Report Number: WUCS-90-31

1990-09-01

Visualizing Concurrent Computations

Authors: Kenneth C. Cox and Gruia-Catalin Cox

This paper describes the underlying model for a visualization environment concerned with exploring, monitoring, and presenting concurrent computations. The model is declarative in the sense that visualization is treated as the composition of several mappings which, as a whole, map computational states into full-color images of a 3-D geometric world. The mappings defining the visualizations are specified using a rule-based notation. The visualization methodology is proof-based, i.e., it captures abstract formal properties of programs (e.g. safety and progress) rather than operational details. An algorithm for termination detection in diffusing computations is used to illustrate the specification method and to demonstrate its conceptual elegance and flexibility.

VISUALIZING CONCURRENT COMPUTATIONS

Kenneth C. Cox
Gruia-Catalin Roman

WUCS-90-31

September 1990

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

Abstract

This paper describes the underlying model for a visualization environment concerned with exploring, monitoring, and presenting concurrent computations. The model is declarative in the sense that visualization is treated as the composition of several mappings which, as a whole, map computational states into full-color images of a 3-D geometric world. The mappings defining the visualizations are specified using a rule-based notation. The visualization methodology is proof-based, i.e., it captures abstract formal properties of programs (e.g., safety and progress) rather than operational details. An algorithm for termination detection in diffusing computations is used to illustrate the specification method and to demonstrate its conceptual elegance and flexibility.

Keywords: visualization, concurrency, rule-based specification.

Correspondence: All communications regarding this paper should be addressed to

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

(314) 889-6190
roman@CS.WUSTL.edu
fax: (314) 726-7302

1. Introduction

During the past decade visualization has become an indispensable scientific tool. In medicine, physics, meteorology, engineering, and elsewhere researchers use images to cope with large and dynamic data volumes gathered through observations or generated by simulations. A single image is able to convey vast amounts of information directly tied to the physical phenomenon under consideration. Seeing a simulated tornado move across the screen, a specialist is more likely to discover a simulation error than by hovering for long hours over columns of numbers. Similarly, images can assist non-specialists to develop a certain degree of understanding of scientific undertakings. The power of abstraction inherent in visual representation and the innate human ability to rapidly process very large volumes of visual information are the key to the success of scientific visualization. [6,7]

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

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

Our research is concerned with the development of a methodology, rooted in a strong formal foundation, for selecting proper visualizations of concurrent computations. The particular approach we are currently exploring could be characterized in general terms as a *proof-based visualization*. We justify our selection of this approach by observing that in the concurrent domain operational thinking is often rendered ineffective by the very large number of possible interleavings of events. By contrast, assertional reasoning [2] has emerged as an effective tool for acquiring valuable insight about the workings of such programs—today, a concurrent program without a proof is hardly given any serious consideration. Since program understanding is also the primary objective of visualization, the connection between proofs and visualization seems to be a natural one, at least in principle.

We showed previously [9] that there is also a way to exploit this relation at a practical level. Safety and liveness properties, typically used in reasoning about concurrent computations, have appropriate visual counterparts. A safety property can be rendered as a stable visual pattern, while liveness may be captured by an evolving one. Moreover, we have some preliminary evidence that the form of the logical predicate capturing the program property (e.g., existential versus universal quantification) also can provide valuable hints on the type of visual representation one ought to consider.

One important step supporting the development of the visualization methodology involves building an exploratory environment which allows us to specify a broad range of concurrent computations and visualizations. Concurrent computations are specified using *Swarm* [10]. Swarm is a computational model which extends UNITY and its proof logic to include content-based accessing to data, dynamic statements, and dynamic partial-synchrony. Swarm was selected because of its expressive power, its assertional-style proof logic, and the uniform tuple-like representation of computational states. Abstract representations of the computation are generated by mappings computational states to a three dimensional world of geometric objects. Changes in the state of the geometric world, i.e., primitive visual events, are later mapped into animation sequences which are finally mapped to images. The mapping approach, called

declarative visualization, makes it possible to alter dynamically the definition of a particular visualization without impacting the specification of the computation. This high degree of flexibility, while desirable in general, is critical to the success of an exploratory environment.

This paper is concerned with the method we use to specify visualizations and with the motivation for the choices we have made so far. Section 2 introduces a sample concurrency problem, some Swarm notation required to capture the program state, and several important formal properties of the algorithm. The sample problem, diffusing computations, is used as an example throughout the paper. Section 3 overviews the specification method and basic notational conventions. Section 4 describes the process of specifying a complete visualization for diffusing computations. Section 5 provides a status report on the development of the exploratory visualization environment. Conclusions appear in Section 6.

2. Specifying Concurrent Computations

In this section we introduce the diffusing computation problem and explain how it is expressed in Swarm [10]. Since the focus of this paper is the specification of visualizations, not computations, we explain Swarm and its notation only to the extent necessary to support the remainder of the presentation. The reference list, however, includes several published articles that discuss Swarm and its proof logic [3].

Underlying the Swarm language is a state-transition model similar to that of UNITY [2]. In Swarm, the state of a computation is represented by the contents of the *dataspace*, a set of content-addressable entities. The model partitions the dataspace into three subsets: the *tuple space*, a finite set of data tuples; the *transaction space*, a finite set of *transactions*; and the *synchrony relation*. Every element of the dataspace has tuple-like representation, i.e., it is a pairing of a type name with a sequence of values. In addition, a transaction has an associated behavior specification. The execution of a transaction is modeled as a transition between dataspaces. An executing transaction examines the dataspace, then deletes itself from the transaction space and, depending upon the results of the dataspace examination, modifies the dataspace by inserting and deleting tuples and synchrony relation entries and by inserting (but not deleting) other transactions. 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 along with all other transactions belonging to the same synchronic group. The entire synchronic group is executed. The transaction selection is fair in the sense that each transaction in the transaction space will eventually be chosen. The notation for tuples and simple transactions are the only portions of the Swarm model required to handle the diffusing computation problem.

Problem. In the *Diffusing Computations Problem* one is given a collection of N interconnected processes which communicate by sending messages over channels and an external environment which can send messages to some of the processes. Each process is either active or inactive; only active processes can send messages. An active process may become inactive at any time; an inactive process becomes active only on receipt of a message. Computation begins when the environment sends messages to one or more processes and ends when all processes are inactive and the environment is not sending any more messages. The environment must detect the termination of the computation, i.e., when all processes are inactive and no further messages can be received by any process.

Solution outline. The solution we present here is based on that proposed by Dijkstra [4]. Each process acknowledges every message it receives. A process that is inactive records the identity of the process (henceforth called parent process) whose message caused a transition from the inactive to the active state, but delays the acknowledgement of that message until its state changes back to inactive. An active process immediately acknowledges all the messages it receives. Termination is detected by the environment when it has received acknowledgements for every message sent. The correctness of this solution relies on the following two invariants:

(1) The set of all active processes forms a tree whose root is the environment.

(2) For each process, either the process is active or the number of pending acknowledgements is zero.

Furthermore, under the assumption that the underlying computation terminates, the following progress condition holds:

(1) A configuration in which only the environment is active eventually leads to a configuration in which neither the environment nor the processes are active.

It is this particular solution and the properties above that we will be using to illustrate the mechanics of specifying visualizations.

Data representation. In our approach visualizations are specified as mappings from the state space of the computation to images on the screen. The computation state in Dijkstra's solution is defined by the local state of the environment and of each process and communication channel. If we ignore the message contents and the computation carried out by the individual processes, the computation state can be captured in the following manner. The environment state consists of a single data tuple, of type *environ*, which stores the number of messages the environment will be initiating in the future and the number of unacknowledged past messages. The process state consists of a single tuple, of type *process*, which stores the process identity, its status (active or inactive), the number of messages sent but not yet acknowledged, and the identity of the initiation source (the process whose message caused the most recent transition from inactive to active status, i.e., the parent in the tree). The process identifiers range over 1 through N. The special identifier 0 is used to refer to the environment. The channels are bidirectional; messages flow in one direction and acknowledgements in the opposite one. The state of each channel consists of two tuples, one of type *msg* and the other of type *ack*. Each tuple stores the source and destination for the message (acknowledgement) and the number of messages (acknowledgements) in transit. In Swarm, the four tuple types introduced so far are declared as follows:

```
environ(messages_to_send : natural; pending_acks : natural);
process(id : processid; status : active_or_inactive; pending_acks : natural; parent : processid);
msg(source : processid; destination : processid; in_transit : natural);
ack(source : processid; destination : processid; in_transit : natural)
```

where *processid* is a natural in the range 0 to N and *active_or_inactive* is the enumeration {active, inactive}.

Computation/communication. In Swarm, both computation and communication are reduced to atomic transformations of the dataspace called transaction executions. Transactions appear in the dataspace and have a tuple-like representation consisting of a transaction type name and a sequence of fully instantiated parameters. A transaction execution involves a query evaluation over the dataspace which results in the binding of some local variables and is followed, if successful, by deletions and insertions into the dataspace, in this order. Since multiple sets of dataspace entities may satisfy the same query, nondeterminism is manifest in the query evaluation. Nevertheless, once the variables are bound, the dataspace changes are fully specified and deterministic.

The environment behavior, for instance, can be captured by a single transaction of type *Env()* defined as follows:

```
Env() ≡
        i,k,c,q : environ(k,c)†, k > 0, msg(0,i,q)†
            →  environ(k–1,c+1), msg(0,i,q+1)
    ||  c,i,q : environ(0,c)†, c > 0, ack(i,0,q)†, q > 0
            →  environ(0,c–1), ack(i,0,q–1)
    ||  ¬environ(0,0)
            →  Env()
```

Env is a transaction class consisting of a single transaction instance *Env()* which, in turn, consists of three synchronously executed subtransactions separated by the ||. The synchronous execution forces the three subtransactions to synchronize after the evaluation of the query, after the performance of all dataspace deletions (marked by daggers), and finally after the completion of all dataspace insertions.

Each subtransaction consists of a query followed by an action shown after the arrow. The query part of the first subtransaction above, for instance, checks if there are still messages that the environment must send out by looking for a tuple *environ(k,c)* with *k* greater than zero and selects some process with which the environment can communicate directly by looking for any tuple of the form *msg(0,i,q)*, i.e., a channel from 0 (the environment's identifier) to *i* with *q* messages in transit. Having bound the variables *i*, *k*, *c*, and *q*, the action part is performed in two phases. The deletion of the tuples *environ(k,c)* and *msg(0,i,q)* takes place first, followed by the insertion of two new tuples containing the updated environment and channel states. Insertions are specified by listing a data tuple or transaction after the arrow. Deletions are specified either by listing a data tuple marked by a dagger in the action part or, for reasons of convenience, by placing a dagger on a tuple in the query part.

Using similar mechanics, the second subtransaction checks if all environment messages have been dispatched and, if this is the case, seeks out arriving acknowledgments. If any acknowledgement is found, it is removed from the respective channel and from the environment state by decrementing the counters in *environ(0,c)* and *ack(i,0,q)*. Finally, the last subtransaction recreates the transaction *Env()* as long as there are messages that need to be delivered and acknowledgements that have not arrived. This is necessary because a transaction is automatically deleted as part of its execution.

Process representation. In Swarm there is no concept of process and there are no sequential programming constructs. Transaction types and transaction instances are available instead. Since Swarm lacks any sequential constructs, sequencing is accomplished by defining continuations in the form of new transactions to be inserted into the dataspace. Similarly, non-deterministic selection among multiple behavior choices is modeled by placing in the dataspace a transaction for each alternative. For this reason, each process *I* participating in the diffusing computation is represented by three transactions: *SendMsg(I)* which processes the dispatching of messages whenever the process is active; *GetMsg(I)* which processes the arrival of messages and the transition from inactive to active status; and *PassAck(I)* which processes the arrival of acknowledgements and the transition from active to inactive status. The definition of the corresponding transaction classes (parameterized by *I* in the range 1 to *N*) is given below:

```
SendMsg(I) ≡
        i,q,c,p : process(I,active,c,p)†, msg(I,i,q)†
            →  process(I,active,c+1,p), msg(I,i,q+1)
    ||  true
            →  SendMsg(I)

GetMsg(I) ≡
        i,q,c : process(I,inactive,0,⊥)†, msg(i,I,q)†, q > 0
            →  process(I,active,0,i), msg(i,I,q–1)
    ||  i,q,c,p,k : process(I,active,c,p), msg(i,I,q)†, q > 0, ack(I,i,k)†
            →  msg(i,I,q–1), ack(I,i,k+1)
    ||  true
            →  GetMsg(I)

PassAck(I) ≡
        i,c,p,k : process(I,active,c,p)†, c > 0, ack(i,I,k)†, k > 0
            →  process(I,active,c–1,p), ack(i,I,k–1)
    ||  i,c,p,k : process(I,active,0,p)†, ack(I,p,k)†
            →  process(I,inactive,0,⊥), ack(I,p,k+1)
    ||  true
            →  PassAck(I)
```

The symbol ⊥ stands for an undefined process identifier.

Initialization. The initial configuration of the dataspace is defined by

```
[I, J : 1 ≤ I ≤ N, 0 ≤ J ≤ N, J can send to I ::
        environ(K,0);
        process(I,active,0,⊥); msg(J,I,0); ack(I,J,0);
        Env();
        SendMsg(I); GetMsg(I); PassAck(I)
]
```

where *K* is some natural number representing the number of messages the environment will send. All transaction instances are persistent (i.e., re-create themselves and hence remain in the dataspace) except *Env()*. The data tuples change to reflect changes in the state of the environment and of the processes.

Termination detection. Termination is defined as a state where all processes are inactive, no messages or signals are in transit, and the environment has no more messages to send. From the above transaction specifications, it is possible to show that

$$\neg \text{Env}() \Rightarrow \text{termination}$$

i.e., that the absence of any *Env()* tuple in the dataspace indicates that the computation has terminated.

3. Declarative Visualization

In general, program visualization methods are either imperative or declarative. The distinction is based on the way in which computations are related to graphical representations.

Imperative methods treat visualization as a side effect of the computation. They require annotation of the underlying program with directives issued to the display device. In Balsa [1], for instance, the animator identifies sections of sequential code that represent "significant events" and augments them with calls to operations on predefined graphical objects. The animator needs in-depth knowledge of the code and must program all display changes associated with each significant event. The approach is somewhat tedious and inflexible. Also, it is not clear how one can extend the approach to concurrent programming where the "significant event" may be a global state change caused by particular combination of local state changes, i.e., when there may not be a unique piece of code that is always associated with the occurrence of the particular event.

Declarative methods, on the other hand, see visualization as a mapping either from program states to graphical objects or from program events (i.e., state transitions) to object transformations. Because of difficulties in specifying events (particularly in a concurrent environment), most declarative visualization systems opted for simple mappings from variables to object attributes. In PROVIDE [8] and PVS [5], for instance, icon parameters depend on process variables; changes to the variables are automatically transmitted to the display mechanism. The principal shortcoming of these systems is the low level of abstraction. The graphical objects are necessarily closely related to the program variables, and no mechanism for higher levels of abstraction is available (for example, it is not easy to detect and display concepts such as "all variables have values greater than 10").

We believe that, at present, an inhibiting factor in the use of declarative visualization is the difficulty of specifying states and events in modern programming languages. The tuple-like representation of the Swarm program state clearly helps alleviate this problem and makes it possible to attempt the application of this method to complex concurrent programs. With regard to the choice between event versus state mapping, we favor the latter. The proof logic on which we base our visualization methodology relies on reasoning about program states. Abstract objects can be easily conceived as abstractions of the program state. The approach is equally applicable to process-based computations (e.g., shared variables and message passing) where the notion of control state plays an important role; to rule-based systems where control state is practically absent; and to database systems where portions of the data may never be touched by any executing transactions.

In an earlier paper [9] we suggested treating declarative visualization as a two-step mapping (Figure 1): the program state is first mapped to a world of 3D geometric objects which, in turn, is mapped to images on the screen under direct user control. The two mappings are called visual *abstraction* and image *rendering*, respectively. Visual abstraction is supported by a set of heuristics rooted in methods for formal verification of concurrent programs and based on the notion that both safety (**stable, invariant, constant**) and progress (**ensures, leads-to**) properties have direct visual counterparts. Rendering exploits the capabilities made available by today's high-power graphics workstations which offer an exceptionally rich visual vocabulary including not only rapidly rendered complex three-dimensional objects but also color, movement, real-time transformations, hidden-surface elimination, lighting, etc. In the remainder of this section we propose a refinement of this model and appropriate notational changes.

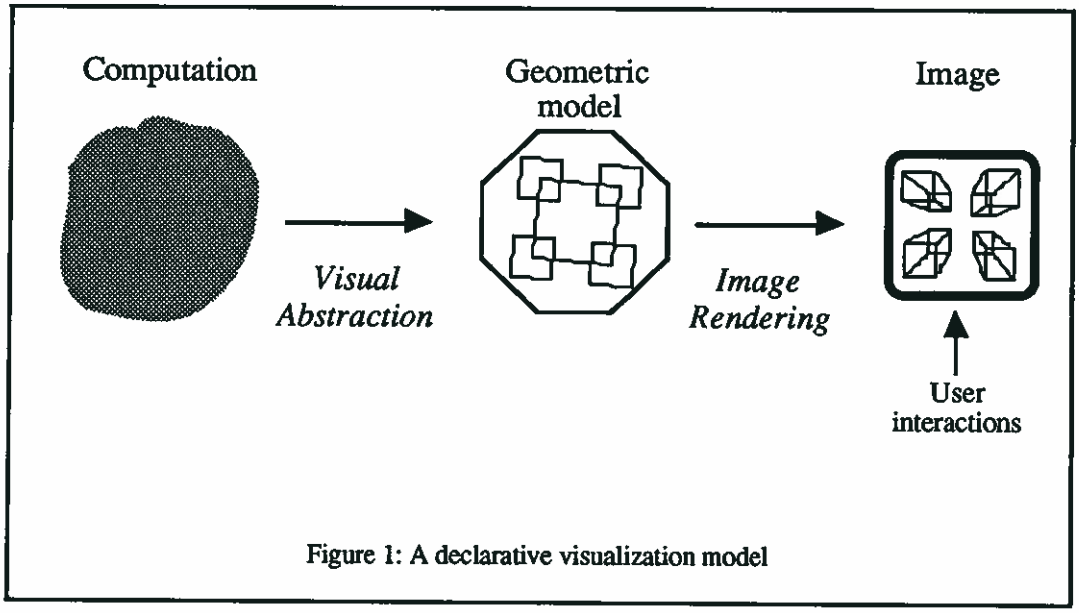

Figure 1: A declarative visualization model

3.1. Rule-based specification

The refinement of the visualization model resulted in the decomposition of the abstraction and rendering mappings into four other mappings whose name and purpose will be discussed in the following two subsections. The range and the domain of all these mappings are sets of tuples, generically called *spaces*. The mappings are specified by collections of visualization rules which are similar in style to Swarm subtransactions or to expert system rules. Operationally, a rule examines the input space of the mapping and produces tuples for the output space of the mapping. All mappings are re-evaluated after each atomic transformation of the state space.

Each visualization rule consists of an optional list of *variables*, a *query* part, and a *production* part. The notation for the rules is similar to the Swarm subtransaction notation, except that the symbol $\Rightarrow$ is used in the rule instead of the Swarm $\rightarrow$ to separate the query part from the production part:

$$v : Q \Rightarrow P$$

This notational choice was made both to syntactically distinguish visualization rules from subtransactions and to emphasize the important semantic difference between the two. A simple transaction, i.e., one consisting of a single subtransaction, specifies an atomic transition from one dataspace configuration to the next. A visualization rule, however, defines a logical relation between the input space and the output space. Any change in the input space is instantaneously reflected in the contents of the output space.

The query part of a visualization rule is an arbitrary predicate which includes tests for the presence of tuples in the input space. The production part is a list of tuples, which again may contain variables.

These tuples are associated with the output space of the mapping to which the rule belongs. We say that a variable is *instantiated* by the query if it occurs free in at least one component of the query; all variables which appear in the production part must be instantiated by the query. This convention is common in most rule-based languages.

The interpretation of a visualization rule is as follows: For every instantiation of the variables such that the query predicate is true, the corresponding tuples of the production will be present in the output space. Given a mapping (a collection of rules), the output space produced by the mapping is simply the union of all tuples produced by the rules.

More formally, assume we have a rule R which maps input space I to output space O. R has the form

$$v : Q \Rightarrow P$$

where v is a collection of variables, Q a query over I, and P a list of tuples in O possibly containing variables from v. Let i represent an arbitrary instantiation of the variables v (that is, i is a vector of values having the same length as v). Then the output of R for input space(s) I is the set of tuples

$$\text{out}(R,I) = \{\, i : Q_i^v \text{ is true for } I : P_i^v \,\}$$

The notation { variables : domain : term } represents the set of all terms such that the domain is true, and Q_i^v represents the result obtained by substituting for each occurrence of a variable of v in Q the corresponding value from i. We use the phrase "p is true for I" to mean that the query defined by p is true for the space I. Given a mapping M which consists of a set of rules, we define the output space of M given input space I as

$$\text{out}(M,I) = (\cup R : R \in M : \text{out}(R,I))$$

For reasons that will become apparent soon, the single-space domain of some of the mappings, called the primary input space, has been extended so as to include the previous instance of the input and output spaces. Under the assumption that there are no naming conflicts between the various spaces, the notation has been augmented as follows. Given α, a tuple in which the components may contain variables, on the left hand side of some rule: α refers to testing for the pattern α in the previous instance of the output space or current instance of the primary input space; and $old.\alpha$ refers to testing for the pattern α in the previous instance of the input space.

3.2. Decomposition of the abstraction mapping

Methodological considerations prompted the decomposition of the abstraction mapping into two parts. The proof-based visualization methodology we have been applying requires one to first develop a UNITY-like proof (or a proof outline) of a concurrent program in terms of program-wide properties. Next, visual counterparts of these properties are chosen in order to determine the structure and attributes of the geometric model. A stable property, for instance; once it becomes true it remains true forever—the definition applies equally well to a predicate over the program state or to a visual property of some image.

Since only those aspects of the program state which relate to the property of interest contribute to the selection of the geometric model, it makes sense to map first the state space of the computation (see Figure 2) to some more abstract and less complex representation. We call this newly introduced space and mapping the *proof space* and *proof mapping*, respectively. Because it is often the case that proofs rely on the use of auxiliary variables which maintain historical information, the proof rules need to access the old values of the auxiliary variables—in other words, the previous proof space—in order to compute the new ones.

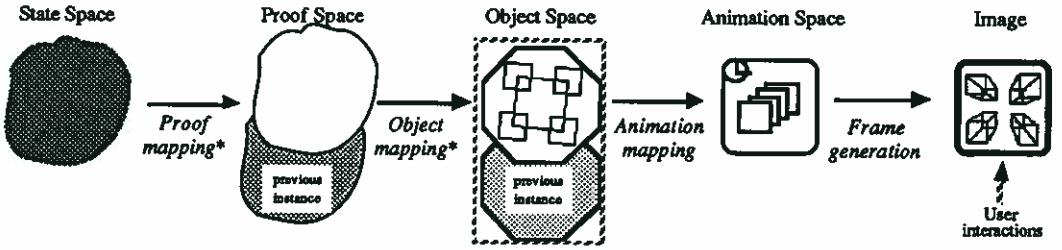

Figure 2: Revised visualization model
The input space of mappings marked by a star is extended so as to include the previous instance of the output space.

The proof space is mapped into a 3-D geometric model called the *object space* by employing an *object mapping*. The object mapping rules embody our judgments with regard to the best correspondence between formal program properties and their visual counterparts. Take, for instance, the property *P leads-to Q*, meaning *"if the computation reaches a state in which P is true it will eventually reach a state in which Q is true."* If, on one hand, proving a leads-to property involves proving the existence of *k* simpler transitions, one can expect to see *k* corresponding visual transitions, henceforth called visual events. If, on the other hand, the proof is by induction, one can expect to observe a sequence of visual representations which become gradually closer to some target representation, under some measure of similarity or distance. The object mapping rules are also permitted to access the previous instance of the output (object) space. The primary purpose of this accomodation is to permit certain forms of viewer interaction. For example, geometric information (such as position) can be stored in the object space; the user can modify this information by a "click-and-drag" interface with the image, thus re-positioning the object involved.

3.3. Decomposition of the rendering mapping

An animator may find our model as described very limiting because of the inability to accommodate smooth transitions between the images representing successive states. Full animation capability is simply not possible unless one is able to recognize events. This does not mean, however, that one needs to detect computational events! Our model does not preclude the use of animation techniques, but moves the concern with special effects into the visual realm: we treat these special effects as decompositions of primitive visual events. An exchange of the positions of two objects on the screen might be the direct result of a change in the state of the underlying computation. The animator can take this event and decompose it in a sequence of simpler visual events involving highlighting and synchronized movement along some prescribed trajectories. In this manner we are able to preserve the formality of state-based mapping without sacrificing aesthetics.

In order to support this visual event decomposition we have introduced the *animation mapping*, which takes as input the two most recent instances of the object space and generates tuples representing objects in 4-D (3-D plus time) space called the *animation space*. These animation tuples are interpreted by the *frame generation mapping* and converted into sequences of images called *frames*. The frame generation mapping is performed by an interpreter which makes direct use of the rendering capabilities available on some graphics workstation, the Personal Iris in our case. The interpreter also takes into account user interactions affecting the viewpoint, scaling, light sources, etc.

Each tuple of the animation space represents one graphical object in the sequence of frames. The tuple type corresponds to the object type (*line*, *sphere*, etc.). The tuple components correspond to the various attributes of the object; for example, a *line* object has the attributes *lifetime*, *from*, *to*, and *color*. For notational convenience, we permit the tuples of the animation space to take the form

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

where each *attribute* is the name of an object attribute and each *value* is a value of the appropriate type. It is not necessary to specify values for all the attributes of the object; defaults are used for any value not specified. For example, the tuple

line(*from* = [0,0,0], *to* = [5,0,0])

would represent a *line* object from coordinate (0,0,0) to coordinate (5,0,0). The line would be colored white (the default for that attribute).

Animation necessarily involves certain concepts related to time; we measure time in terms of frames, with the first frame produced in each transition being numbered 0, the next 1, and so on. Animation is accomplished by assigning time-dependent values to the attributes of the objects. A collection of functions are provided to generate these time-dependent values. For example, the function *ramp*(*t0*, *v0*, *t1*, *v1*) generates a linear interpolation between times *t0* and *t1*, with the value of the interpolation being *v0* at time *t0* and *v1* at time *t1*. Mechanisms for composing the effects of the various functions are also provided.

Given an animation space, the image mapping first determines how many frames will be generated for the transition. This is done be examining the time arguments of all the functions and the *lifetime* attributes of all the objects (the *lifetime* attribute is a pair of values [*tmin*, *tmax*] which indicates that the object is to be produced only between frames *tmin* and *tmax*). The maximum value of any time involved is used as the number of frames to be generated. Once this is determined, the sequence of frames is produced and displayed. In each frame, each animation tuple is used to determine the attributes of the object and the necessary graphics commands are executed to produce the object.

4. Visualization of the Diffusing Computations Problem

In this section we specify a visualization for the diffusing computations program presented earlier. We first construct a complete simple visualization by giving the rules that define the proof, object, and animation mappings (the frame generation mapping is fixed by the behavior of the interpreter). We then show several ways in which this visualization could be modified to emphasize different aspects of the computation behavior.

4.1. First Visualization

We use as a starting point for our visualization the key invariants required in the correctness proof of the diffusing computation program:

- The parent information in those process tuples whose status is active forms a directed tree rooted at the environment.
- For each process, either the process is active or the number of pending acknowledgements is zero.

The first of these invariants establishes a tree structure which is used to detect termination. The second ensures that no process will become inactive until it has received an acknowledgment for every message it has sent.

The first invariant is easily captured—we generate a tree consisting of all active processes and the environment. The second invariant is also captured by the tree representation: the second invariant ensures that no process becomes inactive while it has active children, so violation of this invariant (the removal from the tree of a node whose children are still present) leads to the creation of an easily observed forest.

Proof mapping. The diffusing computations program makes use of four data tuple types and four transaction types. Not all of them are essential to capturing the invariants above. In the proof mapping, we isolate those aspects of the computation which are of interest. In this case, we wish to collect the information needed to generate the tree. This information will be represented in the proof space by

treenode tuples containing for each node of the tree the node's id, the id of the node's parent, and the distance of the node from the root (the environment). To generate the distance information, we make use of a known property of the underlying computation: a process can become active in a single atomic action only if its parent was already active. (Note that this property holds even if a single atomic action is the simultaneous and synchronous execution of multiple transactions.) Therefore, a node can be added to the tree only if its parent was present in the tree in the previous step; we can compute the node distance by querying the previous proof space to determine the distance of its parent.

Two rules accomplish the proof mapping. The first puts a node representing the environment into the tree, but only if termination has not been detected already. By examining the properties of the algorithm, we know that termination is detected if and only if no *Env()* transaction is present in the dataspace. This gives the rule:

```
    Env()
⇒   treenode(0,⊥,0)
```

The second rule puts the nodes for the active processes into the tree. The query part of the rule examines the previous proof space to determine the distance of the node's parent from the root:

```
id, ack, p, pp, dist :
    process(id, active, ack, p), treenode(p, pp, dist)
⇒   treenode(id, p, dist + 1)
```

For this particular problem, the proof mapping is quite simple; indeed, the mapping from the *Env* and active *process* tuples of the state space to the *treenode* tuples of the proof space is bijective. For more complex visualizations this would not be the case—for example, several tuples of the state space might contribute to a single tuple of the proof space. The ability to formulate arbitrarily complex queries in such cases is one of our model's chief strengths.

Object mapping. The object mapping transforms the abstracted information of the proof space into tuples which represent 3-D geometric objects (although these tuples are not necessarily in one-to-one correspondence with the final graphical objects). This mapping also completes the abstraction process by changing the state-oriented information of the proof space into geometric information.

In our example, we will construct the tree in three dimensions. The distance of each node from the root (environment) will be mapped to the Z-coordinate (but "reversed", so that the root has the most positive Z coordinate); the X and Y coordinates will be used to lay out the nodes representing the processes.

In order to lay out the tree, we need a function which, when given a process identifier, returns the X and Y coordinates at which that process is to be placed. Formally, we can express this as

mappos : ProcessIds → (Reals × Reals)

Functions are quite simple to represent as collections of tuples (obviously, because formally a function is nothing more than a collection of pairs). In this case, the function definition will consist of *mappos* tuples having three components, corresponding to the process identifier and its mapping to X and Y coordinates. The function is "applied" by formulating a query to instantiate variables for the process id and the X and Y coordinates. The function will be stored in the object space and "copied forward" by the rule:

```
id, x, y :
    mappos(id,x,y)
⇒   mappos(id,x,y)
```

This approach has several advantages. For example, it permits the function to be modified, either by the visualization process or by a user, and the modifications will be retained thereafter. This can be used for viewer interaction with the image: the viewer could change the position of a node (perhaps by a "click and drag" interface), and the change would alter the *mappos* tuple in the object space, resulting in the new

position applying to all objects generated thereafter. A similar approach could also be used to store such constants as the size and color of the various objects, the relative scaling in each dimension, and so on.

We construct the tree using spheres to represent the nodes and lines connecting the sphere centers to show the parent-child relations. Three-dimensional coordinates will be represented by triples enclosed in brackets. The following rules transform the tuples of the proof space into object tuples representing the tree:

```
x, y :
     treenode(0,⊥,0), mappos(0,x,y)
⇒    tree_sphere([x,y,0])

id, pid, dist, xi, yi, xp, yp : id ≠ 0 ::
     treenode(id,pid,dist), mappos(id,xi,yi), mappos(pid,xp,yp)
⇒    tree_sphere([xi,yi,–dist]), tree_line([xi,yi,–dist],[xp,yp,–(dist–1)])
```

The first rule generates the root node. The second generates a sphere for each non-root node and the line connecting it to the parent's sphere.

Animation mapping. The animation mapping transforms the object space into the animation space. The tuples of the animation space are in one-to-one correspondence with primitive graphical objects of the final image and are slightly different in style from the tuples of the other spaces. Specifically, the type of each tuple is the name of a class of primitive graphical objects that are part of the visual vocabulary of an animation space interpreter. Also, the tuple components assign time-dependent values to the object attributes.

The simplest mapping involves no animation—the spheres and lines representing the tree simply appear and disappear. Two rules which accomplish this are:

```
p :
     tree_sphere(p)
⇒    sphere(center = p, radius = 0.2)

p, pp :
     tree_line(p,pp)
⇒    line(from = p, to = pp)
```

We can improve the appearance of the visualization by animating the addition and removal of the spheres and lines. These occurrences are visual events that must be detected by examining two consecutive instances of the object space. Up to this point we have been able to postpone dealing with events. Although we need to recognize events and map them to sequences of images, our approach is distinct from others in several important ways. First, we deal with visual and not with computational events. Second, events are used to support exclusively the animation process. Third, the mapping of individual events into sequences of events requires no reference to the program code. Finally, the declarative style and the rule-based notation is preserved by extending the input space of the animation rules to include the previous instance of the object space, thus allowing to check for differences between the two, and by treating the animation space as a 4-D space where the fourth dimension is time.

The proposed animation is accomplished by recognizing three separate cases: the addition of an object, the persistence of an object, and the removal of an object. We need rules for each of these cases. Consider first the addition of a node to the tree, resulting in the addition of a sphere and (probably) a line to the image. Assume we want to animate this as follows: the line will grow out from the parent sphere until it reaches the position of the new sphere; the new sphere will then expand outward from a point to its full radius. This is accomplished by the following rules:

p, pp :
    tree_line(p,pp), ¬old.tree_line(p,pp)
⇒ line(lifetime = [1,10], from = pp, to = ramp(1,pp,5,p))

p :
    tree_sphere(p), ¬old.tree_sphere(p)
⇒ sphere(lifetime = [5,10], center = p, radius = ramp(5,0.0,10,0.2))

The first rule detects when a line is added and generates a line which will be present from frame 1 to frame 10 (the *lifetime* attribute). One endpoint of the line will be at the center of the parent (location *pp*); the other endpoint will move from the center of the parent to the center of the child between frames 1 and 5, and will remain at the center of the child thereafter. The second rule generates a sphere which is present from frame 5 to frame 10; the sphere's radius changes from 0.0 at frame 5 to 0.2 at frame 10. Together, the two rules achieve the effect of a line growing from the parent to the location of the child, then the sphere growing outward from that point. (The frame numbers were chosen to produce a pleasing effect in the final animation. Our ability to rapidly change the visualization by changing the rules is quite helpful in such development.)

The rules which detect the case where a line or sphere is present in both the previous and current spaces are:

p :
    tree_sphere(p), old.tree_sphere(p)
⇒ sphere(center = p, radius = 0.2)

p, pp :
    tree_line(p,pp), old.tree_line(p,pp)
⇒ line(from = p, to = pp)

We omit the rules which accomplish the removal of a line and sphere, as they are quite similar to those for the addition.

4.2. Changing the Visualization

The visualization rules can be easily augmented and modified at any time during the visualization process, either for the sake of refining the visual presentation or out of the desire to explore visually additional properties of the computation. Rules that rely on the use of historical information which is not available at the time the rule is first introduced may require the restart of the execution. All other rules take effect as soon as a new atomic transformation of the dataspace is accomplished. To illustrate the ease with which new rules may be formulated and added to the visualization we consider next several enhancements to the visualization already introduced so far.

Assume that we want to show the state (active or inactive) of all the processes and message channels. This might be prompted by a desire to see that all the active processes and none of the inactive processes are present in the tree. We will represent processes by spheres and channels by lines linking the spheres into a graph. To visually link this process state information with the tree, we use the same mapping of process identifiers to X and Y coordinates in both the graph and the tree. We will place the graph "above" the tree, at a Z coordinate of 2 (chosen somewhat arbitrarily). States will be represented using colors, red for inactive and green for active.

Proof rules:

Env() ⇒ active_proc(0)

¬Env() ⇒ inactive_proc(0)

id, ack, p : process(id, active, ack, p) ⇒ active_proc(id)

id, ack, p : process(id, inactive, ack, p) ⇒ inactive_proc(id)

i, j, c1, c2 : i < j, msg(i,j,c1), msg(j,i,c2), (c1 > 0 ∨ c2 > 0) ⇒ active_channel(i,j)

i, j : i < j, msg(i,j,0), msg(j,i,0) ⇒ inactive_channel(i,j)

The first two rules detect when the environment is active or inactive (i.e., termination detected or not) and generate an *active_proc* or *inactive_proc* tuple as appropriate. The next two rules do the same for the other processes. Finally, the last two rules determine if a channel is active or not; activity on a channel is defined as the presence of a message in either direction. The predicate $i < j$ ensures that exactly one tuple appears for each channel.

Object rules:

```
i, x, y :
     active_proc(i), mappos(i,x,y)
⇒    process_sphere([x,y,2],[0,255,0])

i, x, y :
     inactive_proc(i), mappos(i,x,y)
⇒    process_sphere([x,y,2],[255,0,0])

i, j, xi, yi, xj, yj :
     active_channel(i,j), mappos(i,xi,yi), mappos(j,xj,yj)
⇒    channel_line([xi,yi,2],[xj,yj,2],[0,255,0])

i, j, xi, yi, xj, yj :
     inactive_channel(i,j), mappos(i,xi,yi), mappos(j,xj,yj)
⇒    channel_line([xi,yi,2],[xj,yj,2],[255,0,0])
```

These rules generate object space tuples representing the processes and channels. Colors are expressed as triples representing the red, green, and blue components of the color on a scale of 0 to 255; thus, [255,0,0] is red and [0,255,0] is green.

Animation rules:

```
p, c :
     process_sphere(p,c), old.process_sphere(p,c)
⇒    sphere(center = p, radius = 0.2, color = c)

p, c1, c2 :
     process_sphere(p,c1), old.process_sphere(p,c2), c1≠c2
⇒    sphere(center = p, radius = 0.2, color = step([255,255,255],1,c1))

p1, p2, c :
     channel_line(p1,p2,c), old.channel_line(p1,p2,c)
⇒    line(from = p1, to = p2, color = c)

p1, p2, c1, c2 :
     channel_line(p1,p2,c1), old.channel_line(p1,p2,c2), c1≠c2
⇒    line(from = p1, to = p2, color = step([255,255,255],1,c1))
```

These rules simply transform the object tuples into the primitive graphical objects which make up the final image. As a way of attracting the viewer's attention, the channels that change status are made to shine bright and then change to the new color (the three parameters of the *step* function begin the initial value, the time at which the value changes, and the final value).

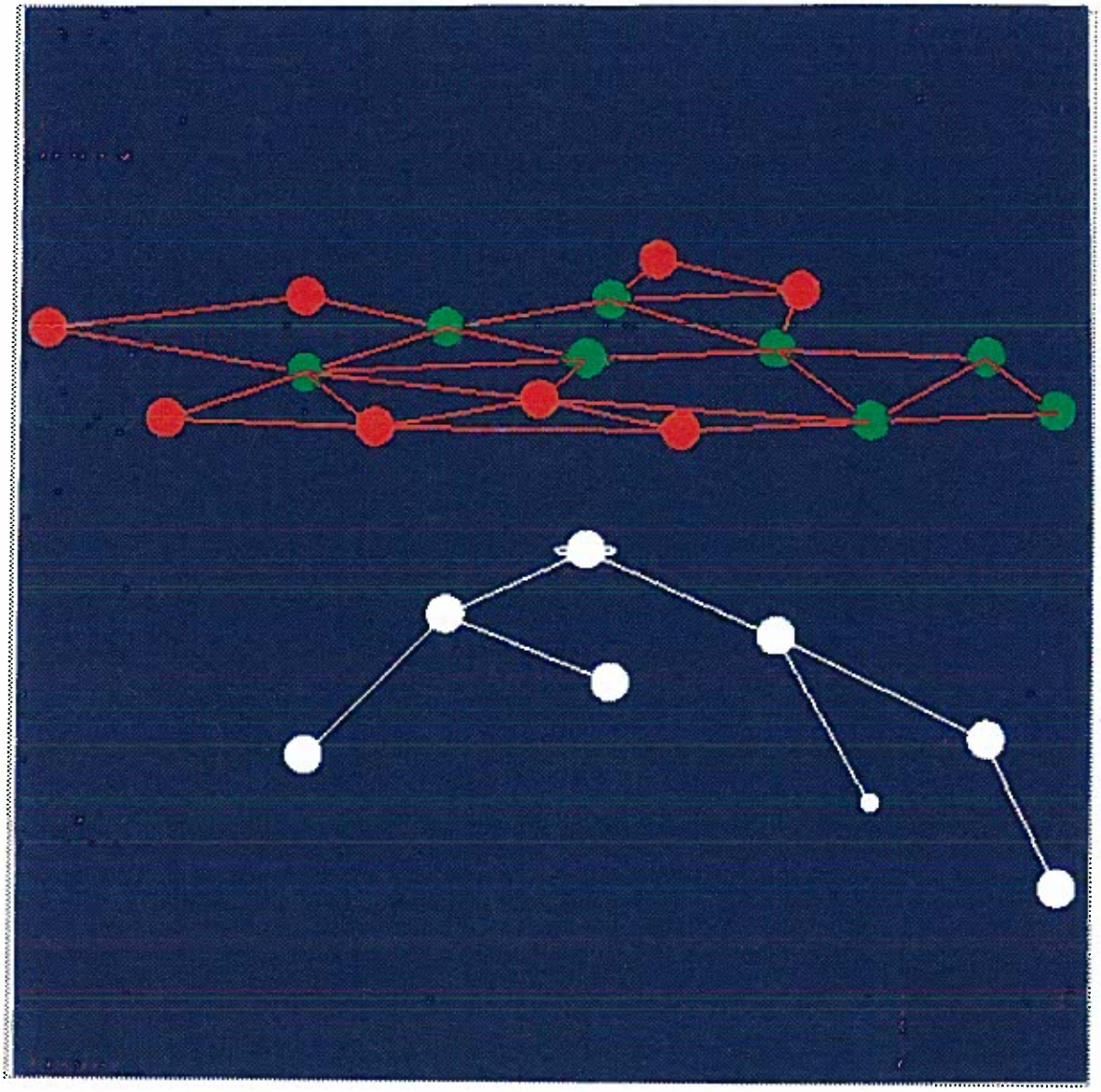

Figure 3: Sample visualization

One frame of the visualization resulting from this final collection of rules augmented by rules that tag the environment is depicted in Figure 3. This particular frame is being viewed from a point slightly above the X-Y plane; the Z-axis is directed vertically upward through the center of the figure. The graph in the upper part of the figure is the computation graph; eight nodes can be seen to be active (green), including the environment node (circled in green). The lower part of the graph is the termination-detection tree; again, the environment is circled. Because of the chosen angle of view, tree height generally corresponds to vertical position on the page. This particular frame was selected from the middle of a transition in which a node was being added; the new sphere is visible toward the bottom right of the tree and is slightly smaller than the others.

Naturally, the limitations of the print media prevent capturing the full effect of the visualization. In particular, we cannot effectively capture such animation effects as the growth of the sphere and the flashing of the elements of the computation graph. A video tape showing a complete animation is available.

5. Implementation

The specification techniques described in this paper are being used in the implementation of a versatile exploratory visualization environment called *Pavane*. Its goal is to enable experimentation with various visualization methodologies and to investigate strategies for integrating concurrent programming, visualization, and video production.

Our exploratory visualization environment consists of two main components. The first of these, *SwarmExec*, is a Prolog-based Swarm simulation which runs on a Macintosh. SwarmExec simulates the operation of a Swarm program and superimposed visualization. The various tuple spaces involved in the simulation, including the Swarm dataspace and the spaces of the visualization model, are represented as collections of Prolog facts. The window-based visual editors facilitate the specification of both Swarm programs and visualization rules and their translation into Prolog goals. The editors may be invoked by the user any time during the visualization.

The second component of our visualization environment is *SwarmView*, a graphics program and interpreter which runs on a Personal Iris. The Macintosh and Iris are connected through the Ethernet; the Macintosh sends a description of the animation space produced by SwarmExec to SwarmView across this link. SwarmView reads the successive animation spaces (transitions) produced by SwarmExec and generates the sequence of frames described by each transition. SwarmView also provides facilities whereby the user can change the image viewpoint, pause the animation, and take "snapshots" of the images.

6. Conclusions

In this paper we presented a new model for visualizing concurrent computations. The key features of the model are its declarative nature, its rule-based notation, its ties to program verification, and the emphasis on three-dimensional visualizations. *At the start of this research we were faced with two principal questions: Is it possible for a strict declarative approach based on mapping states to images to handle elegantly (1) auxiliary variables commonly used in proofs and (2) special effects essential in most visual presentations?* Having answered *yes* to both questions, we plan to turn our attention to several new issues: a formalization of our proof-based visualization methodology, a study of the adequacy of our current visual vocabulary, and the addition of non-textual methods for constructing animation rules.

7. References

[1] Brown, M. H. and Sedgewick, R., "A System for Algorithm Animation", *ACM Computer Graphics (Proceedings SIGGRAPH'84)*, Vol. 18 No. 3, pp. 177-186 (July 1986).

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

[3] Cunningham, H. C. and Roman, G.-C., "A UNITY-Style Programming Logic for a Shared Dataspace Language," *IEEE Transactions on Distributed and Parallel Computing* Vol.1, No. 3, pp. 365-376 (July 1990).

[4] Dijkstra, E., *A Discipline of Programming*, Prentice-Hall, Engelwood Cliffs NJ (1976).

[5] Foley, J. D. and McMath, C. F., "Dynamic Process Visualization", *IEEE Computer Graphics and Applications*, Vol. 6 No. 2, pp. 16-25 (March 1986).

[6] Frenkel, K. A., "The Art and Science of Visualizing Data", *Communications of the ACM*, Vol 31 No. 2, pp. 111-121 (February 1988).

[7] McCormick, B. H., DeFanti, T. A., and Brown, M. D., "Visualization in Scientific Computing", *ACM Computer Graphics*, Vol. 21 No. 6 (November 1987).

[8] Moher, T.G., "PROVIDE: A Process Visualization and Debugging Environment", *IEEE Transactions on Software Engineering*, Vol. 14 No. 6, pp. 849-857 (June 1988).

[9] Roman, G.-C. and Cox, K., "A Declarative Approach to Visualizing Concurrent Computations", *IEEE Computer*, Vol 22 No. 10, pp. 25-36 (October 1989).

[10] Roman, G.-C. and Cunningham, H. C., "A Shared Dataspace Model of Concurrency - Language and Programming Implications," *Proceedings of the 9th International Conference on Distributed Computing Systems*, pp. 270-279 (June 1989).