Report Number: WUCS-88-27

1988-09-01

Declarative Visualization in the Shared Dataspace Paradigm

Authors: Gruia-Catalin Roman and Kenneth C. Cox

This paper is concerned with the use of program visualization as a means for the understanding, debugging, and monitoring of large-scale concurrent programs. Following an overview of the shared dataspace paradigm and the declarative approach to visualization, the paper discusses: (1) mechanisms for specifying declarative visualization in the shared dataspace paradigm and ways of relating the specifications to program verification; (2) a computational model which provides a unified framework for comparing both visual and nonvisual algorithms; and (3) strategies for implementing declarative visualization on parallel machines.

DECLARATIVE VISUALIZATION IN THE SHARED DATASPACE PARADIGM

Gruia-Catalin Roman and Kenneth C. Cox

WUCS-88-27

September 1988

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

As appeared in Proceedings of the 11th International Conference on Software Engineering, May 1989, pp. 34-43.

DECLARATIVE VISUALIZATION IN THE SHARED DATASPACE PARADIGM

Gruia-Catalin Roman and Kenneth C. Cox

Department of Computer Science
WASHINGTON UNIVERSITY
Saint Louis, Missouri 63130

ABSTRACT

This paper is concerned with the use of program visualization as a means for the understanding, debugging, and monitoring of large-scale concurrent programs. Following an overview of the shared dataspace paradigm and the declarative approach to visualization, the paper discusses: (1) mechanisms for specifying declarative visualization in the shared dataspace paradigm and ways of relating the specifications to program verification; (2) a computational model which provides a unified framework for comparing both visual and nonvisual algorithms; and (3) strategies for implementing declarative visualization on parallel machines.

1. INTRODUCTION

Visualization is defined as the graphical representation of objects and processes. The importance of visualization as a communication tool has long been acknowledged. In recent years, however, a growing consensus has emerged regarding its potential for promoting the understanding of complex behaviors exhibited by physical phenomena and computations[12]. This paper is concerned with visualization as a means for the understanding, debugging, and monitoring of large-scale concurrent programs, i.e., programs consisting of many thousands of concurrent processes.

The extremely high volume of information produced during the execution of a concurrent program greatly exceeds human abilities to assimilate in textual form. This is in part due to the sequential processing of textual information. The human visual system is more suited to information in the form of images. Humans can process large quantities of image information in parallel, detecting and tracking complex visual patterns with incredible speeds.

Nevertheless, as the number of processes grows, the viewer's ability to understand the resulting image can be rapidly saturated unless the level of abstraction of the information being displayed is increased. For this reason, abstraction plays an important role in visualization. By providing flexible abstraction mechanisms, a visualization system can, in principle, help the programmer select displays which are easily specified and understood. Ease of specification may be achieved by treating visualization as a formally defined abstraction of the program state rather than as a mechanism for producing animated displays. Ease of understanding may be facilitated if the information being displayed relates to formal properties of the program and if the choice of rendering maps the property of interest to a visual pattern which may be detected and tracked with minimal effort.

Treating images as abstractions of the program state is also desirable from a pragmatic perspective. It permits the use of *declarative* specifications which are more concise and easier to employ than the *imperative* strategy which is based on specifying interesting events and associated actions.

As explained in later sections, our work relies heavily on the declarative approach to visualization. Actually, the emphasis on declarative visualization was a major factor in the selection of the underlying model of concurrent computation, the *shared dataspace paradigm*. The shared dataspace is the only concurrency paradigm we are aware of which elegantly accommodates declarative visualization. This is because the paradigm is based on a model in which *processes* use powerful transactions to manipulate a common content-addressable data structure called the *dataspace*. Since all important state information is captured by the dataspace, one can treat visualization as a mapping from the dataspace to a geometric or graphical model which, in turn, may be rendered on some imaging device.

In the broader sense, our search for ways to make effective use of images is a vehicle by which we seek to establish a sound technical foundation for the visualization field. We see *abstraction*, *human perception*, *formal verification*, and *computational complexity* to be the cornerstones of such a foundation, at least as far as visualization of concurrent computations is concerned. Ultimately, the value of this work will be judged by the extent to which it will contribute to:

(1) the identification of image properties that can be detected and tracked visually with a high degree of reliability;
(2) the development of visual representations that capture directly important formal program properties (e.g., safety and progress) and facilitate immediate and reliable visual detection of instances where these properties fail to hold; and
(3) the establishment of computational models that allow the comparison of the effectiveness of alternative visualizations with respect to each other and with respect to nonvisual alternatives.

The purpose of this paper is to report on some of the decisions we have made and the path we are pursuing.

The next two sections of this paper give an overview of the shared dataspace paradigm and the declarative approach to visualization. Section 4 discusses mechanisms for specifying declarative visualization in the shared dataspace paradigm and ways of relating them to program verification. In section 5 we propose a computational model which provides a useful unified framework for comparing both visual and nonvisual algorithms and for formally incorporating elements of visual perception. Section 6 is concerned with strategies for implementing declarative visualization on parallel machines and includes a brief status report on our current implementation efforts on a 64-node multiprocessor.

2. SHARED DATASPACE

Concurrent programming languages may be divided into four broad categories depending on the nature of the interprocess communication they employ. *Shared variables* as used in Concurrent Pascal[5], for example, allow processes to communicate by reading and writing (directly or indirectly via monitors) the values of a fixed set of variables whose names are known to the communicating processes. *Message based communication* requires all information sharing to take place by sending and receiving messages in accordance with some predefined protocol. CSP[13] and Actor languages[3] are representative of this category. *Remote operations*, such as the remote procedure call used in Ada[1], permit a process to invoke operations associated with some other process. The parameters and returned values represent the information shared by the processes involved in the exchange. The last category we choose to call *shared dataspace*. It is comprised of languages in which processes have access to a common, content-addressable data structure (typically a set of tuples) whose components may be asserted, read, and retracted. Associons[18], Linda[4], and some artificial intelligence languages such as OPS5[8] belong here.

Our first attempt to study the relation between the shared dataspace paradigm and large-scale concurrency relied on the use of a language called SDL[20,21]. In this paper, we explain the shared dataspace paradigm by means of a much simpler language called *Swarm*[19]. In Swarm, concurrent computations are defined in terms of a *dataspace*, which is a finite set of *tuples*, and a *transaction space*, which is a finite set of *transactions*. Each transaction represents an atomic transformation of the data and transaction spaces. An individual transaction may examine, assert, and retract tuples in the dataspace and it may examine and assert (but not retract!) transactions in the transaction space. Transaction execution results in the removal of the transaction from the transaction space. Transactions are selected from the transaction space in a nondeterministic order and executed; the selection is fair in the sense that all transactions in the transaction space are eventually executed.

A Swarm program‡ consists of the initial configurations for the data and transaction spaces and a finite set of transaction type definitions. A transaction type definition defines a name for the transaction type and a set of parameters bound at the creation of the transaction. The body of the transaction (enclosed in ⟦ ⟧) consists of a number of subtransactions separated by the symbol ‖. Each subtransaction consists of a *query part* (a predicate in first-order logic) which interrogates the database for tuples, and an *action part* which defines tuples to be asserted and retracted and transactions to be asserted. The scope of existentially bound variables from the query part includes the action part. The query and action parts are separated by the symbol →.

During execution of a transaction, the query parts of all subtransactions are evaluated and the action parts associated with the successful queries are performed. Some built-in queries, such as *NOR*, may depend on the success or failure of other query parts; for instance, the query *NOR* succeeds only if all other query parts in the transaction fail. It is worth noting that, even when the actions of the subtransactions are not disjoint, the language guarantees noninterference between subtransactions by executing all retractions associated with a transaction before any of the assertions.

‡ The syntax used in this paper is for illustrative purposes only, and does not necessarily represent the actual syntax of Swarm.

INITIAL DATASPACE:

{ ρ : ρ∈ Pixels : (ρ,Intensity(ρ),is_labeled,ρ) }
where
Pixels={x,y: 1≤x≤N, 1≤y≤M : (x,y)}
Intensity : Pixels → IntensityRange

INITIAL TRANSACTION SPACE:

{ ρ : ρ∈ Pixels : Label(ρ) }

TRANSACTION TYPES:

```
Label(P) ::
  ⟦   ∃ ρ,ι,λ1,λ2 : [P,ι,is_labeled,λ1]†,
        ρ four_neighbors P, [ρ,ι,is_labeled,λ2], λ2>λ1
          → (P,ι,is_labeled,λ2), Label(P)
    ‖ NOR → Label(P)
  ⟧
```

Figure 1: Region Labeling in Swarm

The notation [*pattern*] represents a membership test against the dataspace. For example, ∃ λ : [**pair**, 17, λ] is a test for a dataspace tuple having three elements, the first of which is the atom **pair**, the second the constant 17, and the last an arbitrary value. If such a tuple exists, the query succeeds and the variable λ is bound to the value in the tuple for the remainder of the subtransaction. The notation [*pattern*]† indicates that the tuple matching the pattern is to be removed from the dataspace; when this notation appears in the query part, the removal occurs only if the entire query succeeds. The similar notation *tname*[*pattern*] represents a membership test against the transaction space for a transaction having the type *tname*. Some relatively complex queries can be written in just a few lines in Swarm. For example, the following query counts the number of tuples matching a particular pattern by summing a 1 for each such tuple:

(Σ *variables* : [*pattern*] : 1)

Nesting of such queries within other queries is permitted.

In the action part, the notation (*tuple*) indicates the tuple is to be added to the dataspace. The notation *tname*(*values*) indicates a transaction of type *tname* is to be asserted in the transaction space, with the transaction type parameters bound to the given values.

Figure 1 shows a simple version of a region-labeling program in Swarm. Each tuple in the dataspace represents a pixel in a N by M image and consists of the pixel position ρ, the pixel intensity, and a label. Initially, each pixel is labeled by its own pixel position. Associated with each pixel in the dataspace there is a transaction of type *Label* in the transaction space.

A *Label* transaction associated with the pixel *P* consists of two subtransactions. The first one checks to see if there exists a four-connected neighbor ρ of the pixel at *P* which has

the same intensity ι but whose label λ2 is greater than the label λ1 currently associated with *P*. If such a neighbor is found, the pixel *P* is relabeled using λ2 by deleting the tuple (*P*,ι,**is_labeled**,λ1) and asserting the tuple (*P*,ι,**is_labeled**,λ2) and the transaction Label(*P*). The role of the second subtransaction is to recreate the transaction Label(*P*) when no such neighbor is found.

The result of executing this program is that, at some point in the computation, all pixels in a connected region of uniform intensity are assigned the same unique label. However, no attempt is made to detect this condition and consequently the program fails to terminate (although it does reach a fixed-point state in which neither the dataspace nor the transaction space change).

3. DECLARATIVE VISUALIZATION

Before discussing visualization in the shared dataspace paradigm we need to provide the reader with a better understanding of what is meant by declarative visualization. The background for this discussion is the evolution of methods for program visualization (or algorithm animation, as it is often called). We explicitly exclude from consideration visual programming, an area whose primary concern is the development and use of non-textual languages.

Originally, program visualization research was motivated by the desire to explain, by means of animated displays, the workings of sequential algorithms. Among the visualization systems that are a product of this perspective, Balsa[6,7] is probably the best known and the most influential. Balsa is also representative for what we choose to call the *imperative* approach to visualization. Constructing an animation requires augmenting the algorithm with calls to a series of library routines designed to interface with the mechanics of display generation. The subroutine calls must be placed in such a way as to capture significant program events and to trigger needed changes in the display. Despite the high degree of modularity built into the approach, the need to include the calls to the visualization routines within the program adds complexity and lacks flexibility.

Flexibility becomes particularly important when the purpose of visualization is to help the programmer understand and debug algorithms. The programmer needs the freedom to change both the information being displayed and the way in which it is displayed. Moreover, changes to the executing code should be avoided whenever possible.

These and other considerations led to the emergence of *declarative* approaches to visualization. The declarative approach sees visualization as a relation between the program state and the state of the graphical objects depicting it. Flexibility is derived from the ease with which this relation can be changed.

In PROVIDE[16], for instance, program variables are bound to formal parameters associated with visual objects. Changes in program variables are automatically translated into updates of the display. This type of declarative visualization could be called *event binding*. In our opinion, event binding will prove inadequate for large-scale concurrent programs because of its low level of abstraction.

The declarative approach advocated in this paper is based on the more abstract notion of *state mapping*. Program visualization is defined as a function from the program state to the state of one or more graphical objects. Both the level of abstraction and the choice of graphical objects may be controlled by changes in the program state; i.e., as the program state changes the representation may also change. The resulting increases in flexibility and the ease of specification, however, do not come for free. Even when the program state can be captured cleanly, there is still the burden of ensuring that visualizations are computed rapidly enough to maintain the continuous display we expect from visualization systems.

As shown in later sections, the shared dataspace paradigm is particularly well suited for the state mapping approach. The program state may be formally defined as a set of tuples, thus eliminating the need to consider the program text and the programming language syntax and semantics. Mathematically speaking, transactions (which are the basic computational entities in the shared dataspace paradigm) are relations defined over the combined data and transaction spaces. As such, the same mechanisms that implement transaction execution are useful in implementing the visualization system. Finally, the computational power needed to maintain the continuous displays is readily available on the highly parallel machines for which shared dataspace languages are targeted.

4. VISUAL ABSTRACTION

Declarative visualization in shared dataspace languages is facilitied by the ease with which one can specify properties of the program state. The current program state is simply given by two sets of tuples: the transaction space *T* and the dataspace *D*. Properties of the program are easily defined in terms of predicates in first-order logic, i.e., the very same formalism used to define the query part of the transactions. In the region-labeling program, for instance, to determine if there is a point in the image having more than one label we can simply write

$$\exists\ \rho,\iota,\lambda1,\lambda2:$$
$$[\rho,\iota,\textbf{is\_labeled},\lambda1],\ [\rho,\iota,\textbf{is\_labeled},\lambda2],\ \lambda1\neq\lambda2$$

The reader may want to think of how one might ask the same question about a program written in a traditional language such as Pascal!

Because graphical objects can also be specified using the tuple notation, we introduce a new set of tuples *O*, called the *object space*, which contains the set of graphical objects to be rendered at any point during the computation, and a visualization mapping *V* which computes *O* in terms of *T* and *D*. In addition, a (device-specific) *rendering function* translates the contents of *O* into an image.

We specify the visualization mapping *V* by a set of rules having the form

variables : query over T and D ⇒ list of tuples in O

where the variables are existentially quantified implicitly. Such a rule defines a (constantly changing) set of tuples as follows. The query is evaluated, and for *each* successful match the variable bindings are used to instantiate the tuple list on the right hand side; all the resulting tuples are members of the set. For any state of the computation, the object space *O* is equal to the union of the sets produced by each visualization rule. Thus, if a new tuple is asserted which matches with some visualization rule, all resulting tuples are immediately added to *O*; likewise, if a tuple is retracted, any

members of O generated by a rule matching the tuple are immediately removed. (It should perhaps be emphasized that this is a *model* of our approach to visualization, and that an implementation would not necessarily compute the image in this fashion.)

For the sake of simplicity let us assume that the universe of graphical objects is limited to points of various colors, i.e., to tuples of the following form:

(**point**, *coordinate, color*)

where **point** is a constant literal, *coordinate* is a pair ⟨x,y⟩, and *color* is a (device-dependent) representation of a particular color.

The rendering function translates a particular state of the object space into a screen image. For the region-labeling program with N by M pixels, we will assume a rendering function with the following properties:

- The screen consists of a rectangular array of at least 4N by 4M points.
- A function *colorize* is available which 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.*
- An object tuple (**point**,⟨2*i*,2*j*⟩,*c*) is mapped to a 3×3 region of screen points having color *c* at screen coordinate ⟨4*i*,4*j*⟩.
- An object tuple (**point**,⟨2*i*+1,2*j*⟩,*c*) is mapped to a 1×3 region at screen coordinate ⟨4*i*+3,4*j*⟩.
- An object tuple (**point**,⟨2*i*,2*j*+1⟩,*c*) is mapped to a 3×1 region at screen coordinate ⟨4*i*,4*j*+3⟩.
- An object tuple (**point**,⟨2*i*+1,2*j*+1⟩,*c*) is mapped to a 1×1 region at screen coordinate ⟨4*i*+3,4*j*+3⟩.
- If no object tuple is mapped to some screen point, the point is a recognizable color called the *background color.*
- If two or more object tuples map to some screen point, the point is a recognizable color called the *overlap color* and is flashed in the image.

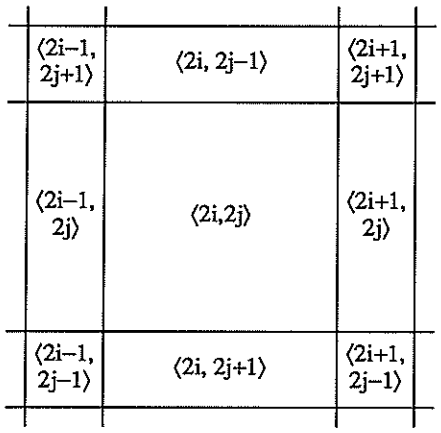

Figure 2: Rendering Function Grid

The coordinates are those of the **point** tuple which is mapped to the group of screen points.

The reason for the unusual mapping of point coordinates to screen coordinates will be clarified below. The result of the mapping is a grid as illustrated in Figure 2.

Considering the region-labeling program again, one can map pixels (program objects) to points (graphical objects) having a color determined by the current label:

V0: $\rho,\iota,\lambda : [\rho,\iota,\text{is\_labeled},\lambda] \Rightarrow (\text{point},2\rho,\text{colorize}(\lambda))$

Note that a pixel at ⟨*x*,*y*⟩ is mapped to a point at ⟨2*x*,2*y*⟩, which is depicted by the rendering function as one of the 3×3 screen squares. This visualization is illustrated in Figure 3.

At first glance, this rather simple visualization may appear to be all that one needs to monitor and understand the program's behavior. Unfortunately, although the entire computational state is captured fully, this visualization is flawed in two fundamental ways:

(1) the programmer cannot reliably detect algorithmic faults from the display; and,
(2) the display captures the low-level mechanics of the program execution rather than fundamental program properties used by the programmer to reason about the particular computation.

These are precisely the kinds of shortcomings a more formal treatment of visualization is meant to overcome.

In the remainder of this section we outline a strategy for approaching the visualizations of concurrent computations and apply it to region labeling. The starting point for our approach is the formulation of safety and progress properties of the program in a manner similar to the work of Chandy and Misra[10]. We chose program verification as the foundation for our strategy because the complexity of concurrent computations defeats any kind of operational reasoning about programs—it is impossible to keep track of all conceivable interleavings of events.

Given a particular safety property (typically a program invariant) we seek to render it as a stable visual pattern in such a way that any change, representing a violation of the invariant, would be readily observable. Progress properties are captured by ordered pairs of patterns where a state which generates the first pattern is to lead to a state which generates the second. Lack of progress is represented by a failure to reach the second pattern. Unfortunately, not all manifestations of lack of progress are easy to detect. Total lack of activity or cyclic patterns are among the readily observable progress failures.

The principal invariants and progress conditions used in the correctness proof for the region-labeling program (not presented here) are:

I1: Region boundaries do not change.
I2: Two neighboring pixels belonging to two different regions never have the same label.
I3: In any given region, the pixel having the highest coordinate is labeled by its own coordinate.
P1: If a pixel ρ has a neighbor belonging to the same region and labeled by the highest coordinate in that region, ρ will eventually be labeled by the highest coordinate in that region.

Given these formal properties we can now turn to the issue of selecting an appropriate rendering of the program. It is our intent to convince the reader that these properties provide concrete directives toward the kind of visual patterns we seek.

As earlier stated, invariants are rendered as stable patterns such that violations of the invariant are easily observed.

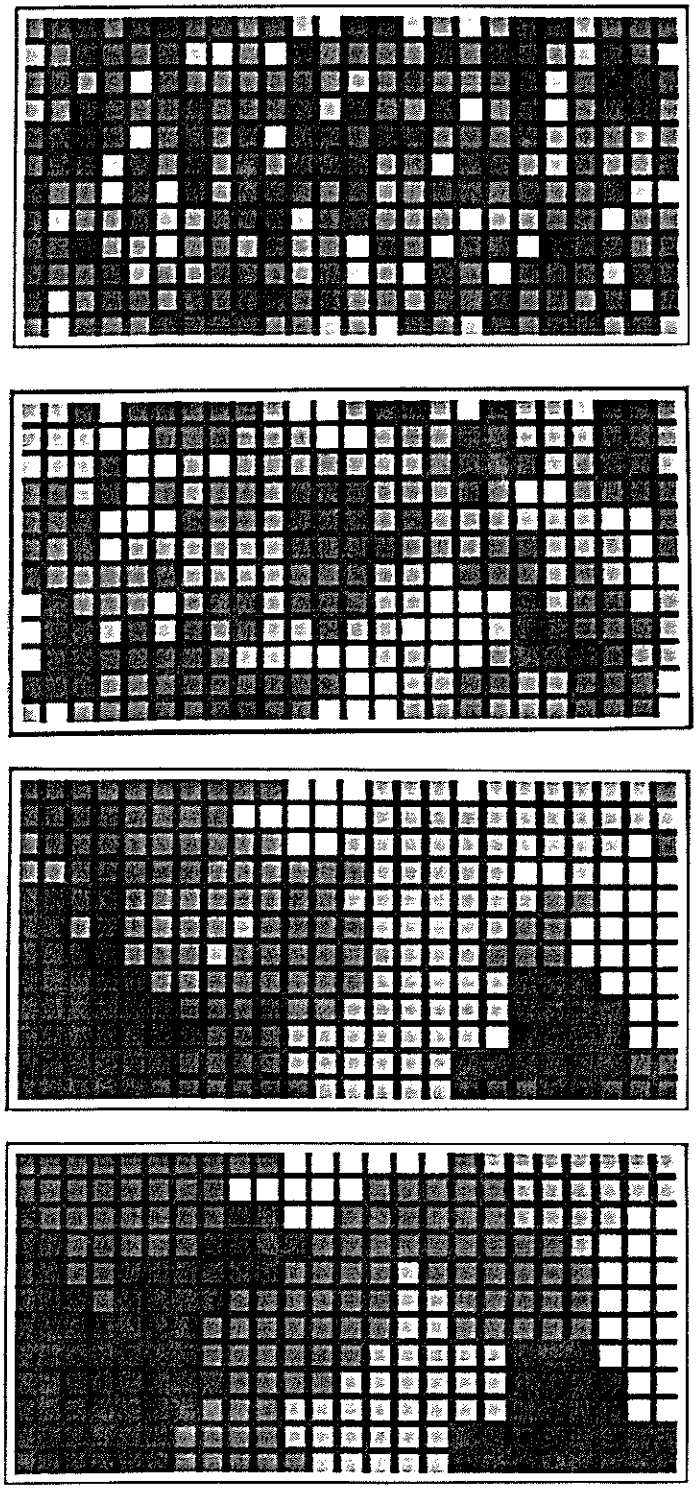

Figure 3: Sample Visualization of Region Labeling

The visualization rule *V0* is in effect.
Time increases down the page.

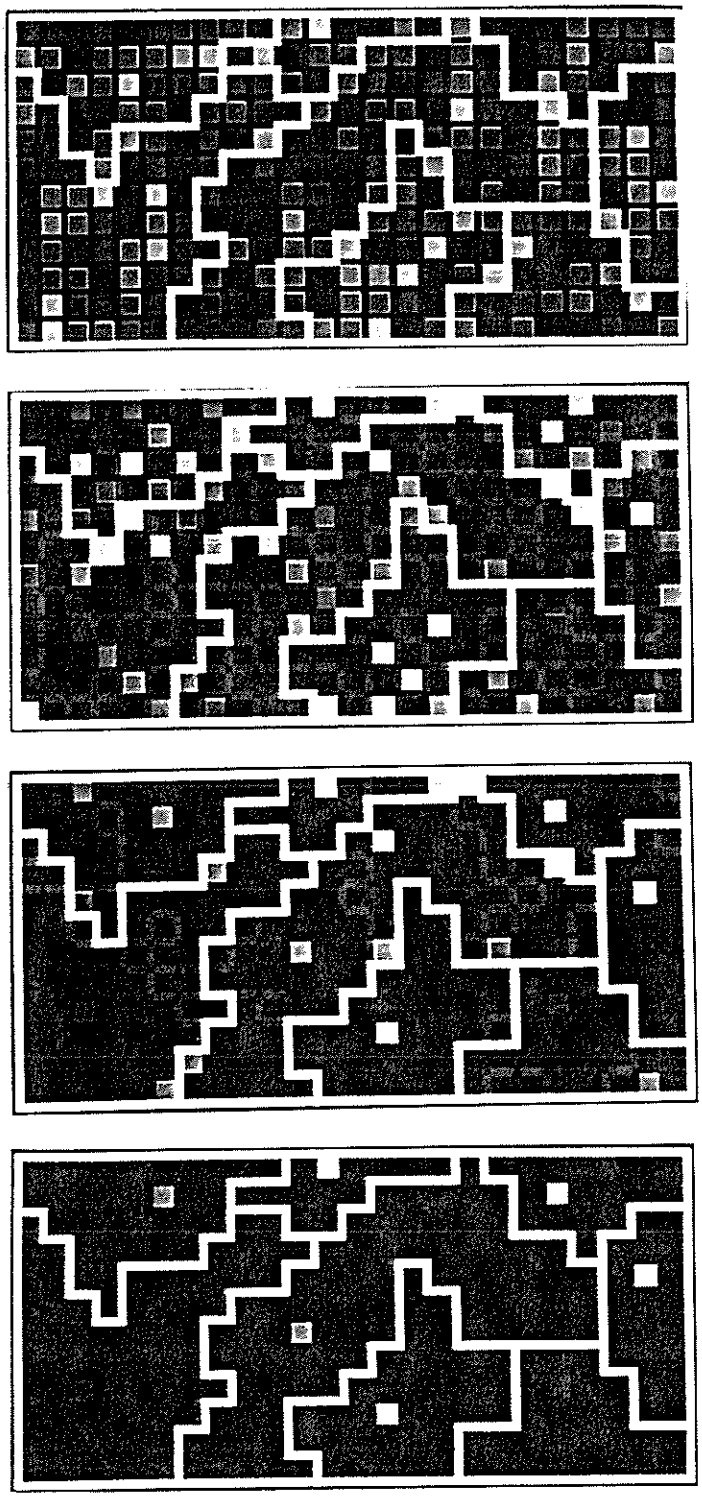

Figure 4: Sample Visualization of Region Labeling

The visualization rules *V1* through *V4* are in effect.
Time increases down the page.

However, *I1*, *I2*, and *I3* have very distinct logical forms — *I1* is universally qualified, *I2* involves a negation, and *I3* includes an embedded existential quantification. Because of this, one should expect to see distinct visualization strategies applied to each.

The formulation of *I1* strongly suggests the need to render the borders between pixels of differing intensities in some recognizable color, for example white. If *I1* is violated these borders will move or disappear during execution. Furthermore, examining the other invariants and progress conditions we note that *P1* states that progress is made by labeling all pixels in the same region with the same label. This suggests that a representation of the region boundaries would also be useful in visualizing *P1*.

To render the borders we use the gaps between the 3×3 regions; by using the eight_neighbors relationship, the small 1×1 screen areas are filled as well as the 1×3 and 3×1 areas. The rule which visualizes *I1* is:

V1: ρ1,ι1,λ1,ρ2,ι2,λ2 :
[ρ1,ι1,**is_labeled**,λ1], [ρ2,ι2,**is_labeled**,λ2],
ρ1 eight_neighbors ρ2, ι1≠ι2
⇒ (**point**,ρ1+ρ2,white)

In the case of *I2*, because the invariant deals with the absence of a condition no particular visual representation seems to be required. Violations, however, must be detected; one way to do this is by rendering the gap between any two neighboring pixels which have different intensities but the same labels in the overlap color. This rendering is accomplished by the rule:

V2: ρ1,ι1,ρ2,ι2,λ :
[ρ1,ι1,**is_labeled**,λ], [ρ2,ι2,**is_labeled**,λ],
ρ1 four_neighbors ρ2, ι1≠ι2
⇒ (**point**,ρ1+ρ2,overlap)

Visualizing *I3* seems to cause difficulties, because the invariant refers to a pixel which, although known to exist and to be unique for each region, must be precomputed. A more careful analysis of this invariant allows us to produce a visual representation of this invariant without precomputation by introducing some initial uncertainty that is gradually reduced. The idea is to color all pixels which 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 largest coordinate in the region colored. The following rule can be used for this:

V3: ρ,ι : [ρ,ι,**is_labeled**,ρ]
⇒ (**point**,2ρ,colorize(ρ))

The progress condition *P1* can be visualized by marking the boundaries of areas which have the same labeling with some recognizable color (e.g., red):

V4: ρ1,ι,λ1,ρ2,λ2 :
[ρ1,ι,**is_labeled**,λ1], [ρ2,ι,**is_labeled**,λ2],
ρ1 four_neighbors ρ2, λ1≠λ2
⇒ (point,ρ1+ρ2,red)

The previous visualization rule *V1* outlines the boundaries of the regions in white, while *V3* marks the pixel having the highest label in each region. Progress is recognized by the expansion of the areas within red boundaries toward the white boundaries, and completion is recognized by the disappearance of all red boundaries.

Figure 4 is a sample visualization of the region-labeling program in which rules *V1* through *V4* are in effect. The overlap color is bright yellow; naturally, since the program and this implementation are correct, it does not appear. The two program runs in Figures 3 and 4 are identical, and corresponding photographs were taken after equal amounts of progress; only the visualization differs.

This simple example demonstrates the importance of using formal program properties as the basis for deciding which visualization rules are appropriate. It is our 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 predicates used in the program correctness proof and not on knowledge of the operational details of the program. Such an approach would facilitate true exploration of the program rather than its mere animation.

5. COMPLEXITY MODEL

This section describes preliminary work on the development of a complexity model useful in comparing the efficiency of alternate visualizations against each other and against nonvisual algorithms solving the same problem. We limit our concern to yes/no type questions and to visual representations consisting of colored points in 2D-space.

A question is simply a predicate over the program state:

q : ProgramStates → {true, false}

Given a concurrent shared dataspace program P and a question q, we say P answers q if execution of P starting in some initial state causes a tuple (**answer**,q(initial-state)) to be asserted in the dataspace after some finite time and not thereafter retracted. Note that because the execution mechanism for shared dataspace programs is nondeterministic, there may be many ways in which P can be executed; however, each must cause the assertion of the answer tuple in finite time.

For any transaction *execution* τ, we define Operands(τ) as the set of tuples and transactions asserted or retracted by the execution of τ. Two transaction executions $\tau 1$ and $\tau 2$ are conflict-free iff

$$\text{Operands}(\tau 1) \cap \text{Operands}(\tau 2) = \varnothing$$

A set of transaction executions is called conflict-free if all pairs of distinct elements of the set are conflict-free.

We define a computation as a sequence of state transitions, each caused by execution of a set of conflict-free transactions. With each such transition we associate a weight representing the cost of performing the set of transactions. The cost of a particular computation is defined as the sum of the weights of the transitions from the initial state to the first state in which the tuple (**answer**,q(initial-state)) appears in D.

Two transaction sets are called *reducible* if their union is conflict-free. We define a *maximally concurrent* computation as one in which the transaction sets associated with each consecutive pair of transitions are not reducible. The cost of a nonvisual algorithm for a particular initial state is defined as the minimum of the costs of all maximally concurrent computations. The complexity of the algorithm is defined as the maximum computation cost over all initial states. (This definition of algorithm complexity as the maximum cost over states, where the cost for a state is the minimum time required to find the result, is based on similar work with

nondeterministic parallel Turing machines and random access machines[11,14].)

In the case of visual algorithms, the question is solved whenever the correct answer may be deduced reliably from the visual representation of the program state. Any state whose visualization allows one to deduce reliably the answer is called a *revealing-state.* Revealing-states replace answer-states in determining the algorithm efficiency.

Region labeling again provides the basis for a simple illustration of the ideas, if we take the liberty to assume that some mechanism which detects the completion of the labeling activities has been added to the program and that once the labeling is completed the tuple *(done)* is added to the dataspace. The question we consider here is: *Do pixels P1 and P2 belong to the same region?* One nonvisual solution can be constructed by adding to the initial transaction space the transaction described in Figure 5.

The efficiency of this solution is determined by the cost of detecting termination. If the total number of pixels is $M \times N$, typical solutions exhibit a complexity $O(M \times N)$ (with the worst case exhibited by regions with a spiral shape). If we consider now the visual approach and assume the visualization rules used in the earlier section, it turns out that the initial state of the program is already a revealing-state. Because the region boundaries are present in the rendering the question reduces to the visual task of telling if two points are inside of the same contour. Hence, using the definitions introduced so far, the program efficiency is $O(1)$.

It is obvious from the example that there are some hidden costs not accounted for in the definition of efficiency: the computation of the visualization V and the human effort associated with performing the visual task. V was defined as a mapping from the program state to the object space

V : ProgramStates → ObjectSpaceStates

followed by application of the rendering function to produce a screen image.

The cost of computing V is determined by the nature of the visualization rules. Since the implementation of visualization rules is expected to employ mechanisms similar to transaction execution, the cost attributed to V may be computed in the same manner as the cost associated with the program execution. In our example this cost is actually a constant (per pixel) because each pixel contributes to at most nine points in the object space (its own image and those of the bordering points). Provided the rendering function is relatively simple (i.e., a linear mapping from object space to screen, with each object producing no more than a constant number of screen points) the cost of applying the rendering function may be incorporated into that of computing V.

Since different visual tasks require different levels of effort on the part of the human observer, one could consider associating a cost function with the performance of the task. For a given object space (and associated rendering function) one might be able to determine the cost function experimentally. We question, however, the usefulness of such an undertaking. We prefer to view any visual task whose complexity exceeds some threshold (e.g., 1 second) to be too tedious to be useful.

This argument also applies to a second observation. For a visual algorithm, we have defined a *revealing-state* as one from which the correct answer may be deduced reliably. However, it may be the case that the answer can be deduced reliably but the complexity of the task exceeds the threshold discussed above. Returning to the region-labeling example, a particular image may be divided into a number of closely intertwined regions. Although the initial state (containing the region boundaries) contains sufficient information to determine whether two points lie in the same region, actually determining this may be as difficult as solving a maze. Thus, although the image does represent a revealing-state, the cost of answering the question is too great.

```
SameRegion(P1,P2) ::
  [[   ∃ ι,λ :
         [P1,ι,is_labeled,λ], [P2,ι,is_labeled,λ]
           → (true)
     || ∃ ι1,ι2,λ1,λ2 : ι1≠ι2,
         [P1,ι1,is_labeled,λ1], [P2,ι2,is_labeled,λ2]
           → (false)
     || ∃ ι,λ1,λ2 : [done], λ1≠λ2,
         [P1,ι,is_labeled,λ1], [P2,ι,is_labeled,λ2]
           → (false)
     || NOR → SameRegion(P1,P2)
  ]]
```

Figure 5: Non-Visual Answer Computation

The ability to perform the visual task required to determine the answer to some question given a particular image is captured by a function called an *oracle*:

oracle : ObjectSpaceStates → {**can't-tell, true, false**}

If the object space state is too complex to allow determination of the answer, the value of the oracle is **can't-tell** and the associated oracle cost is infinite. On the other hand, if the answer can be easily deduced the value of the oracle is either **true** or **false** and the cost is a small constant‡. The issue of how to capture our current knowledge of visual cognition and formalize it in terms of oracles is an important open question but the growing body of information[17] being accumulated in this area gives us strong reasons for optimism.

6. IMPLEMENTABILITY

In this section we consider the implementability of the state mapping approach to declarative visualization, i.e., the algorithms that allow us to compute the visualization mapping V. State mapping belongs to the broad class of problems which address the issue of global state detection and monitoring.

Distributed snapshots and instant replay are two examples of problems belonging to the same class. The distributed snapshots problem[9] involves the detection of stable properties of a distributed computation. Given a set of processes which communicate asynchronously by sending and receiving

‡ Defining the cost function in this way leads to what might be called the "Eureka effect": A visualization which starts in a state having an oracle value of **can't-tell** and ends in one having a value of **true** or **false** must have a single transition in which the cost of the oracle drops from infinite to a small constant. Before this moment, the user cannot easily determine the answer from the image; after this moment, he can.

messages, a global state is computed cooperatively without interfering with the underlying computation. The computed state is not necessarily one through which the computation passed, but it is guaranteed that the computation *could* have passed through the state between the initiation and the completion of the snapshot. If the computed state has the desired property, stability guarantees that the property is true for any current and future global state of the system; in particular, the property will hold for the actual system state at the time of the snapshot's completion. The instant replay problem concerns the reproducibility of the execution behavior of concurrent programs. This is logically equivalent to forcing the system to go through the same sequence of global states as it did in some previous computation. One solution to this problem[15] assumes that all interprocess interactions take place via shared objects and uses version numbers associated with each object to record the order of all significant events occurring in the system.

In this section we formulate a new global state detection and monitoring problem which addresses the computation of the visualization mapping V and show one solution for this new problem. The problem is called *global state tracking*. The motivation for this problem is the need to extract from the dataspace the current state of any tuples matching the left hand side of the visualization rules, and to use these tuples to update incrementally the object space and hence the display. Our statement of the problem is independent of the specifics of declarative visualization.

In general terms, global state tracking is defined as the problem of maintaining at a centralized location a continuously updated representation of the global state of a concurrent computation. We will refer to the actual global states of the computation as the *distributed states* and to their representations as the *centralized states*. Because the central location does not have direct access to the state of the concurrent computation, it can learn about it only from messages received from processes participating in the computation. Consequently, the centralized state is always behind the distributed state and some of the constructed centralized states may not have actually existed in the actual concurrent computation. This is acceptable as long as for a given computation we require consistency between the sequence of centralized states and the sequence of distributed states *and* we require a fixed upper bound on the time it takes for a change in the distributed state to be manifested in some centralized state. The two sequences of states are called *consistent* if dependent changes to the system state appear in the same order in both sequences—in database terminology, this definition reduces to the equivalence of two serial schedules. Examples of dependencies that must be considered are read-write dependencies in database-type systems and get-send dependencies in systems communicating via messages.

The solution we present assumes that the underlying computation is a shared dataspace program. Furthermore, to satisfy the fixed upper bound constraint we must make several important assumptions. First, we assume that both the time it takes for a message to arrive at the central location and the size of the transaction space have upper bounds related to the problem size. Second, we assume that the central location has sufficient computing power and input communication bandwidth to accommodate the receipt and processing of all the messages sent by the processes participating in the concurrent computation.

The key to our algorithm is the ability of the central location to determine the order in which dependent dataspace transformations (i.e., transactions) actually occurred. For this reason we associate with each transaction an unique identifier which is used to tag tuples and transactions with the identity of their creators. The essence of the algorithms may be described as follows:

When a transaction τ having identifier ι is executed:

(1) τ locks all the tuples (in the data and transaction spaces) it needs to examine or retract and computes a dependency set $D(\iota)$ containing the identities of all the creators of the locked tuples.
(2) τ performs all necessary retractions and assertions making sure to attach its identity ι to all the tuples it asserts (in the data and transaction spaces) and releases all locks.
(3) For every tuple retracted or asserted, τ sends to the central location a message consisting of its identity ι, the number of tuples retracted and asserted, and the dependency set $D(\iota)$. (The reasons for sending separate messages are to avoid the need for extra communication required to bring all the tuples together prior to sending the update information to the central location, and to allow the possibility that agents other than the transaction might assume the task of generating these messages.)

At the central location:

(1) A message is processed when all messages from the same transaction have arrived and all messages sent by any transaction appearing in its dependency set have been processed.
(2) All messages associated with the same transaction are processed together as an atomic update of the centralized state.

The solution outlined above can be adapted to other underlying computational models far different from the shared dataspace paradigm and is practical whenever the number of tuples represents a small subset of the dataspace. Unfortunately, the algorithm does not take into account the right hand side of the visualization rule and makes no attempt to distribute the computation of object space. We are currently seeking to refine the formulation of the global state tracking problem and to develop fully distributed algorithms that solve it.

Our current implementation efforts are directed to the development of a prototype transaction system and associated visualization capability on an NCUBE-7 equipped with a Real-Time Graphics Board[2]. The NCUBE-7 is a highly-parallel hypercube-configured MIMD machine which uses a message-passing protocol for interprocessor communication. Our NCUBE is configured with 64 processors, each equipped with 512 Kbytes of RAM. The Real-Time Graphics Board is a dedicated image processing device with storage for 2048×1024 pixels of 8-bit data. The data can be modified at the feature level (i.e., lines, rectangles, circles) by two Hitachi ACRTC Graphics Controllers or at the pixel level by a group of 16 processors identical to those in the main hypercube. The 16 graphics processors can communicate with the main hypercube at a combined rate of 90 Mbytes/second and can also send commands to the ACRTC processors.

We are implementing a transaction system capable of supporting the shared dataspace paradigm to run on the NCUBE's main hypercube. The design dedicates some of the hypercube processors as *database nodes* which maintain a distributed shared dataspace of tuples. The remaining processors are available for user applications which communicate with the database nodes to assert, query, modify, and retract tuples. The transaction system provides full support for tuples containing arbitrary data, matching of patterns against the stored tuples, and searching of the distributed database.

The 16 graphics processors are used to provide the visualization capability. These processors initiate a global state tracking algorithm (which in the current design will be performed cooperatively by the database, user, and graphics processors). Continuously updated state information is transmitted to the graphics processors, which maintain a centralized state as described above. The visualization rules are then applied by the graphics processors to the centralized state and the visualization image generated.

7. CONCLUSIONS

Central to the ideas presented in this paper is the notion that visualization can play a key role in the exploration of concurrent computations. Hidden between the lines, however, is the concern that the full potential of visualization may not be reached if the art of generating beautiful pictures is not rooted in a solid, formal technical foundation. This paper shows that many of the key concepts on which to build such a foundation already exist and outlines an approach that can bring them together into a unified framework. Although most of the work needed to make the framework a practical reality is yet to be done, the results to date are highly encouraging.

Acknowledgments: This work was supported by the Department of Computer Science, Washington University, Saint 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 paper and his contribution to the development of the shared dataspace concept. The NCUBE-7 is one of the computational resources maintained by the Computer and Communications Research Center of Washington University.

8. REFERENCES

1. "Reference Manual for the Ada Programming Language," ANSI/MIL-STD-1815A-1983, American National Standards Institute, Inc., Washington, D.C. (January 1983).
2. "NCUBE Users Handbook," Version P2.1, NCUBE Corporation, Beaverton, Oregon (October 1987).
3. Agha, G., *Actors: A Model of Concurrent Computation in Distributed Systems,* MIT Press, Cambridge, Massachusetts (1986).
4. Ahuja, S., Carriero, N., and Gelernter, D., "Linda and Friends," *Computer* 19(8), pp. 26-34 (August 1986).
5. BrinchHansen, P., "The Programming Language Concurrent Pascal," *IEEE Transactions on Software Engineering* 1(2), pp. 199-206 (1975).
6. Brown, M. H. and Sedgewick, R., "A System for Algorithm Animation," *Computer Graphics* 18(3), pp. 177-186 (July 1984).
7. Brown, M. H., "Exploring Algorithms Using Balsa-II," *Computer* 21(5), pp. 14-36 (May 1988).
8. Brownston, L., Farrell, R., Kant, E., and Martin, N., *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming,* Addison-Wesley, Reading, Massachusetts (1985).
9. Chandy, M. and Lamport, L., "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM Transactions on Computer Systems* 3(1), pp. 63-75 (February 1985).
10. Chandy, M. and Misra, J., *Parallel Program Design—A Foundation,* Addison-Wesley (1988).
11. Fortune, Steven and Wyllie, James, "Parallelism in Random Access Machines," pp. 114-118 in *Proceedings of the 10th ACM Symposium on the Theory of Computing*, ACM, San Diego, CA (May 1987).
12. Frenkel, K. A., "The Art and Science of Visualizing Data," *Communications of the ACM* 31(2), pp. 110-121 (February 1988).
13. Hoare, C. A. R., "Communicating Sequential Processes," *Communications of the ACM* 21(8), pp. 666-677 (August 1978).
14. Kozen, D., "On Parallelism in Turing Machines," pp. 89-97 in *Proceedings of the 17th Annual IEEE Symposium on Foundations of Computer Science*, IEEE, Houston, TX (October 1976).
15. LeBlanc, T. J. and Mellor-Crummey, J. M., "Debugging Parallel Programs with Instant Replay," Technical Report TR 194, The University of Rochester, Computer Science Department, Rochester, New York (September 1986).
16. Moher, T. G., "PROVIDE: A Process Visualization and Debugging Environment," *IEEE Transactions on Software Engineering* 14(4), pp. 849-857 (June 1988).
17. Pinker, S. (editor), *Visual Cognition,* MIT Press (1984).
18. Rem, M., "Associons: A Program Notation with Tuples Instead of Variables," *ACM Transactions on Programming Languages and Systems* 3(3), pp. 251-262 (July 1981).
19. Roman, G.-C. and Cunningham, H. C., *A Shared Dataspace Model of Concurrency — Language and Programming Implications.* In preparation.
20. Roman, G.-C., Cunningham, H. C., and Ehlers, M. E., "A Shared Dataspace Language Supporting Large-Scale Concurrency," *Proceedings of the 8th International Conference on Distributed Computing Systems*, pp. 265-272. IEEE, June 1988.
21. Roman, G.-C., "Language and Visualization Support for Large-Scale Concurrency," *Proceedings of the 10th International Conference on Software Engineering*, pp. 296-308. IEEE, April 1988.