Report Number: WUCS-91-07

1990-11-01

Visualization in Concurrent Contexts: A Model

Kenneth C. Cox

Visualization is defined as the transformation of information into a graphical form. In recent years, visualization has increasingly been used in a variety of applications. Our work focuses on the visualization of concurrent computations. We wish to collect information about the operational behavior of concurrent computations. We wish to collect information into a graphical representation, and display the resulting image.

Visualization in Concurrent Contexts: A Model

Kenneth C. Cox

WUCS-91-07

November 1990

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

1 Introduction

Visualization is defined as the transformation of information into a graphical form. In recent years, visualization has increasingly been used in a variety of applications. Our work focuses on the visualization of concurrent computations. We wish to collect information about the operational behavior of concurrent programs, transform that information into a graphical representation, and display the resulting image.

The primary thrust of our work has been involved with the development of a model suitable for the concurrent domain. Existing visualization systems (designed for visualization of single-processes) use an *imperative* model; events occurring in the computation are detected, and as a side effect cause changes in the image. We feel that this approach, although suitable for sequential processes, is unsuited for the concurrent domain where the concept of *event* is often ill-defined.

We instead model visualization in a *declarative* fashion, as a transformation from the state of the computation to an image. The visualization is specified by defining the mappings that transform the state to the image. The concept of event still exists, in that changes in the state result in changes to the image; but events *per se* are not detected We believe that this approach has a number of significant advantages compared to existing models. In particular, it provides easy separation, both conceptually and operationally, between the computation being visualized (the *underlying computation*) and the process of visualization. For this reason we refer to the visualization as being *superimposed* on the underlying computation.

The remainder of this paper describes our visualization model. Section 2 is an overview of the model. Sections 3 and 4 describe the two major components of the model, the *spaces* and the *mappings* between spaces, in greater detail. Section 5 contains several example visualizations. Additional information about the model and our testbed implementation can be found in the papers listed in the Bibliography.

2 Overview

We treat visualization as a series of transformations or *mappings* from the state of the computation to an image. The various intermediate results of the mappings are referred to as *spaces*. Each space consists of a set of tuples having the general form

$$typename(component_1, component_2, \ldots, component_n).$$

Four spaces are included in the model:

- The *state space* is that used by the computation.
- The *proof space* is an abstraction of those properties of the state space which are required in the visualization. Both the current and previous instances of the proof space are available for examination in the model. Each successive instance of the proof space is produced from the state space and the previous instance of the proof space by the *proof mapping*.
- The *object space* contains the collection of graphical objects making up the image. Both the current and previous instances of the object space are available for examination in the model. Each instance of the proof space is produced from the state space and the previous instance of the object space by the *object mapping*.
- The *animation space* contains tuples which represent graphical events. It is produced from the object space and previous object space by the *animation mapping*.

We consider production of the animation space to complete the process of visualization; in practice, a graphical engine is required to transform the animation space into the final images. We have developed such an engine, called *SwarmView*, which runs on a Silicon Graphics Personal Iris®.

Each of the three mappings are specified declaratively as collections of rules. Each rule defines a logical relationship between two of the spaces of the model, called the rule's *input space* and *output space*. Any change in a rule's input space are instantaneously reflected in the output space. A complete visualization consists of declarations of all three mappings; the composition of these mappings defines a logical relationship between the state space and the animation space (and hence, the final image).

Rules belonging to the proof mapping are called *proof rules*; they have as input space the state space and as output space the proof space. Rules belonging to the object mapping are called *object rules*, have as input space the current and previous proof spaces, and have as output space the object space. Rules belonging to the animation mapping are called *animation rules*, have as input space the current and previous object spaces, and have as output space the animation space. The relations between the various spaces and mappings are represented in Figure 1.

Each visualization rule contains an optional list of *variables*, a *query* and a *production*:

variables : *query* $\Rightarrow$ *production*

The query is an arbitrary predicate which may include patterns to be matched against the rule's input space. The production is a list of tuples from the rule's output space. Informally, such a rule can be treated as producing a collection of tuples as follows: For every instantiation of the variables such that the query is true for the particular input space, the corresponding tuples of the production will be present in the output space.

Operationally, a visualization is performed by taking a particular state space, applying all the rules of the proof mapping and collecting the resulting tuples as the new proof space. This proof space and the previous space are then used as input space for the object mapping and produce the new object space. Finally, the new object space and the previous space are used as input space for the animation mapping and produce the new animation space.

We assume that the state space undergoes atomic transitions, and that the visualization process is sufficiently fast that each state can be completely transformed to the animation space before the next transition. We also assume that only the current state of the computation may be examined by the visualization process; this assumption may prove to be overly pessimistic in practice, but is required to keep the model general.

We have selected *Swarm* as our underlying computational model. Swarm models the state of a concurrent computation using an entity called the *dataspace*. The information contained in the dataspace is represented by tuples. Tuples represent both data and process activity; the latter is embodied by *transactions*, which specify atomic transformations to be applied to the dataspace, and the *synchrony relation*, which specifies collections of transactions which are to be executed simultaneously.

Transactions are represented in the dataspace by tuples, each representing a specific instance of the transaction type; the transaction behavior is specified separately. A transaction is made up of one or more subtransactions. A subtransaction consists of a *query* which attempts to match certain patterns against the dataspace and an *action* which specifies a modification of the dataspace. The synchrony relation is also represented by tuples, each linking two transactions. The reflexive symmetric transitive closure of the synchrony tuples present in the dataspace defines the synchrony relation; naturally, this is an equivalence relation. The synchrony relation partitions the transactions into collections called *synchronic groups*.

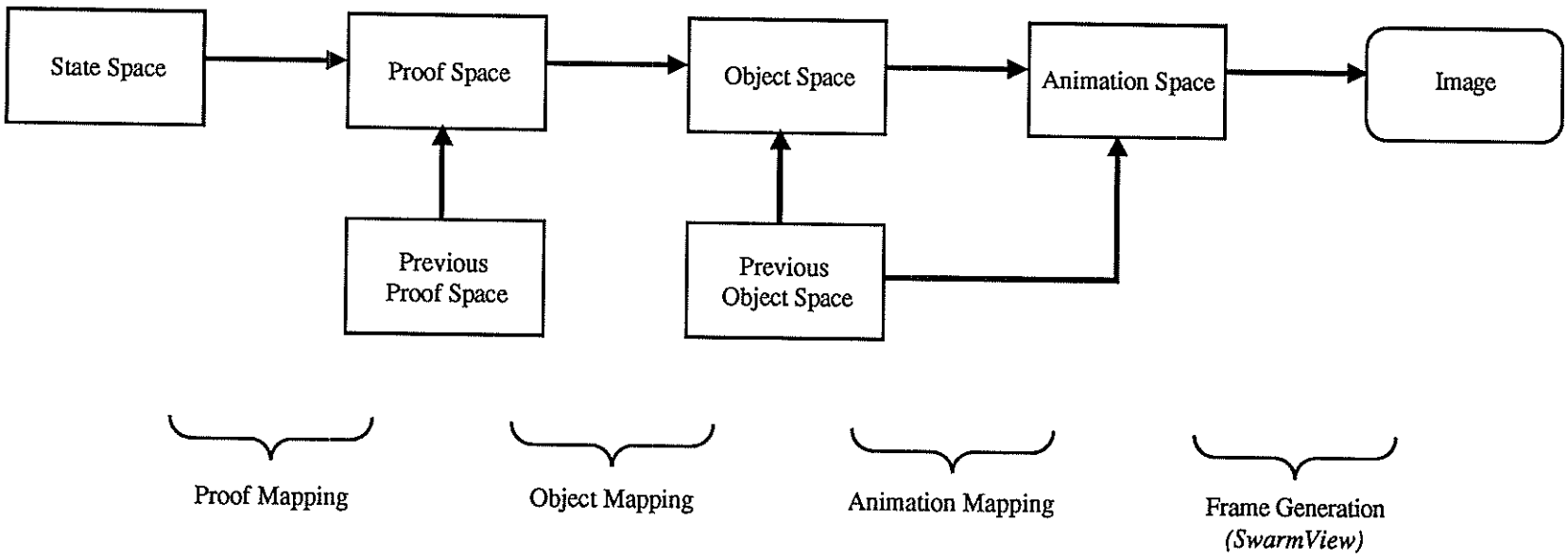

Figure 1. Relations among model spaces and mappings.

Computations under the Swarm model begin with an initial dataspace containing data tuples and transaction tuples. Computation proceeds by selecting one of the synchronic groups in a non-deterministic but fair fashion and executing the transactions which are contained within the group in parallel. When a transaction is executed, the tuple representing it is removed from the dataspace and the corresponding transaction specification is performed. The subtransactions making up the transaction are executed in parallel (and thus all subtransactions in the synchronic group are executed in parallel). Each subtransaction attempts to match the pattern in its query against the dataspace. If the match is successful, the action part of the subtransaction is executed and modifies the dataspace; if it is unsuccessful, the subtransaction takes no action. Selection and execution of synchronic continues until no more transaction tuples are present in the dataspace.

Swarm's greatest advantage for visualization is that all the information about the computational state is contained in a single, uniform, easily-accessed form. Issues of extracting the computational state from the running computation are thus minimized, allowing us to focus our efforts on the development of the model.

3 Model Spaces

The model has four spaces, the *state*, *proof*, *object*, and *animation* spaces. The state space is part of the underlying computation; we assume it to be organized as a set of tuples which changes by a series of atomic transitions. The remaining three spaces are collectively called the model spaces. Two separate instances of the proof and object spaces, the current space and the previous space, are maintained by the model and can be accessed by the mappings.

Each of the model spaces is a set of tuples having the syntactic form

typename$(component_1, component_2, \ldots, component_n)$.

In the proof and object spaces, the *typename* is an identifier chosen by the visualization designer; it need not have any particular meaning, but of course careful choice of typenames will make the purpose of each type more apparent to the reader. The *components* are also arbitrary and may include structured data such as sets and lists.

The structure of the tuples in the animation space is somewhat different. In the animation space, the *typename* of the tuple is one of a limited set of identifiers, each associated with a primitive graphical object which can be rendered by the graphical engine. The SwarmView graphical engine provides a number of such objects, including lines, polygons, circles, and spheres. The *components* have the structure

attribute = *expression*

where the *attribute* is some property of the graphical object represented by the *typename* and the *expression* is a value to be assigned to the attribute. The expression may be a constant or a time-dependent function; the time-dependence is used to create animation in the final image. The process of translating the animation space into the final images is described in the paper "SwarmView Animation Vocabulary and Interpretation", the appendices of which are reproduced in this paper.

We make one restriction on the tuples that may be used: the typenames associated with the state space, proof space, object space, and animation spaces must be disjoint. This permits us to immediately identify the space to which a given tuple belongs.

The *proof space* is an extraction of those properties of the underlying computation which are deemed to be significant to the visualization. The term *proof space* derives from our methodological belief that the program correctness properties describing the behavior of the program serve as a basis for determining which properties of the program should be visualized. Because it is often the case that proof rely on the use of auxiliary variables which maintain historical information, the proof rules need to access the old value of the auxiliary variables – in other words, the previous proof space – in order to compute the

new ones. We therefore permit the rules creating the proof space to examine both the state space and the previous proof space.

The proof space and associated proof mapping also embody the concept of state collection. In our model, we are not overly concerned with how information about the underlying computation is collected and made available to the superimposed visualization (indeed, we selected Swarm as a computational model precisely for this reason). However, this issue is of great concern in visualization implementations; program state information must be collected and relayed to the visualization process. The proof mapping indicates what information must be collected and how it should be presented.

The *object space* is an abstraction in graphical form of the properties contained in the proof space. The tuples in the object space do not directly represent graphical objects in the final image – that is the function of the animation space – but instead embody abstract objects that the visualization designer finds most convenient. A single tuple of the object space may generate several objects in the final image (for example, an object-space tuple representing a cube might be rendered using six rectangles), or an object in the image might use information from several tuples in the object space.

We permit the rules generating the object space to examine both the proof space and the previous object space. This ability has a number of applications. The most important is the ability to store geometric information in the object space and access this information, either when creating the new object space or when creating the animation space. For example, we might logically identify certain object in the object space by a number. We would then store a function mapping the identification numbers to geometric coordinates; the function would be stored as a collection of tuples. The function is preserved by rules which copy the tuples from the previous object space to the new one. This ability can also be utilized to facilitate implementation of certain types of viewer interaction with the image. The viewer might decide, for example, to move one of the graphical objects in the final image; the object then should remain in its new location. If the coordinates of the object are stored as suggested above, this is easily accomplished; moving the object would change the corresponding tuple of the stored function, and the new position will be preserved because the new tuple value is copied.

The *animation space* is the final graphical representation of the program visualization. The tuples in the animation space are in one-to-one correspondence with the graphical objects making up the final image. The animation space is the only point in the visualization process where we recognize the concept of an event; the animation mapping is permitted to examine the current and previous object spaces, detect changes between these spaces, and animate the changes appropriately. This ability is incorporated to permit smooth transitions between the graphical representations of two successive state spaces. Animation was not incorporated in an earlier version of our model, and the resulting animations suffered from discontinuities which made them difficult to follow.

Each tuple in the object space represents a single *primitive graphical event* in the final image. A primitive graphical event or *PGE* is a description of the attributes of a single primitive graphical object over a short span of time. Animation is achieved by making the attributes of the object time-dependent; for example, to make an object move from one location to the other we give the positional attribute of the object a value which changes with time. The graphical engine takes the tuples of a single animation space and transforms them into a series of *frames*. Each frame is a single image representing the results of evaluating the animation space tuples at a particular time.

4 Mappings

The mappings of the visualization model specify relations between the various spaces. Each mapping has an *input space* (actually consisting of two spaces in all of the model's mappings) and an *output space*. The mapping specifies relations between the input and output spaces using rules. Each rule is of the general form "given that some property Q is true of the input space, the tuples in the list P will be present in the output space".

4.1 Notation and Terminology

A visualization rule has the general form

$$v : Q \Rightarrow P$$

where *v*is a collection of variables, Q is a query over the rule's input space and P is a list of patterns from the rule's output space.

The query Q consists of a comma-separated list of predicates and tuple patterns. Predicates are any logical expression which is either true or false; the predicate may test the values of the variables. Tuple patterns have the same notation as tuples: a *typename* and a list of *components* which may include variables. The typename of the tuple must be associated with one of the input spaces of the rule. Except in one case, the space to which a tuple pattern appearing in a query belongs can be deduced from the tuple's type and the mapping to which the rule belongs (i.e., a tuple appearing the query part of a rule from the proof mapping must belong to either the state space or the previous proof space, and since the typenames are disjoint between spaces we can determine the space by inspection). The sole exception is in the case of the animation mapping, which accesses tuples from both the current and previous object spaces. In this case, we prefix the tuple pattern with "*old.*" to indicate the tuple is to belong to the previous instance of the object space; tuple patterns without this prefix belong to the current instance.

The production P consists of a comma-separated list of tuples. The typename of the tuples must be associated with the output space of the rule. We say a variable from *v* is *instantiated* by Q if the variable appears in Q; we require that all variables from *v* that appear in P be instantiated by Q. This restriction is common to most rule-based languages.

4.2 Formal Basis

For model purposes, we assume that any change in the input space of a mapping is instantaneously reflected in the output space. Let the configuration of spaces be represented by a 6-tuple $\langle S, P, P_p, O, O_p, A \rangle$ where S is the state space, P is the proof space, P_p is the previous proof space, O is the object space, O_p is the previous object space, and A is the animation space. Let the output space produced by a mapping M given an input space I be represented by $out(M, I)$; since in all cases I consists of two spaces, we also use the notation $out(M, \langle s1, s2 \rangle)$ Then given a particular configuration of spaces

$$\langle s, p, p_p, o, o_p, a \rangle$$

with proof mapping M_P, object mapping M_O, and animation mapping M_A, an atomic change in the state space from s to s' results in a new configuration given by

$$\langle s', p', p, o', o, a' \rangle$$

where $p' = out(M_P, \langle s', p \rangle)$, $o' = out(M_O, \langle p', o \rangle)$, and $a' = out(M_A, \langle o', o \rangle)$. That is, the new proof space is derived from the new state space and the old proof space, the existing proof space becomes the previous proof space, and so on. The first two mappings use two distinct spaces, while the third uses two instantiations of the object space; this has certain effects on the syntax of the rules as discussed above.

We now need to define the *out* function.which is used above. We start by defining *out* for rules. Consider a rule R having the form $v : Q \Rightarrow P$. 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 Q_i^v (Q with all occurrences of *v* replaced with i) represents a predicate that is either true or false for any given input space (using the convention that the presence of a tuple in Q_i^v is to be read as "the tuple Q_i^v is present in the input space), and P_i^v is a list of tuples in the output space. The set of tuples produced by the rule R for a given input space $\langle s1, s2 \rangle$ is

formed by collecting all tuples appearing in any list P_i^v, where i is any instantiation of v such that the predicate Q_i^v is true given the input space:

$$out(R, \langle s1, s2 \rangle) = \{ i : Q_i^v \text{ is true for } \langle s1, s2 \rangle : P_i^v \}$$

Given a mapping M which consists of a set of rules, the output space of M is defined as the union of the sets produced by all rules in M:

$$out(M, \langle s1, s2 \rangle) = (\cup R : R \in M : out(R, \langle s1, s2 \rangle)$$

5 Examples

Each of the subsections of this section begins with a description of some aspect of a computation to be visualized. We then determine a suitable visual representation of the properties to be captured. Finally, the three mappings are specified.

5.1 Array Summation

The underlying computation contains an array stored as a collection of tuples of the form *array(index, value)* where the *index* components are integers in the range from 0 up to the size of the array and the *value* components are positive numbers. During part of the computation, the sum of the array will be computed. The summation will be performed by selecting and deleting two distinct *array* tuples having different *index* components and inserting a new tuple whose *index* is the smaller of the two *index* components and whose *value* is the sum of the two *value* components. This can be expressed in Swarm as a transaction *Sum* whose behavior is

```
Sum() ≡
    i, j, v, w :
        array(i, v)†, array(j, w)†, i < j
    →
        array(i, v + w), Sum()
```

(The dagger (†) symbol indicates deletion of the tuple from the dataspace. The *Sum* transaction reproduces itself as long as it succeeds in finding at least two *array* tuples; when the query fails, there is no more work to be done and the transaction disappears.)

We want to visualize the correctness and progress of this simple algorithm. Specifically, we want to illustrate the following two correctness properties:

- (Invariant) The sum of *value* components over all *array* tuples is constant.
- (Progress) If the number of *array* tuples is greater than one, the number will become smaller.

We can represent the first property by constructing a bar whose length is equal to the sum of the *value* components. Invariance of the property is indicated by the length of the bar remaining constant. The second property can be visualized using the same bar, by composing the bar out of a number of sub-bars, each having a length equal to the *value* component of one of the *array* entries. The progress is indicated by a reduction in the number of sub-bars. Figure 2 illustrates the desired visualization behavior; the bars indicated are generated from several successive state spaces, reading down the page.

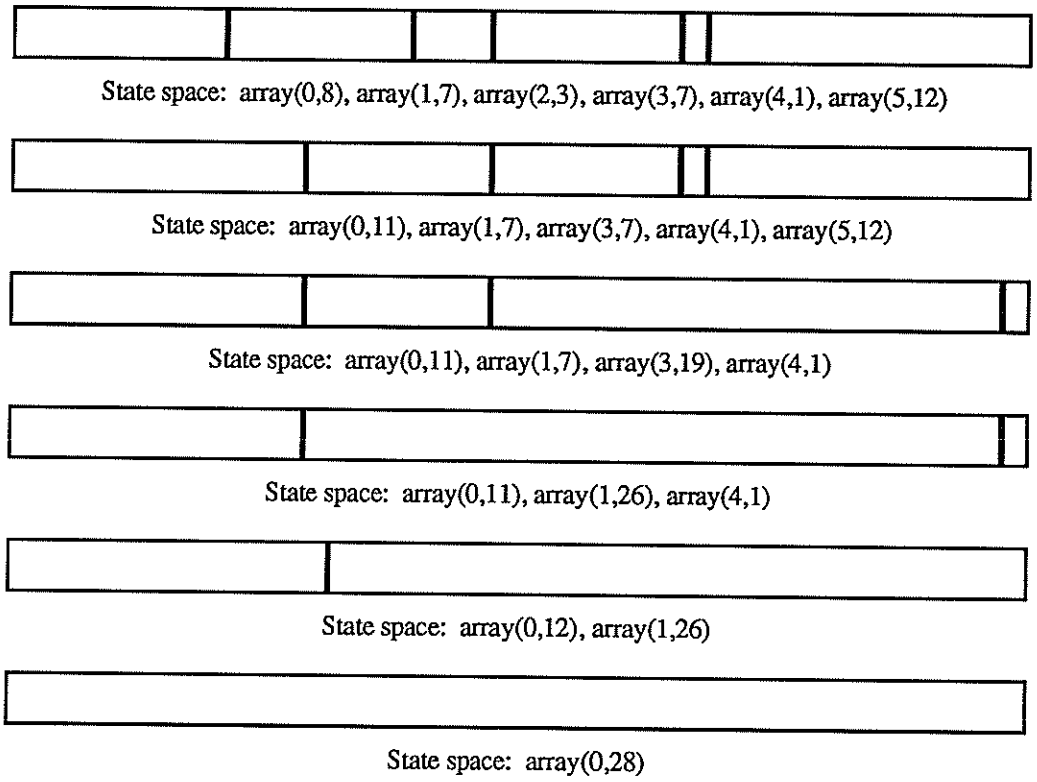

Figure 2. Successive state spaces and corresponding bar for the array-summation example.

Our proof space will contain tuples of the form *array_bar(index,value)*, one for each *array* tuple in the state space. The proof mapping rule producing this is:

```
i, v :
    array(i, v)
⇒
    array_bar(i, v)
```

Although this seems trivial, and indeed might lead one to question the need for a proof space, recall that the *array* tuples may be only a small part of the state space; indeed, there may be thousands of other tuples in many different tuple types. This proof rule clearly states that of all these tuples, the only ones we are interested in are the *array* tuples, and indicates how the data from the *array* tuples is to be made available to the visualization.

Our object space will contain tuples of the form *sub_bar(position,length)*, where *position* will be the "X-coordinate" (in some arbitrary units) of the left edge of the rectangle and *length* will be the size of the rectangle in the X-direction. The following object mapping rule produces the required tuples:

```
i, v, p :
    array_bar(i, v), p = (Σ j, w : j < i ∧ array_bar(j, w) : w)
⇒
    sub_bar(p,v)
```

The length of each *sub_bar* is just the *value* of the corresponding *array_bar*, while the position is computed by summing the *values* of the *array_bars* with smaller *index* components. Note that this means that each time the underlying computation takes a single atomic action, the superimposed visualization must do an

amount of work equal to that being performed by the array summation itself. This is unfortunately common in visualization; even with specialized hardware, the generation and display of graphics is a lengthy process and results in a slowdown of the underlying computation.

Our animation space will contain *rectangle* tuples, one for each *array_bar*. No special animations are required; however, we do want to transform the arbitrary coordinate system of the *array_bar* tuples into a system that is appropriate for our display. For this reason, we return to the object space and include a tuple of the form *bar_position(xpos,ypos,xscale,yscale)* which will be used to position and size the *rectangle* tuples of the animation space. The initial values stored in the *bar_position* tuple are chosen appropriately by the designer of the visualization, and an object mapping rule is included to propagate the tuple:

```
xp, yp, xs, ys :
    bar_position(xp, yp, xs, ys)
⇒
    bar_position(xp, yp, xs, ys)
```

In this rule, the *bar_position* pattern on the query side refers to a test for the tuple in the previous instance of the object space; the tuple on the production side refers to production of the tuple in the new instance of the object space. This can be determined by examination, since the rule is known to be an object mapping rule.

Returning to the animation space, we want to produce the *rectangle* tuples. Referring to the "library" of graphical objects (see the appendices, which are reproduced from the paper "SwarmView Animation Vocabulary and Interpretation") we find that the *rectangle* tuples require the center of the rectangle (in XYZ coordinates) and the X and Y sizes; we will use the default values of the other attributes. The *rectangles* are produced by the animation mapping rule:

```
xp, yp, xs, ys, pos, len :
    bar_position(xp, yp, xs, ys), sub_bar(pos, len)
⇒
    rectangle(    position = [ xp + xs * (pos + len / 2), yp – ys / 2, 0 ],
                  xsize = xs * len,
                  ysize = ys )
```

The *rectangle*'s X-size is obtained by multiplying the *sub_bar*'s length by the scaling factor stored in the *bar_position* tuple and its Y-size is that stored in the *bar_position* tuple. The position of the center is calculated in a similar fashion.

5.2 Communication Monitoring

The underlying computation consists of a number of processes which communicate by passing messages over channels. The processes are interconnected in a toroidal configuration of some known size M by N, and each process is identified by a pair [x,y] (with $0 \le x < M$ and $0 \le y < N$) of "coordinates" which give its position in the torus (Figure 3). Message-passing occurs by means of a rendezvous during which one process sends a message and another receives it. The message-passing protocol is such that the sending process blocks until the message is received, and thus at most one message can be "in transit" on any channel at any time. A rendezvous begins when process P_1 attempts to send a message to process P_2; this is indicated by the presence of a tuple *sending(P_1, P_2)* in the state space. When process P_2 begins receiving the message a tuple *receiving(P_2, P_1)* is present. When the rendezvous completes both tuples are removed from the state space simultaneously (i.e., as part of a single atomic change in the state space).

We wish to provide a visual monitor of the communication behavior of this computation. No correctness properties are provided – we are not observing a concurrent algorithm to gain understanding of its behavior, but instead are simply examining the message traffic. We therefore will use an ad-hoc approach to constructing the visualization.

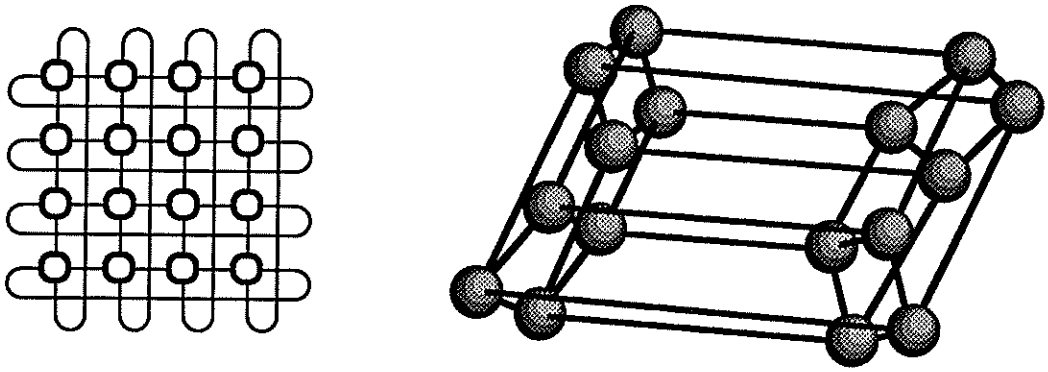

Figure 3. A 4 × 4 toroidal configuration of processes.
Left – Two-dimensional representation, with the processes arranged in an array by [x, y] identifiers.
Right – Three-dimensional representation of the same configuration.

We will generate a three-dimensional representation as illustrated on the right of Figure 3, representing the processes by spheres and the channels by lines connecting the spheres. Process state information will be indicated by colors – for example, a non-communicating process will be blue, a process that is waiting to send will be red, a process that is sending will be green, and a process that is receiving will be yellow. These colors are chosen quite arbitrarily, and it is possible that they would prove difficult to interpret or unattractive. For this reason, we will use the technique of storing the colors in the object space where they can be easily changed. We will use a similar color technique to represent channel activity; an inactive channel will be gray and an active channel will be white.

Our proof space will contain tuples of the form *process_state(id, state, partner)* for each communicating process. The *id* is the process' identifier; the *state* is one of *blocked*, *send*, or *receive*; and the *partner* is the identifier of the process with which the *id* process is communicating. The following proof mapping rules generate the needed tuples:

id, pid :
    sending(id, pid), receiving(pid, id)
⇒
    process_state(id, send, pid), process_state(pid, receive, id)

id, pid :
    sending(id, pid), ¬receiving(pid, id)
⇒
    process_state(id, blocked, pid)

The object space tuples represent both the processes and communications links; further, we need to have tuples for all processes and links, not just those involved in communication. There are two ways to generate the needed information about the topology. The first would be to use the facts that the processes are toroidally-connected and process identifiers are pairs of integers [x,y] with $0 \le x < M$ and $0 \le y < N$, where M and N are the torus size. By using several rules we can generate the required information; for example, one rule to detect an inactive channel and generate the needed tuple would be

M, N, x, y :
    torus_size(M, N), $0 \le x < M$, $0 \le y < N$,
    ¬process_state([x,y], send,[(x+1) **mod** M, y]),
    ¬process_state([x,y], receive,[(x+1) **mod** M, y])
⇒
    link([x,y], [(x+1) **mod** M,y])

This approach has a number of disadvantages; most obviously, the rules are rather long. A more significant disadvantage is the lack of flexibility. The above rule is suitable for a toroidal topology, but completely inapplicable to other topologies. For these reasons, the second approach is preferable. In this approach, we store the needed information about the topology in either the proof space or the object space. This approach is used below, where we assume proof-space tuples *process_set(S)* and *channel_set(S)* are used to store the set of all processes and the set of all channels. (The topography could be stored in other ways, depending on both personal preference and efficiency considerations.)

We also select the color of the nodes and links in this mapping. The colors are stored as tuples in the object space (and copied by appropriate rules). We generate tuples of the form *node(id, color)* and *link(id_1, id_2, color)*. The following rules generate the object space:

```
id, pid, bc :
    process_state(id, blocked, pid), blocked_color(bc)
⇒
    node(id, bc)

id, pid, sc, ac:
    process_state(id, send, pid), send_color(sc), active_chan_color(ac)
⇒
    node(id, sc), link(id, pid, ac)

id, pid, rc, ac:
    process_state(id, send, pid), receive_color(rc)
⇒
    node(id, rc)

PS, id, nc:
    process_set(PS), id ∈ PS,
    ¬(∃ pid : pid ∈ PS : process_state(id, blocked, pid) ∨
                         process_state(id, send, pid) ∨
                         process_state(id, receive, pid) ),
    noncom_color(nc)
⇒
    node(id, nc)

PC, id, pid, ic :
    channel_set(PC), [id, pid] ∈ PC,
    ¬process_state(id, send, pid), ¬process_state(pid, send, id),
    ¬process_state(id, receive, pid), ¬process_state(pid, receive, id),
    inactive_chan_color(ic)
⇒
    link([id,pid], ic)
```

The fourth rule may require some notational clarification. It states, "if a process *id* is in the process set, and it is not the case that there exists another process *pid* such that *id* is blocked, sending, or receiving in a communication with *pid*, then *id* must be non-communicating". Also note that active channels are identified in the second rule; they could also be identified in the third, or even both (since the spaces are sets, it does not matter if a particular tuple is generated multiple times).

The animation-space tuples are easily created from the object-space tuples. We need to position the various graphical objects; again, there are two approaches. We can store (in the object space) tuples which serve to map process identifiers to three-dimensional coordinates. Alternatively, we can store information about the torus as a whole (X and Y center, radius, etc.) and use this information to generate the coordinates. Both approaches have advantages and disadvantages. The first is more flexible (any topology can be so represented) but requires considerable initialization and does not permit the torus to be treated as a single object. The second allows the torus to be treated as a single object, but does not allow

the individual elements of the torus to be treated separately. The rules below use the second approach, in which a tuple

```
torus_configuration(M, N,  xcenter, ycenter, zcenter, radius, thickness, sphere_rad)
```

stores the information used to generate the torus. The expressions involved map the [x,y] pairs of process identifiers to three-dimensional coordinates.

```
M, N, xc, yc, zc, r, t, sr, x, y, c :
    torus_configuration(M, N, xc, yc, zc, r, t, sr), node([x, y], c)
⇒
    sphere(  center = [ xc + (r + t * cos(360*y/N)) * cos(360*x/M),
                        yc + (r + t * cos(360*y/N)) * sin(360*x/M),
                        zc + t * sin(360*y/N) ],
             radius = sr,
             color = c)

M, N, xc, yc, zc, r, t, sr, x1, y1, x2, y2, c :
    torus_configuration(M, N, xc, yc, zc, r, t, sr), link([x1, y1], [x2, y2], c)
⇒
    line(    from = [ xc + (r + t * cos(360*y1/N)) * cos(360*x1/M),
                      yc + (r + t * cos(360*y1/N)) * sin(360*x1/M),
                      zc + t * sin(360*y1/N) ],
             to = [ xc + (r + t * cos(360*y2/N)) * cos(360*x2/M),
                      yc + (r + t * cos(360*y2/N)) * sin(360*x2/M),
                      zc + t * sin(360*y2/N) ],
             color = c)
```

(Clearly it would be desirable to add some sort of function definition, or at least a macro facility, to any implementation in order to avoid expressions such as the above.)

5.3 An Extension

Assume that we want to make the visualization of the previous subsection a little more interesting. Let us say that we want to animate the act of message-passing by having a small "zap", colored differently from the main link, transfer from the sender to the receiver during the communication, as illustrated on the left in Figure 4.

Most of the rules of the previous mapping remain the same (one of the advantages to a declarative approach to visualization). The first change is in the object mapping, where we identify those links where the effect is to occur. Since these are exactly those links which correspond to active channels, we only need to make a minor change to the second object mapping rule from the previous subsection:

```
id, pid, sc, ac, zc:
    process_state(id, send, pid), send_color(sc), active_chan_color(ac,zc)
⇒
    node(id, sc), zap_link(id, pid, ac, zc)
```

As before, we assume that the desired colors are stored in the object space and copied by appropriate rules. In this case, there are two colors corresponding to the active channel color and the color of the zap.

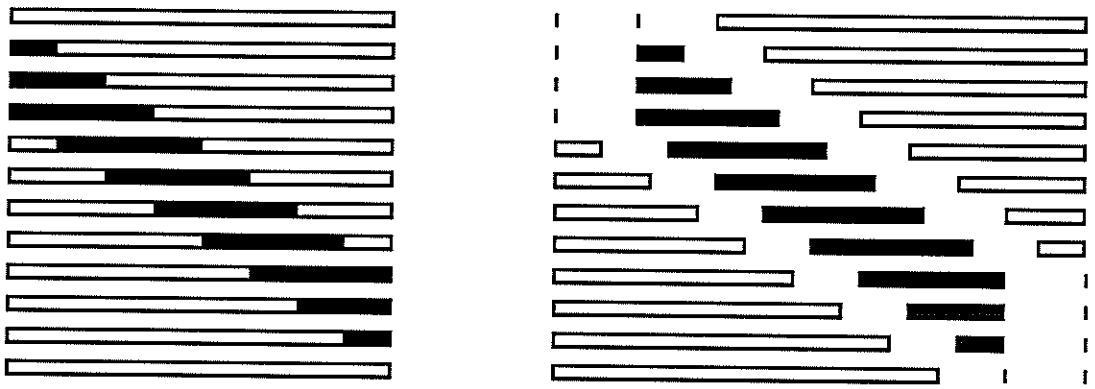

Figure 4. Illustration of a "zap" animation, with motion from left to right (time increases down the page).
Left – Desired effect.
Right – Breakdown of "zap" into three separate lines.

The next stage is to determine how to generate the zap animation. The basic problem is, given the two endpoints, to generate a line with the time-changing pattern illustrated in Figure 4. The obvious way, generating the entire line in the active color then "overwriting" part of it in the zap color, does not work. The order in which the animation tuples are interpreted and produced is deliberately unspecified, so the interpreter could generate the zap color part then "overwrite" it with the active color part.

One method which works is illustrated to the right of Figure 4, where the link is decomposed into three lines – two in the active color and one in the zap color – which together make up the whole pattern. Examining the endpoints of each line, it is clear that we need to smoothly change the endpoints during the course of the animation. The *ramp* function provided by the animation vocabulary is ideal. We also need to make certain timing decisions, specifically when the various ramp functions begin and end. As in the case of geometric constants, these times should be stored in a tuple in the object space and retrieved as required. The rule which generates the desired animation tuples is:

```
M, N, xc, yc, zc, r, t, sr, x1, y1, x2, y2, ac, zc, P1, P2:
    torus_configuration(M, N, xc, yc, zc, r, t, sr),
    zap_link([x1, y1], [x2, y2], ac, zc),
    P1 = [ xc + (r + t * cos(360*y1/N)) * cos(360*x1/M),
         yc + (r + t * cos(360*y1/N)) * sin(360*x1/M),
         zc + t * sin(360*y1/N) ],
    P2 = [ xc + (r + t * cos(360*y2/N)) * cos(360*x2/M),
         yc + (r + t * cos(360*y2/N)) * sin(360*x2/M),
         zc + t * sin(360*y2/N) ]
    zap_times(start_leading, start_trailing, end_leading, end_trailing)
⇒
    line(   from = P1,
            to = ramp(start_trailing, P1, end_trailing, P2),
            color = ac),
    line(   from = ramp(start_trailing,P1,end_trailing,P2),
            to = ramp(start_leading, P1, end_leading, P2),
            color = zc),
    line(   from = ramp(start_leading, P1, end_leading, P2),
            to = P2,
            color = ac)
```

If we assume the lines in Figure 4 represent successive frames of animation, then appropriate values for the times would be: *start_leading* = 0, *start_trailing* = 3, *end_leading* = 8, and *end_trailing* = 11.

6 Acknowledgments

The author would like to thank Dr. Jerome R. Cox of the Department of Computer Science at Washington University for his support. This research was supported in part by the National Fellowship Program in Parallel Processing, supported by DARPA/NASA and administered by the University of Maryland Institute for Advanced Computer Studies (UMIACS).

Bibliography

Visualization surveys, position papers, and important systems:

Brown, M. H. and Sedgewick, R., "Techniques for Algorithm Animation", *IEEE Software*, Vol. 2 No. 1, pp. 28-39 (January 1985).

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

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

Chang, S.-K., Ichikawa, T., and Ligomenides, P. A. (editors), *Visual Languages* (Plenum Press: New York, 1986).

Chang, S.-K., "Visual Languages: A Tutorial and Survey," *IEEE Software*, Vol. 4 No. 1, pp. 29-39, January 1987.

Myers, B. A., "Visual Programming, Programming by Example, and Program Visualization: A Taxonomy," *Human Factors in Computing Systems: Proceedings SIGCHI'86*, pp. 59-66 (April, 1986).

Raeder, G., "A Survey of Current Graphical Programming Techniques," *IEEE Computer*, Vol. 18, No. 8, pp. 11-25 (August 1985).

Shu, N. C., *Visual Programming* (Van Nostrand Reinhold Company: New York, 1988).

Gorny, E. P., and Tauber, M. J. (editors), *Visualization in Programming* (Springer-Verlag: New York, 1987).

Research on visualization of concurrent computations at Washington University:

Cox, K. C., *Visualization of Concurrent Computations* (Doctor of Science Dissertation Proposal), Technical Report WUCS-89-32, Department of Computer Science, Washington University in St. Louis (June, 1989).

Cox, K. C. and Roman, G.-C., "Visualizing Concurrent Computations", Technical Report WUCS-90-31, Department of Computer Science, Washington University in St. Louis, September 1990. Submitted to the *13th International Conference on Software Engineering*.

Cox, K. C., "Visualization in Concurrent Contexts: A Model", Technical Report WUCS-91-7, Department of Computer Science, Washington University in St. Louis, November 1990.

Cox, K. C., Wilcox, C. D., and Plun, J. Y., "SwarmExec: A Prolog-Based Execution Engine for a Shared-Dataspace Language with Visualization Capabilities", Technical Report WUCS-91-8, Department of Computer Science, Washington University in St. Louis, December 1990.

Cox, K. C., "SwarmView: A Graphical Engine for the Interpretation and Display of Visualizations", Technical Report WUCS-91-9, Department of Computer Science, Washington University in St. Louis, January 1991.

Cox, K. C., "SwarmView Animation Vocabulary and Interpretation", Technical Report WUCS-91-10, Department of Computer Science, Washington University in St. Louis, November 1990.

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

Roman, G.-C. and Cox, K., "Declarative Visualization in the Shared Dataspace Paradigm", *Proceedings of the 11th International Conference on Software Engineering* (May 1989).

Swarm notation and proof system:

Cunningham, H. C., *The Shared Dataspace Approach to Concurrent Computation: The Swarm Programming Model, Notation, and Logic*, Doctor of Science Dissertation, Department of Computer Science, Washington University in St. Louis, August, 1989.

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).

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).