class multiprocessor are still in their initial phases, all images in this article are from simulated executions of the various programs.) Rather, we focus on our concept of the future of program visualization. We present arguments in favor of the declarative visualization paradigm and build a case for program verification as the technical foundation for a formal approach to visualization.

Declarative visualization

The visualization field can be divided into three broad areas. Visualization in scientific computing, or ViSC, refers to the animation of data such as that produced by supercomputer simulations, satellites, and measuring devices used in astronomy, meteorology, geology, and medicine. Visual programming is the specification of programs in a notation using two or more dimensions, as by flowcharts, graphs, diagrams, or icons. Program visualization, also called algorithm animation, uses images to represent some aspect of a program's execution.

Program visualization research has been motivated by the desire to explain, by means of animated displays, the workings of sequential algorithms. The Balsa[7] system was designed with this goal in mind. Balsa was one of the earliest visualization systems and is still probably the best known and the most influential. Balsa uses an animator to construct visualizations. The animator determines which events in program execution should be captured and how they will be represented. The animator then augments the algorithm with calls to library procedures that interface with the mechanics of display generation. The procedure calls, which are embedded in the existing algorithm code at points where the key events occur, trigger changes to the display.

Balsa uses an imperative approach to algorithm animation. Image generation is, in essence, treated as a side-effect of program execution: Specific events modify the image in particular ways. This approach, while quite successful, has the inherent drawback that the animator must modify the program code. It is also generally difficult to change the information being displayed and the way in which it is displayed. Although many systems allow the viewer to select from several abstractions provided by the animator, they do not permit the viewer to specify an entirely new abstraction. Generally, this would require identifying new events and marking them in the code.

Possibly in response to these difficulties, a recent trend is to use declarative methods of algorithm visualization. In several systems, the algorithm animator declares a number of graphical objects — usually considered to be icons — with parameters that can be changed by program operation. The Aladdin system[8] uses this approach. However, as in imperative systems, the Aladdin animator must still modify the program code to update objects.

Several other systems, by binding object parameters to program variables, manage to remove the animator's need to modify the program code. Changes to the variables are transmitted to the visualization system, which changes the icons and updates the display. The Provide system,[9] intended for use in debugging, considers all potentially interesting aspects of algorithm behavior. Therefore, procedure calls inserted by the compiler automatically record all state changes. The PVS system,[10] intended for the monitoring of manufacturing processes, assumes all information of interest is stored in a database accessible to the visualization system. This approach has certain similarities to our own shared-dataspace model of concurrency.

Shared dataspace

Before presenting the visualization methodology, we'll introduce some shared-dataspace concepts and notation by means of a simple, nondeterministic, parallel algorithm. We first express it in a traditional block-structured notation and then restate it as a shared-dataspace program, using notations borrowed from the Swarm language. The algorithm can be specified informally as follows:

> Given an array $X[1..N]$ of strictly positive integers, compute the sum of all the values stored in X and place the result in one of the array entries; the other array entries should be zeroed.

This algorithm's first implementation is given in a hypothetical block-structured language that provides a cobegin-coend construct, atomic predicate evaluation, and conditional atomic multiple-assignment statements.*

For each entry $X[k]$ in the array we create a concurrent branch of a cobegin-coend construct. The branch corresponding to a nonzero array entry $X[k]$ attempts to accumulate into $X[k]$ the values of array entries having an index higher than k while simultaneously zeroing such entries.

*For example, the statement **if** a>0 **then** a,b := 2,5; **endif** is logically equivalent to locking the variables *a* and *b*, executing the statement **if** a>0 **then** a := 2; b := 5; **endif**, and then unlocking the variables.

```
N : positive natural;
X[1..N] : array of positive natural;

cobegin for each k:=1..N–1;
  for j:=(k+1)..N loop
    if X[k] = 0 then exit; endif
    if X[k] ≠ 0 and X[j] ≠ 0 then
      X[k], X[j] := (X[k] + X[j]), 0;
    endif
  end loop;
coend
```

The execution is nondeterministic. Assuming each branch is allocated to a separate processor, the execution time, given in terms of nonoverlapping assignment statements, is at most $O(N)$ and at best $O(\log N)$. The algorithm's performance, however, is not germane to this discussion.

The activity taking place along each branch k involves a search for the next entry $X[j]$ having a nonzero value and the action of adding the value of $X[j]$ to $X[k]$, as long as $X[k]$ is nonzero. Logically, the search requires us to evaluate the predicate

$$\exists\, j : k < j \leq N : X[k] > 0 \wedge X[j] > 0 \wedge \neg(\exists\, i : k < i < j : X[i] > 0)$$

and to perform the action

$$X[k],\, X[j] := (X[k] + X[j]),\, 0$$

using the value of j bound by the predicate evaluation and ensuring that the predicate evaluation and the assignment are performed as a single atomic action.

These requirements match directly with the definition of a Swarm transaction: an atomic inspection and transformation of the dataspace. Swarm partitions the dataspace into several subsets. These include the tuple space, a finite set of data tuples, and the transaction space, a finite set of transactions. Pairing a type name with a sequence of values creates an element of the dataspace. In addition, a transaction has an associated behavior specification. Transaction execution is modeled as a transition between dataspaces. An executing transaction examines the dataspace, then deletes itself from the transaction space and, depending on the results of the dataspace examination, modifies the dataspace by inserting and deleting tuples and by inserting (but not deleting) other transactions. Note that a query that fails removes itself, leaving the dataspace otherwise unchanged. A Swarm program begins executing from a valid initial dataspace and continues until the transaction space is empty. On each execution step, a transaction is chosen nondeterministically from the transaction space and executed. The transaction selection is fair in the sense that each transaction in the space will eventually be chosen.

Returning to our summation example, we can store the array X as a collection of tuples and reformulate each cobegin-coend branch as a Swarm transaction. We can represent X in tuple form by encoding each array entry $X[k]$ with value v as a tuple (k,v) of type entry. The initial tuple space configuration becomes

$$\{\; k : 1 \leq k \leq N : \text{entry}(k, X[k]) \;\}$$

The predicate/action pair associated with branch k can be rewritten as the following transaction. In Swarm, a comma is used in the place of the $\wedge$ operator in predicates.

```
Sum(k) ≡
  ι,μ,ν: k<ι≤N :
    entry(k,μ), entry(ι,ν),
    ¬(∃δ,σ : k<δ<ι : entry(δ,σ))
  →
    entry(k,μ)†, entry(ι,ν)†,
    entry(k,μ+ν)
```

This transaction differs from the original formulation in one important respect. Instead of maintaining information about the array entries that have been zeroed (that is, including a tuple entry(ι,0) in the dataspace), we simply delete these tuples, as indicated by the † symbol in the action part. This makes the original test $\neg(\exists i : k<i<j : X[i]>0)$ simply a test for the presence of any tuple with index in the specified range.

To complete the translation to a shared dataspace program, we need to ensure that for each k a Sum(k) transaction is initially in the transaction space and that successful transactions recreate themselves. The following definition of the initial transaction space guarantees the former condition.

$$\{\; k : 1 \leq k \leq N-1 : \text{Sum}(k) \;\}$$

The reinsertion of Sum(k) can be added to the action part of the transaction:

```
Sum(k)≡
  ι,μ,ν : k<ι≤N :
    entry(k,μ), entry(ι,ν),
    ¬(∃δ,σ : k<δ<ι : entry(δ,σ))
  →
    entry(k,μ)†, entry(ι,ν)†,
    entry(k,μ+ν),
    Sum(k)
```

The resulting Swarm program can now be simplified by removing some of the biases inherited from the original block-structured formulation. There is no need to ensure, when adding two entries, that the entries in between have been zeroed — this condition is only an artifact of the iterative method used by the original program. Associating a transaction with each entry in the array (except for the last one) is also unnecessary. A single transaction may be created at the start of the program, and with each successful execution the transaction may clone itself into two distinguishable transactions. This method leads to an exponential growth in the degree of parallelism employed by the program. The growth can be controlled by assigning each transaction an identifier from a finite set (for example, 1..N). In any case, all transactions eventually disappear when the summation is complete. Once the summation is completed, the query part of the Sum transaction always fails and is removed. With these changes the Swarm program becomes

```
Tuple space:
  { k : 1≤k≤N : entry(k,X[k]) }
Transaction space:
  { Sum(1) }
Transaction type definition:
  Sum(I) ≡
    ι1,ι2,μ1,μ2 : 1≤ι1<ι2≤N :
      entry(ι1,μ1), entry(ι2,μ2)
    →
      entry(ι1,μ1)†,
      entry(ι2,μ2)†,
      entry(ι1,μ1+μ2),
      Sum(ι1), Sum(ι2)
```

Figure 1 shows a sample execution of this program.

Figure 2 shows two possible visualizations of this algorithm, one for the block-structured program and one for the Swarm version. Active branches of the cobegin-coend construct, and transactions in the dataspace, are represented as balls. In both cases there are at most N balls. Each ball has one parameter that defines the ball position along a predefined horizontal line in the image. In the block-structured program, each ball's position is determined by the index value k associated with the particular branch. In the Swarm program, the value I of the Sum(I) transaction determines each ball's position.

The values associated with each array entry are mapped to a bar. The bar consists of a series of rectangles aligned next to each other along a predefined, horizontal

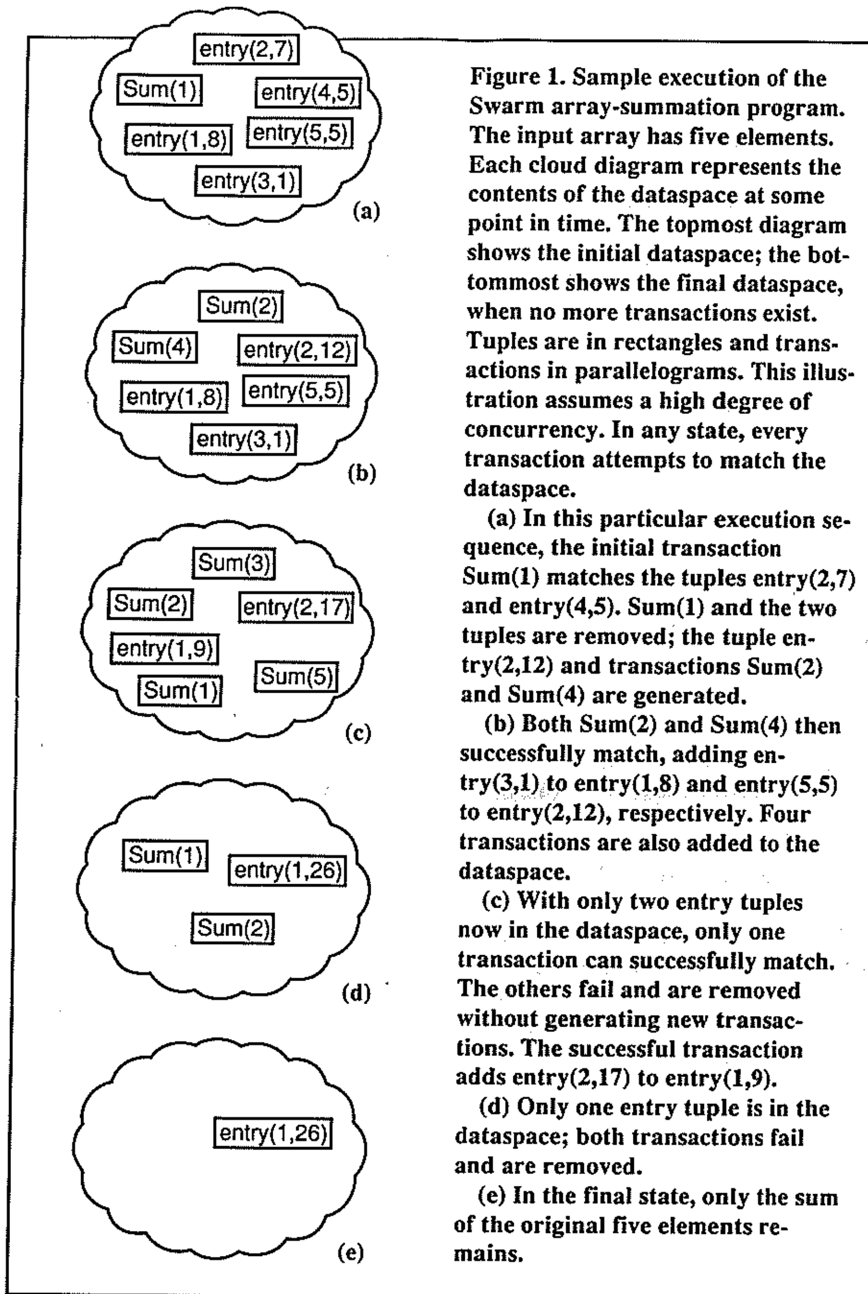


Figure 1. Sample execution of the Swarm array-summation program. The input array has five elements. Each cloud diagram represents the contents of the dataspace at some point in time. The topmost diagram shows the initial dataspace; the bottommost shows the final dataspace, when no more transactions exist. Tuples are in rectangles and transactions in parallelograms. This illustration assumes a high degree of concurrency. In any state, every transaction attempts to match the dataspace.

(a) In this particular execution sequence, the initial transaction Sum(1) matches the tuples entry(2,7) and entry(4,5). Sum(1) and the two tuples are removed; the tuple entry(2,12) and transactions Sum(2) and Sum(4) are generated.

(b) Both Sum(2) and Sum(4) then successfully match, adding entry(3,1) to entry(1,8) and entry(5,5) to entry(2,12), respectively. Four transactions are also added to the dataspace.

(c) With only two entry tuples now in the dataspace, only one transaction can successfully match. The others fail and are removed without generating new transactions. The successful transaction adds entry(2,17) to entry(1,9).

(d) Only one entry tuple is in the dataspace; both transactions fail and are removed.

(e) In the final state, only the sum of the original five elements remains.

line in the image. The length and position of each rectangle are determined by the value and the index of some entry in the array X.

We choose this particular visualization because it captures key properties used to prove the correctness of the two programs:

- The sum of the values of the array entries is constant, and the length of the bar stays constant.
- The number of nonzero entries in the array is reduced by one with each successful assignment or transaction execution, decreasing the number of rectangles.
- Once the number of nonzero entries drops to one, all active branches of the cobegin-coend construct terminate, and each transaction execution reduces the number of transactions in the dataspace by one. In both cases, the number of balls decreases until it reaches zero.

The similarity between the two proofs led to identical visualizations. Differences in the operational details are made evident by variations between the two sequences of images.

Next, we will consider the problem of specifying the visualization for each of the two programs. In Figure 2 we used two types of graphical objects:

Ball(position)
Rectangle(position, length)

Ball(n) indicates that the nth ball is to be depicted. Rectangle(n,l) indicates that the rectangle in position n of the bar is to have length l. We permit the length to be 0, and if a rectangle for position n is missing, a length of 0 is assumed.

Object generation rules provide a convenient way of specifying the mapping from program states to graphical objects. Given a particular program state, each rule defines a set of objects that must be included in the image. For example, in the block-structured program the set of rectangles is

$\{ k : 1 \le k \le N : \text{Rectangle}(k,X[k]) \}$

and is defined by the rule

$k : 1 \le k \le N :$
  true $\Rightarrow$ Rectangle(k,X[k])

This is to be interpreted as "for each entry $X[k]$, generate a graphical object Rectangle(k,$X[k]$)." The rectangles can be similarly specified for the Swarm program:

$\iota, v : 1 \le \iota \le N :$
  entry(ι,v) $\Rightarrow$ Rectangle(ι,v)

The interpretation of this is similar: "for each tuple entry(ι,v) in the dataspace, generate a graphical object Rectangle(ι,v)."

In the Swarm program, the definition of the balls is equally straightforward:

$\iota : 1 \le \iota \le N : \text{Sum}(\iota) \Rightarrow \text{Ball}(\iota)$

However, it is not immediately apparent how the definition might be specified for the block-structured program, unless additional variables are added to capture the necessary control state information and encode it as data. We can introduce such additional variables, often called auxiliary variables, in the form of a Boolean array $B[1..N]$ where $B[k]$ is true if and only if branch k of the cobegin-coend is active:

```
B[1..N] : array of Boolean := true;

cobegin for each k:=1..N–1;
   for j:=(k+1)..N loop
      if X[k]=0 then exit;
      if X[k]≠0 and X[j]≠0 then
```

```
                X[k],X[j] := (X[k]+X[j]),0;
            endif
        end loop;
        B(k) := false
    coend
```

The definition of the balls for this program then becomes

$$k : 1 \le k \le N : B(k) \Rightarrow Ball(k)$$

The need to add auxiliary variables to properly capture the state information in the block-structured program highlights one of the advantages of the shared dataspace paradigm: All state information is contained in the dataspace.

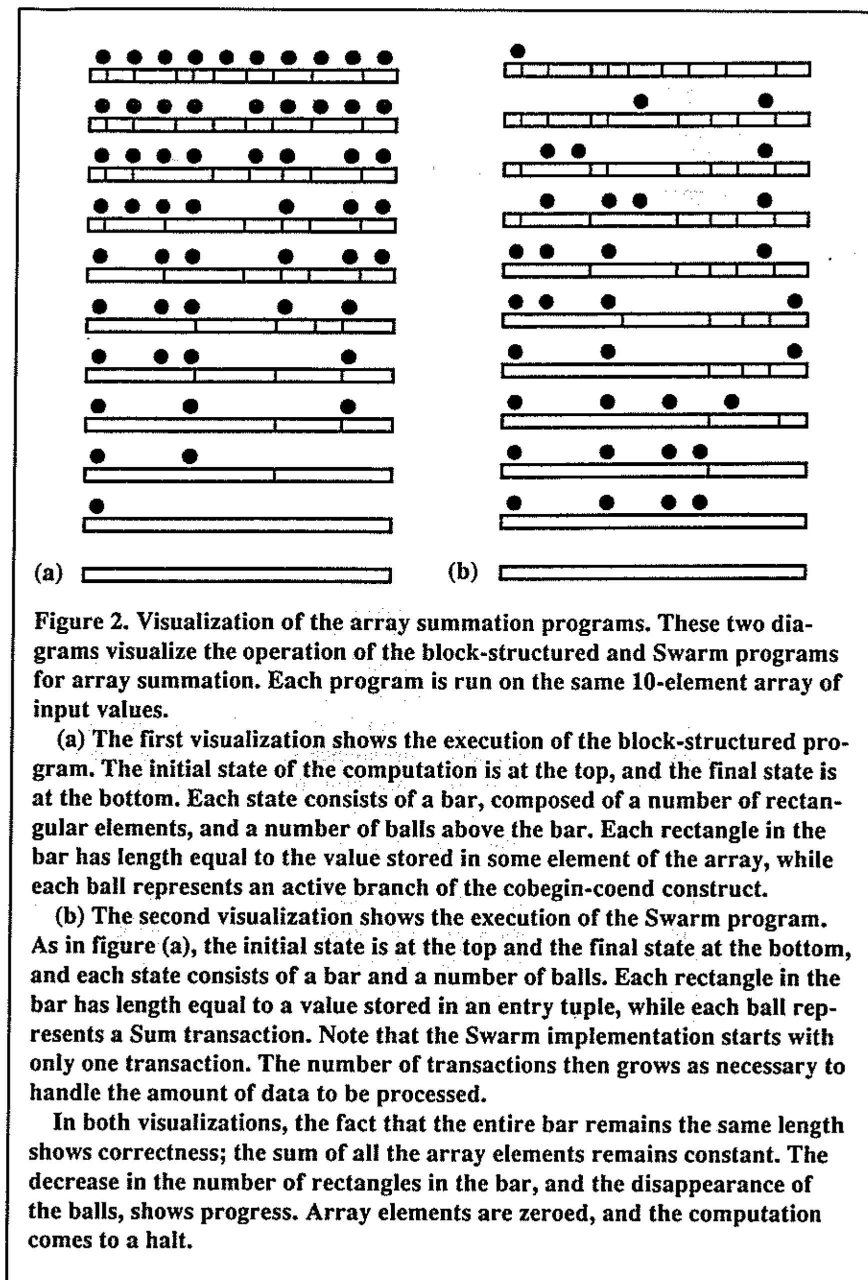


Figure 2. Visualization of the array summation programs. These two diagrams visualize the operation of the block-structured and Swarm programs for array summation. Each program is run on the same 10-element array of input values.

(a) The first visualization shows the execution of the block-structured program. The initial state of the computation is at the top, and the final state is at the bottom. Each state consists of a bar, composed of a number of rectangular elements, and a number of balls above the bar. Each rectangle in the bar has length equal to the value stored in some element of the array, while each ball represents an active branch of the cobegin-coend construct.

(b) The second visualization shows the execution of the Swarm program. As in figure (a), the initial state is at the top and the final state at the bottom, and each state consists of a bar and a number of balls. Each rectangle in the bar has length equal to a value stored in an entry tuple, while each ball represents a Sum transaction. Note that the Swarm implementation starts with only one transaction. The number of transactions then grows as necessary to handle the amount of data to be processed.

In both visualizations, the fact that the entire bar remains the same length shows correctness; the sum of all the array elements remains constant. The decrease in the number of rectangles in the bar, and the disappearance of the balls, shows progress. Array elements are zeroed, and the computation comes to a halt.

Visual abstraction

Declarative approaches treat the visualization of computations as the application of a function to the computational state, yielding an image. We call this function the visualization function. The major parts of our visualization model[11] are the representation of the function's domain and range and the method of declaring the function.

Formally, the visualization function maps the set of all states to the set of all images:

$$V : \mathit{States} \rightarrow \mathit{Images}$$

The function could be declared in this manner. However, we wish to use visualization as a tool for understanding computations, and we feel that understanding results from proper abstractions. We are therefore primarily interested in abstraction, the translation of state information into symbolic form. The problem of rendering, the representation of symbols in an image, is less important, although it does involve some interesting and difficult problems.

We therefore divide the visualization function into two parts, an abstraction function and a rendering function. These can be formally described as

$$A : \mathit{States} \rightarrow \mathit{Objects}$$
$$R : \mathit{Objects} \rightarrow \mathit{Images}$$

with the visualization function equal to the composition of these functions. Again, our primary interest is in the specification of the abstraction function.

To specify the abstraction function, we need some information about its domain (*States*) and range (*Objects*). *States* is the set of all possible computational states. In many paradigms for concurrent computation — for example, communicating processes — the state is difficult to define and even more difficult to capture, as it involves a variety of such widely separated and diverse data as contents of process memories, program counters and code, and message buffers. This adds unnecessary complexity to the process of visualizing a computation.

However, in the shared-dataspace paradigm, and particularly in the Swarm language, all state information is represented in the dataspace. The visualization system can, in principle, examine the dataspace without interfering with the underlying computation. Representing all information as typed tuples further simplifies the paradigm compared to other models. Therefore, using the shared-dataspace paradigm as the computational model has definite advantages in declarative visualization.

The range of the abstraction function (*Objects*) is the set of all possible symbolic representations of states; each such representation is a set of primitive graphical objects. We can extend our uniform data representation to *Objects* by representing each graphical object using the tuple notation. A graphical object will be represented

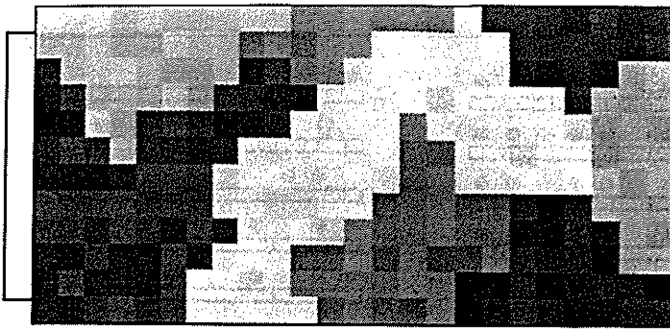


Figure 3. Input image to region-labeling program. The input consists of an image divided into a rectangular array of pixels of various intensities. The program should mark all connected regions of approximately equal intensity with the same label.

by a tuple *type*(parameters), where the type specifies the general class of object (Circle, Line, Rectangle) and the parameters specify the particular object, giving position, size, orientation, color, etc.

We specify the abstraction function itself by a set of rules having the form

variables : *query over dataspace*
$\Rightarrow$
list of object tuples

where the variables are existentially quantified implicitly. Such a rule defines a set of constantly changing object tuples as follows. The query is evaluated against the current dataspace, and for each successful match the variable bindings are used to instantiate the list of tuples on the right-hand side. All the resulting tuples are members of the set. For any state of the computation, the resulting set of graphical objects is the union of the sets produced by each visualization rule.

Thus, if a new dataspace tuple is asserted that matches with some visualization rule, all resulting tuples are immediately added to the set of objects. Likewise, if a dataspace tuple is retracted, any members of the set generated by a rule matching the tuple are immediately removed. Please note, though, that this is a model of our approach to visualization; an implementation would not necessarily compute the graphical objects and image in this fashion.

Visualization methodology

Our visualization methodology provides guidelines for constructing animations in the declarative model. Most systems recognize that certain methods of representing data are more effective than others, but they lack the concept of a methodology, at least in the sense of general rules for constructing animations. We hope to address this problem, and place visualization on a more firmly technical foundation. We base our methodology on program correctness. Program correctness techniques express and prove properties about programs with the dual goals of demonstrating that the program is correct and understanding why it is correct.

Our rationale for using program correctness in our methodology is based on two observations. First, program correctness seeks to explain the behavior of computations. Since our own goal is the use of visualization for understanding, the two mesh nicely. Second, program correctness has proved to be quite successful in its goals, suggesting that the translation of its principles to visualization might be equally successful. Systems for expressing and proving program properties have been developed for several programming paradigms.[2,12,13] A system for expressing and proving properties of shared dataspace programs has also been developed.[6]

The types of properties that can be expressed in these systems fall into two broad categories, safety and progress properties. Safety properties (such as invariants) give conditions that the program may not violate. Referring to the array summation program, property (1) — the sum of the values of the array entries is constant — is a safety property. Progress properties tell what the program is required to do. Properties (2) and (3) in the summation programs — the number of nonzero entries is steadily reduced, and the program terminates after the number of such entries reaches one — are progress properties.

We believe that the same properties used to prove programs correct can be used to indicate what aspects of those programs should be represented in the visualization. Further, we believe that the structure of the property, as it is expressed in whatever proof logic is used, provides a guide to how that property should be visualized. At the highest level, this means that invariants would normally be visualized as stable patterns while progress properties would be visualized as evolving patterns.

Application to region labeling

To illustrate our methodology, we use the image-processing region-labeling problem. In this problem, we are given a digitized black-and-white image such as that shown in Figure 3. First, we want to divide this image into connected regions where each region's pixels are of approximately the same grey level. We then select one pixel from each region to identify that region in later processing activities. We can state the program requirements more formally as follows.

The problem input is an M-by-N array (Intensity) and a function (Threshold). Intensity represents a digitized image divided into pixels; it assigns to each pixel a value representing its brightness. Threshold maps the intensities to a smaller range; for example, given an intensity i, Threshold(i) might be the integer part of $i/10$. Two pixels are in the same bucket if the Threshold function produces the same result when applied to their intensities.

Two pixels are neighbors if they share a side; the pixel at coordinate $\langle x,y \rangle$ neighbors the pixels $\langle x-1,y \rangle$, $\langle x+1,y \rangle$, $\langle x,y-1 \rangle$, and $\langle x,y+1 \rangle$. Two pixels are in the same region if they are connected by a path of neighboring pixels, all of which are in the same bucket. The output of the region-labeling program is to be an M-by-N-array Label, where two pixels are assigned the same label if and only if they are in the same region. In addition, a single "master pixel" is to be identified in each region.

A nondeterministic algorithm for this problem can be described as follows:

Assign a unique label to each pixel
while
  there are neighboring pixels
  $p1$ and $p2$ in the same bucket,
  with $p2$'s label less than
  that of $p1$
loop

```
        Relabel p1 with the label of p2
    end loop
    The "master pixels" are those
      that retain their original label
```

Figure 4 illustrates a Swarm implementation of this algorithm. The dataspace at all times contains a tuple of the form

is_labeled(P,I,L)

for each pixel *P*. This tuple indicates that the pixel with coordinate *P* and intensity *I* is currently labeled with label *L*. Pixel labels are just coordinates; each pixel is initially labeled with its own coordinate. We assume that a primitive operation to compare two labels is provided; we use the symbol < for this.

```
Definitions:
  Pixels = { x,y, : 1≤x≤N, 1≤y≤M : (x,y) }

Tuple space:
  { ρ : ρ ∈ Pixels : is_labeled(ρ,Intensity(ρ),ρ) }

Transaction space:
  { ρ : ρ∈Pixels : Label(ρ) }

Transaction type definition:

  Label(P) ≡
      ι1,λ1,,λ2:
        is_labeled(P,ι1,λ1), ρ neighbors P, is_labeled(ρ,ι2,λ2),
        Threshold(ι1) = Threshold(ι2),λ2<λ1
      →
        is_labeled(P,ι1,λ1)†,
        is_labeled(P,ι1,λ2)
    ||
        true
      →
        Label(P)
```

Figure 4. Swarm program for region labeling. The Label(*P*) transaction performs two operations. The first operation searches for a pixel ρ which neighbors *P*, is in the same bucket, and has a smaller label. If such a pixel is found, the transaction relabels *P* with the label of ρ by removing the old is_labeled tuple and adding a new one.

Simultaneously with the previous operation, the predicate true is evaluated. This obviously succeeds, so the program adds a new Label(*P*) transaction to the dataspace.

We wish to visualize the operation of this algorithm. The first problem with constructing a visualization is to determine the graphical objects that will be used and their layout, that is, the arrangement of the objects. In this region-labeling example, there is a very natural layout where pixels are translated into squares, and the squares are arranged in a grid according to the coordinates of the pixels. We will provide the squares with borders. Both squares and borders can be colored in various ways.

Our graphical-object universe therefore consists of two object types: squares and borders between two squares. Both types of objects have properties that define their position and color. These can be represented in tuple notation:

Square(*coordinate*, *color*)
Border(*coordinate*1, *coordinate*2, *color*)

The representation of colors may be device-dependent. We will assume a colorize function is available that maps pixel labels to the color space. The function is one-to-one, so distinct input labels have distinct colorization values. The range of colorize does not include all colors that can be produced by the device. Colors not in the range of colorize are called recognizable.

The rendering function will translate collections of these objects into a screen image. The overall result will be a grid as illustrated in Figure 5. We are not interested in such rendering function details as the mapping of square coordinates to points on the screen, and the determination of the location of borders from the two coordinates in the tuple. However, we will assume the rendering function has the following properties:

- If no object tuple is mapped to some screen point, that point is rendered in a recognizable color called the background color.
- If two or more object tuples with different colors map to some screen point, that point is rendered in a recognizable color called the overlap color.

Perhaps the most direct representation of the region-labeling is to map each pixel to a square having a color determined by its current label:

```
V0: ρ,ι,λ:
      is_labeled(ρ,ι,λ)
    ⇒
      Square(ρ,colorize(λ))
```

Figure 6 illustrates this visualization.

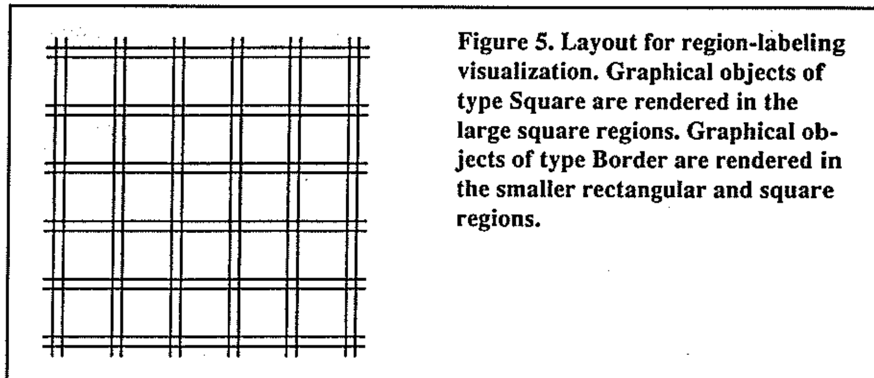


Figure 5. Layout for region-labeling visualization. Graphical objects of type Square are rendered in the large square regions. Graphical objects of type Border are rendered in the smaller rectangular and square regions.

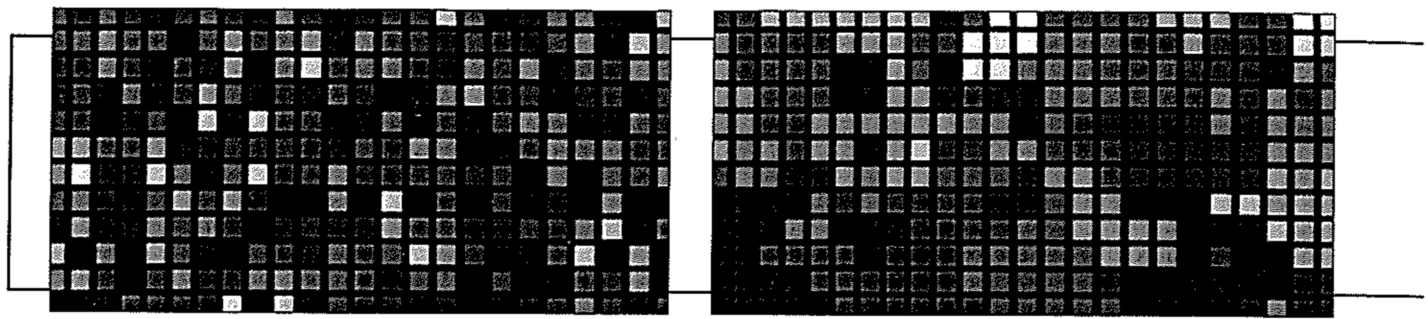


Figure 6. First visualization of region labeling. This sequence of four images visualizes the operation of the region-labeling program using the abstraction rule V0. Each image results from applying V0 and the rendering function to the dataspace at some point in the computation.

The leftmost image represents the initial state of the computation. Each pixel has a unique label and is assigned a unique color by the colorize function.

The next image shows the state after some time has elapsed. Some of the pixels have been relabeled and have changed

This rather simple visualization may appear to be all that one needs to understand the program's behavior. However, this visualization contains several flaws. It is impossible to tell if the final result is a correct labeling. The desired output of a "master pixel" for each region, although present in the data, is not in the visualization. Finally, and most severely, the display captures the low-level mechanics of the program execution rather than the fundamental program properties used to reason about the computation.

We will now apply our visualization methodology to generate another visualization of this algorithm (Figure 7). The principal invariants and progress conditions used in the correctness proof (not presented here) for the program are:

I1: Region boundaries are stable.

I2: Two neighboring pixels belonging to two different regions never have the same label.

I3: In every region, the pixel having the smallest coordinate is labeled by its own coordinate.

P1: If a pixel ρ has a neighbor belonging to the same region and labeled by the smallest label in that region, ρ will eventually be labeled by the smallest label in that region.

As stated earlier, invariants are rendered as stable patterns such that violations of the invariant are easily observed. However, when formally stated in a logical calculus, I1, I2, and I3 have very distinct forms. I1 is universally quantified such that all region boundaries remain the same. I2 involves a negation: it is not the case that two neighbors in different regions have the same label. I3 includes an embedded existential quantification (for every region, there is some pixel that has the smallest coordinate and is labeled with its own coordinate). Because of this, we expect each to be visualized in a different manner. (We are currently investigating the hypothesis that the form of the invariant can be used to indicate the manner in which it should be visualized.)

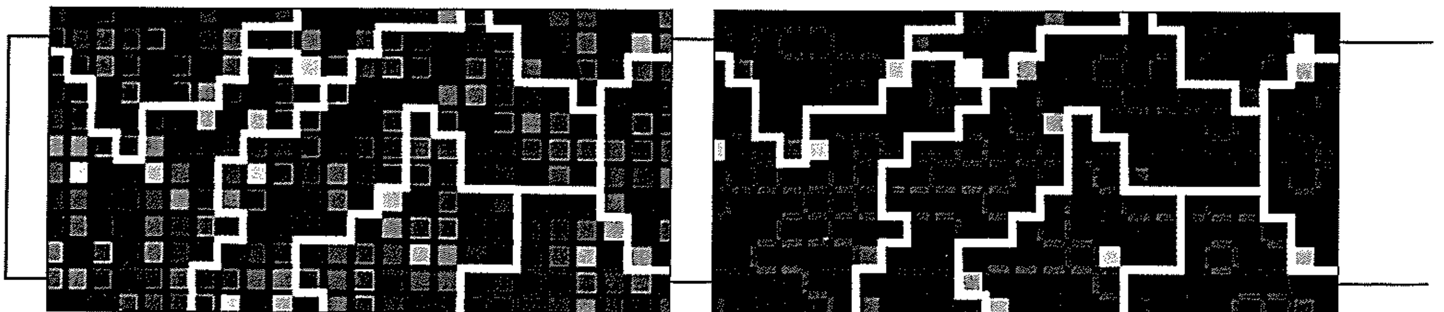
I1 requires a part of the computation state to remain unchanged throughout the computation. This strongly suggests the need to render this part of the state in the visualization; if the state changes illegally, the change will be detectable in the image. We will therefore render the region boundaries in some recognizable color, for example white. If I1 is violated these borders will move or disappear during execution.

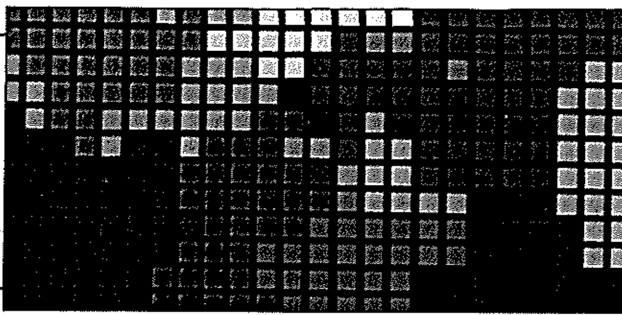
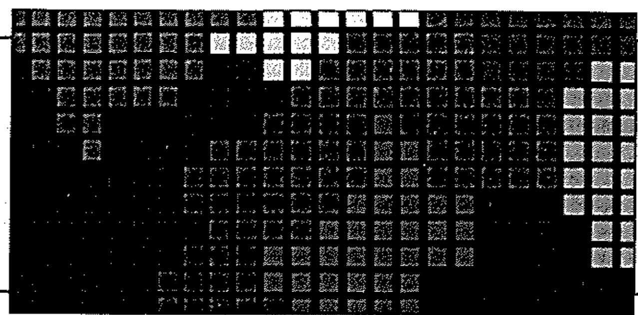
We will use the layout borders to render the region boundaries. By using the eight_neighbors relationship, the small bordering boxes at the corners of the squares will be filled, largely for aesthetic reasons. The rule that visualizes I1 is

V1: ρ1,ι1,λ1,ρ2,ι2,λ2:
 is_labeled(ρ1,ι1,λ1),

Figure 7. Second visualization of region-labeling. This sequence of four images visualizes the operation of the region-labeling program using the abstraction rules V1 through V4. The two program runs in Figures 6 and 7 are identical, and corresponding photographs were taken after equal amounts of computation; only the visualization differs. The bright yellow, overlap (error) color does not appear because the program and implementation ran correctly.

The leftmost image represents the initial state of the computation. The boundaries between regions are clearly shown in white (by V1). Each pixel has its own coordinate-as-label, and so is colored by V3. V4 renders the borders between pixels in



color. Note that small regions of the same color are beginning to form.

The third image shows the state after additional time has elapsed. The relabeling has continued, and the regions of the same color have grown larger.

The fourth image shows the state when the computation has completed. Two pixels have the same label (are the same color) if and only if they are in the same region.

is_labeled(ρ2,ι2,λ2),
ρ1 eight_neighbors ρ2,
Threshold(ι1) ≠ Threshold(ι2)
⇒
Border(ρ1,ρ2, white)

Invariant I2 expresses a condition that must not occur. We therefore detect violations of I2. This can be done by rendering, in a recognizable color such as the overlap, the border between any two neighboring pixels that violate I2. This rendering is accomplished by the rule:

V2: ρ1,ι1,ρ2,ι2,λ:
is_labeled(ρ1,ι1,λ),
is_labeled(ρ2,ι2,λ),
ρ1 neighbors ρ2,
Threshold(ι1) ≠ Threshold(ι2)
⇒
Border(ρ1,ρ2,overlap)

Visualizing I3 seems to cause difficulties, because the invariant refers to a pixel that, although known to exist and to be unique for each region, must be precomputed. However, a closer examination of I3 shows that this invariant specifies the "master pixel" in each region—a pixel that is only identified when the algorithm has completed. I3 can therefore be visualized by introducing some initial uncertainty that is eliminated as the computation progresses. The idea is to color all pixels that retain their initial label assignment. Initially all pixels are colored; as the program proceeds, pixels revert to the background color, leaving only the pixel with the smallest coordinate (the master) colored. The following rule can be used for this:

V3: ρ,ι:
is_labeled(ρ,ι,ρ)
⇒
Square(ρ,colorize(ρ))

Progress properties are to be captured by pairs of patterns, such that a state generating the first pattern will lead to a state generating the second pattern. To represent P1, we wish to capture the idea that if some pixel ρ can be relabeled, it eventually will be relabeled. This can be visualized by marking the boundaries between pixels that are in the same region, but have different labels, with some recognizable color such as red. Progress is recognized by the reduction in the total number of red boundaries:

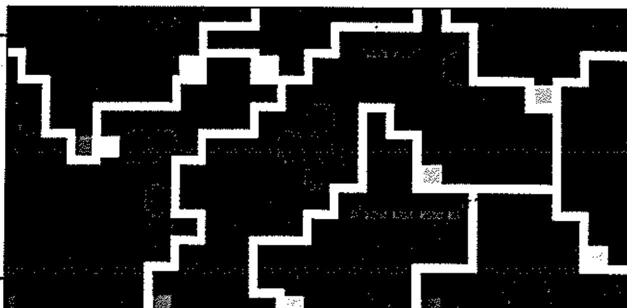
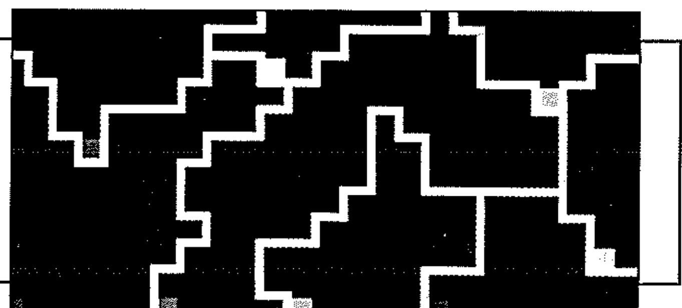
V4: ρ1,ι1,λ1,ρ2,ι2,λ2:
is_labeled(ρ1,ι1,λ1),
is_labeled(ρ2,ι2,λ2),
ρ1 neighbors ρ2,
Threshold(ι1) = Threshold(ι2),
λ1≠λ2
⇒
Border(ρ1,ρ2,red)

Examining all four of these rules together, V1 outlines the boundaries of the regions in white, V2 detects violations of

the same region, but with different labels, in red.

The next two images represent the state after some time has elapsed. Some of the pixels have been relabeled. Those that do not retain their original label have disappeared, as V3 no longer renders them. The white borders have not changed. The number of red borders has steadily decreased, indicating progress.

The fourth image shows the state when the computation has completed. No pixels can be relabeled, as indicated by the total absence of red boundaries. The single, master pixel in each region is clearly indicated by V3.

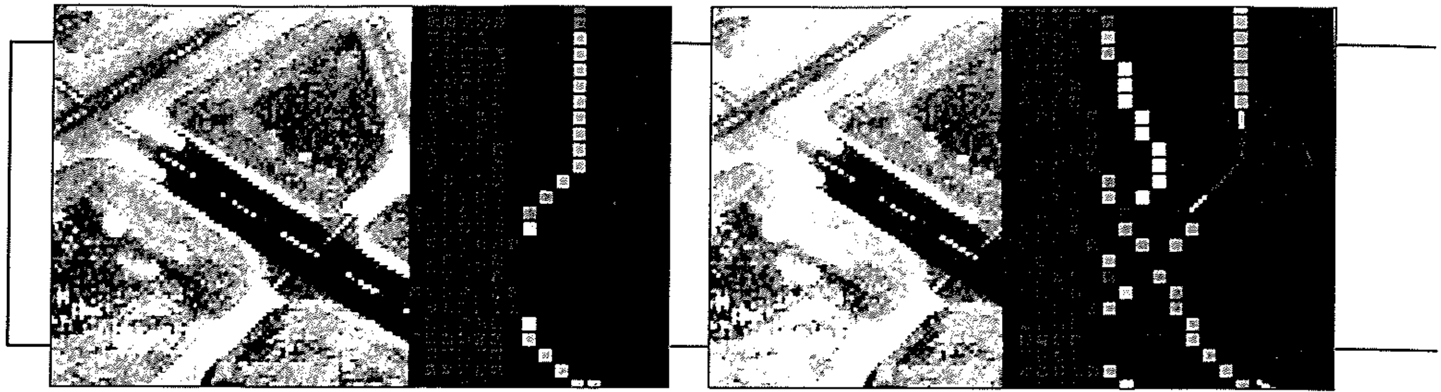


Figure 8. Visualization of polygon-construction free of intervention. This sequence of images visualizes the operation of a program that constructs polygons representing the edges of an image. The sequence shows several states of the computation for a small portion of the image. The earliest state is at the left. The visualization is overlaid on a black-and-white photograph of the terrain being scanned.

Processing of a particular edge pixel occurs in four distinct phases, as discussed in the text. In this sequence of images all

the desired output properties, V3 marks the pixel having the smallest label in each region and thus the final master pixel, and V4 marks the boundaries between pixels that are in the same region but have different labels. As the algorithm progresses, the areas within red boundaries expand toward the white boundaries. The disappearance of all red boundaries marks completion. Figure 7 depicts a visualization of the region-labeling program in which rules V1 through V4 are in effect.

This simple example demonstrates the use of formal program properties as the basis for deciding which visualization rules are appropriate. We hope that this approach will lead to the development of a set of general rules for constructing program visualizations based on the structure of the properties used in the program correctness proof and not on knowledge of the operational details of the program. Such an approach would aid true exploration and understanding of the program.

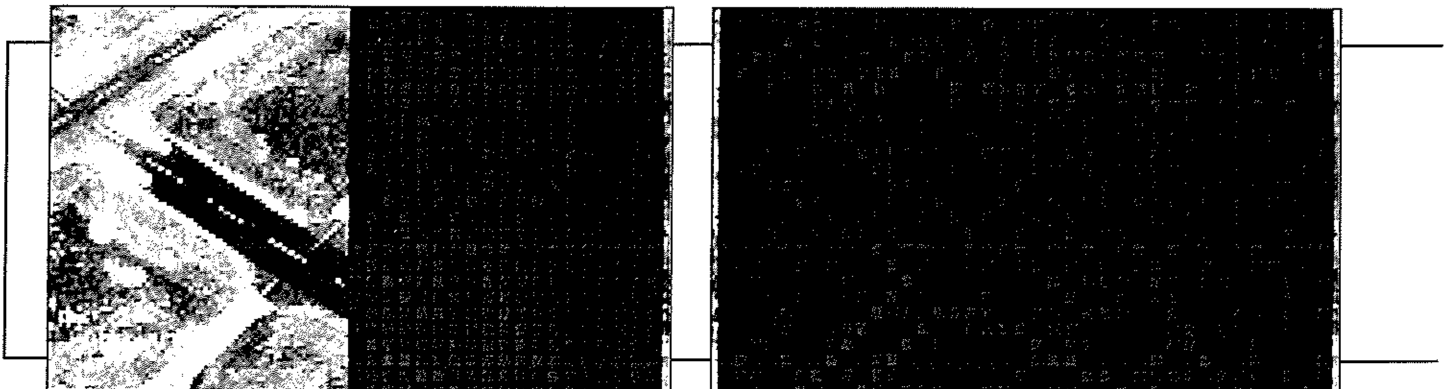
Intervention semantics

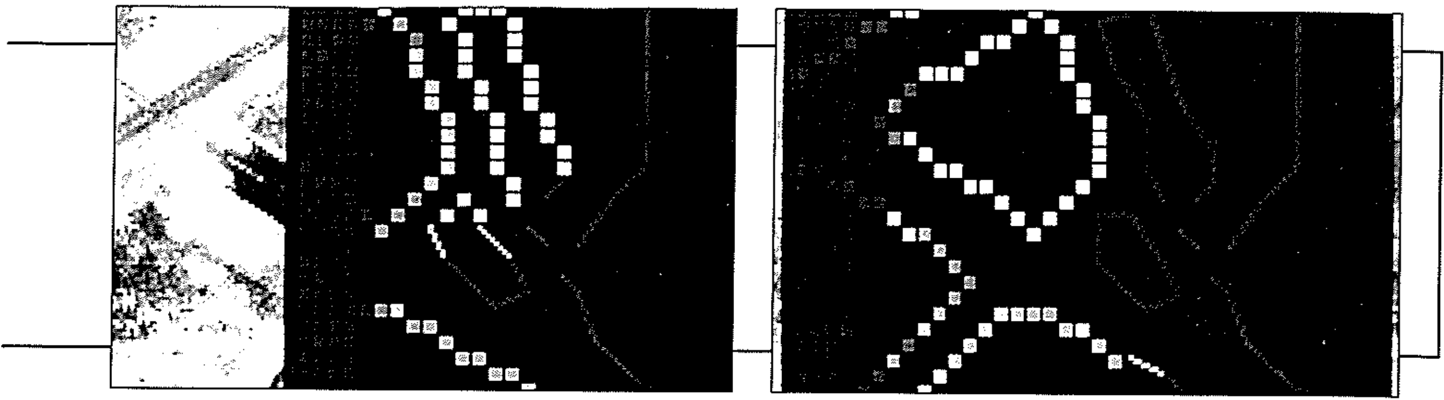
During monitoring and debugging of concurrent programs, one must minimize the degree of interference with the program execution to avoid altering the phenomenon being observed. However, when using visualization to study and understand concurrent computations, the noninterference requirement can be relaxed somewhat. This is because there are no errors that could be masked by slight changes in the order in which events occur. The questions we want to address here are: What interventions are permissible, and how do we guarantee that they do not change the semantics of the program being observed?

To answer these questions, we must consider the underlying model of concurrency used to define the semantics of the language and to construct the proof system. Most models of concurrency, including the operational model employed by Swarm, have no notion of time. They rely, instead, on fairness and atomicity assumptions. In Swarm, for instance, each transaction present in the dataspace is eventually executed, and its execution is an atomic transformation of the dataspace. This means that one has a great degree of latitude in changing the scheduling policy, the order in which transactions are selected and executed, without fearing any changes in the program semantics, as long as fairness is preserved. Since the notion of fairness itself is a very weak requirement — each transaction is eventually executed — certain transactions may be ignored for very long periods of time while others may be selected quickly.

We are only now starting to consider the implications of manipulating the scheduling policy on the visualization methodology. Consider, for instance, the sequence of photographs shown in Figure 8. They depict several states in the execution of a

Figure 9. Visualization of polygon-construction with scheduling restrictions. The program from Figure 8 is run on the same portion of the image, but the scheduling policy has been changed to force each phase to complete before the next begins.





phases are allowed to operate simultaneously. Thus, we see newly created pixels (the red and blue pixels to the left of the image), nonedge elimination (the red pixels and some blue pixels disappear), master selection (the multicolored pixels, which change color as they are relabeled), and polygon construction (the blue growing lines and red final lines). Overlap among the four phases makes understanding the computation more difficult.

relatively complex program.* The program assumes an airborne platform that scans an airport below, constructing an image that is unbounded on one side. For presentation purposes, we show only a small area of the unbounded image. A hardware edge-detector transforms the incoming image into a binary-edge image, which the program converts to a symbolic representation as polygons.

Although this visualization captures faithfully the order in which actions would occur in an actual execution, different parts of the image are in different processing phases, which makes the understanding of the program difficult for someone seeing the visualization for the first time. In Figure 9, the same visualization rules are applied to the same program, but the scheduling policy has been altered. For a finite portion of the input image, transactions are being delayed in a manner that reveals the logical sequence of operations associated with each pixel in the input image:

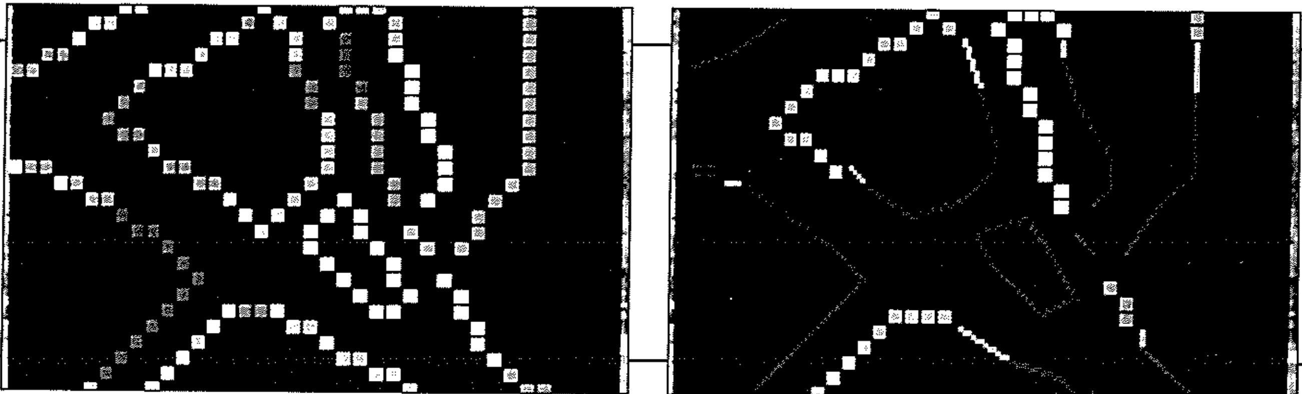
(1) A hardware edge-detector transforms the incoming image into a binary-edge image.
(2) Nonedge pixels and edge pixels having three or more neighboring edge pixels are eliminated.
(3) As a preliminary step to generating the polygons, a version of the region-labeling program selects a master in chains of connected edge pixels.
(4) The master defines the starting point for the polygon construction along each chain.

The scheduling policy imposed in Figure 9 inhibits, over a small area of the image, the execution of any transaction involved in performing one of the processing phases listed above until all transactions associated with the preceding phase terminate. Since the processing associated with each phase is independent of the subsequent phases, and is completed in a finite number of steps for any bounded image, the fairness of the execution is preserved. This would not be the case if the scheduling restrictions were applied to the entire image; because the image is unbounded to one side, the input phase would never terminate.

This example shows that schedule manipulation can become an important component of a visualization methodology aimed at exploring properties of concurrent computations. Yet an appropriate methodology and associated support tools for the investigation of such manipulations must still be developed.

*This particular program was actually written in a shared-dataspace language called SDL, the predecessor of Swarm. Because the visualization does not change under recoding into Swarm, we take the liberty of discussing the program as if it were written in Swarm.

These images show the computation midway through each of the four phases. The separation of concerns enhances understanding of the program's operation.



That visualization can play a key role in the exploration of concurrent computations is central to the ideas we have presented. Equally important, although given less emphasis, is our concern that the full potential of visualization may not be reached unless the art of generating beautiful pictures is rooted in a solid, formally technical foundation. We have shown that program verification provides a formal framework around which such a foundation can be built. Making these ideas a practical reality will require both research and experimentation.□

Acknowledgments

This work was supported by the Department of Computer Science, Washington University, St. Louis, Missouri. The authors express their gratitude to Jerome R. Cox, department chairman, for his support and encouragement. We also thank Conrad Cunningham for his input regarding this article and his contribution to the development of the shared dataspace concept.

References

1. C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall Int'l, Englewood Cliffs, N.J., 1985.

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

3. L. Brownston et. al., *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming*, Addison-Wesley, Reading, Mass., 1985.

4. S. Ahuja, N. Carriero, and D. Gelernter, "Linda and Friends," *Computer*, Vol. 19, No. 8, Aug. 1986, pp. 26-34.

5. G.-C. Roman and H.C. Cunningham, "A Shared Dataspace Model of Concurrency — Language and Programming Implications," *Proc. Ninth Int'l Conf. Distributed Computing Systems*, CS Press, Los Alamitos, Calif., 1989, Order No. 1953, pp. 270-279.

6. H.C. Cunningham and G.-C. Roman, "A Unity-Style Programming Logic for a Shared Dataspace Language," Tech. Report WUCS-89-5, Computer Science Dept., Washington Univ., St. Louis, Mar. 1989.

7. M.H. Brown and R. Sedgewick, "A System for Algorithm Animation," *Computer Graphics* (Proc. SIGGraph 84), Vol. 18, No. 3, July, 1986, pp. 177-186.

8. E. Helttula, A. Hyrskykari, and K.-J. Räihä, "Graphical Specification of Algorithm Animations with Aladdin," *Proc. Hawaii Int'l Conf. Systems Sciences, Vol. II, Software Track*, CS Press, Los Alamitos, Calif., Order No. 1912, 1989, pp. 40-54.

9. T.G. Moher, "Provide: A Process Visualization and Debugging Environment," *IEEE Trans. Software Eng.*, Vol. 14, No. 6, June 1988, pp. 849-857.

10. J.D. Foley and C.F. McMath, "Dynamic Process Visualization," *IEEE Computer Graphics and Applications*, Vol. 6, No. 2, Mar. 1986, pp. 16-25.

11. G.-C. Roman and K.C. Cox, "Declarative Visualization in the Shared Dataspace Paradigm," Proc. 11th Int'l Conf. Software Eng., CS Press, Los Alamitos, Calif., 1989, Order No. 1941, pp. 34-43.

12. L. Lamport, "A New Approach to Proving the Correctness of Multiprocess Programs," *ACM Trans. Programming Languages and Systems*, Vol. 1, No. 1, July 1979, pp. 84-97.

13. Z. Manna and A. Pneuli, "How to Cook a Temporal Proof System for Your Pet Language," *Proc. 10th ACM Symp. Principles of Programming Languages*, 1983, pp. 141-154.

Moving?

PLEASE NOTIFY US 4 WEEKS IN ADVANCE

Name (Please Print)

New Address

City | State/Country | Zip

MAIL TO:
IEEE Service Center
445 Hoes Lane
Piscataway, NJ 08854

ATTACH LABEL HERE

- This notice of address change will apply to all IEEE publications to which you subscribe.
- List new address above.
- If you have a question about your subscription, place label here and clip this form to your letter.

Gruia-Catalin Roman is an associate professor in the Department of Computer Science at Washington University in St. Louis. He is currently researching language and visualization support for large scale concurrency and interactive, high-speed computer-vision algorithms. His consulting work involves developing custom software engineering methodologies and training programs. His previous research was concerned with requirements and design methodologies for distributed systems.

Roman was a Fulbright Scholar at the University of Pennsylvania at Philadelphia, where he received a BS degree in 1973, an MS degree in 1974, and a PhD degree in 1976, all in computer science. He is a member of Tau Beta Pi, ACM, and the IEEE Computer Society.

Kenneth C. Cox is a doctoral student at Washington University in St. Louis. His previous research has been concerned with image processing, feature detection, and visualization of concurrent algorithms.

Cox received a BS in both computer science and electrical engineering in 1985, and an MS in computer science in 1987, all from Washington University.

The authors' address is School of Engineering and Applied Science, Dept. of Computer Science, Washington University, Campus Box 1045, One Brookings Dr., St. Louis, MO 63130-4899. Roman's electronic mail address is roman@cs.wustl.edu; Cox can be reached at kcc@cs.wustl.edu.