Report Number: WUCS-91-26

1991-04-01

Pavane: A System for Declarative Visualization of Concurrent Computations

Gruia-Catalin Roman, Kenneth C. Cox, C. Donald Wilcox, and Jerome Y. Plum

This paper describes the conceptual model, specification method, and visualization methodology for Pavane-- a visualization environment concerned with exploring, monitoring, and presenting concurrent computations. The underlying visualization model is declarative in the sense that visualization is treated as a mapping from program states to a three-dimensional world of geometric objects. The latter is rendered in full color and may be examined freely by a viewer who is allowed to navigate through the geometric world. The state-to-geometry mapping is defined as a composition of several simpler mappings. The choice is determined by methodological and architectural considerations. This paper shows how... Read complete abstract on page 2.

Recommended Citation

Roman, Gruia-Catalin; Cox, Kenneth C.; Wilcox, C. Donald; and Plum, Jerome Y., "Pavane: A System for Declarative Visualization of Concurrent Computations" Report Number: WUCS-91-26 (1991). *All Computer Science and Engineering Research.*
https://openscholarship.wustl.edu/cse_research/644

Pavane: A System for Declarative Visualization of Concurrent Computations

Gruia-Catalin Roman, Kenneth C. Cox, C. Donald Wilcox, and Jerome Y. Plum

Complete Abstract:

This paper describes the conceptual model, specification method, and visualization methodology for Pavane-- a visualization environment concerned with exploring, monitoring, and presenting concurrent computations. The underlying visualization model is declarative in the sense that visualization is treated as a mapping from program states to a three-dimensional world of geometric objects. The latter is rendered in full color and may be examined freely by a viewer who is allowed to navigate through the geometric world. The state-to-geometry mapping is defined as a composition of several simpler mappings. The choice is determined by methodological and architectural considerations. This paper shows how this decomposition was molded by two methodological objectives: (1) the desire to visually capture abstract formal properties of programs (e.g. safety and progress) rather than operational details and (2) the need to support complex animations of atomic computational events. All mappings are specified using a rule-based notation; rules may be added, delated, and modified at any time during the visualization. An algorithm for termination detection in diffusing computations is used to illustrate the specification method and to demonstrate its conceptual elegance and flexibility. A concurrent version of a popular artificial intelligence program provides a vehicle for demonstrating how we derive graphical representations and animation scenarios from key formal properties of the program, i.e., from those safety and progress assertions about the program which turn out to be important in verifying its correctness.

Pavane: A System for Declarative Visualization of Concurrent Computations

Gruia-Catalin Roman
Kenneth C. Cox
C. Donald Wilcox
Jerome Y. Plun

WUCS-91-26

April 1991

Appeared in *Journal of Visual Languages and Computing*, 3, pp 161-193, 1992

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

Abstract

This paper describes the conceptual model, specification method, and visualization methodology for *Pavane*—a visualization environment concerned with exploring, monitoring, and presenting concurrent computations. The underlying visualization model is declarative in the sense that visualization is treated as a mapping from program states to a three-dimensional world of geometric objects. The latter is rendered in full color and may be examined freely by a viewer who is allowed to navigate through the geometric world. The state-to-geometry mapping is defined as a composition of several simpler mappings. The choice is determined by methodological and architectural considerations. This paper shows how this decomposition was molded by two methodological objectives: (1) the desire to visually capture abstract formal properties of programs (e.g., safety and progress) rather than operational details and (2) the need to support complex animations of atomic computational events. All mappings are specified using a rule-based notation; rules may be added, deleted, and modified at any time during the visualization. An algorithm for termination detection in diffusing computations is used to illustrate the specification method and to demonstrate its conceptual elegance and flexibility. A concurrent version of a popular artificial intelligence program provides a vehicle for demonstrating how we derive graphical representations and animation scenarios from key formal properties of the program, i.e., from those safety and progress assertions about the program which turn out to be important in verifying its correctness.

Acknowledgments: This paper is based on work supported by the National Science Foundation under the Grant CCR-9015677 and by the DARPA/NASA National Fellowship Program in Parallel Processing under agreement 26947M. The Government has certain rights in this material. The authors acknowledge Timothy J. Boemker's collaboration in the production of the *Diffusing Computations* video tape.

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 [6]. 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 keys to the success of scientific visualization. [6]

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 of 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 visualization of concurrency 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 as 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 current research on program visualization is centered around the development of a system called *Pavane*. Its goal is to facilitate exploration and validation of methodologies for visualizing concurrent computations. Several design requirements played an important role in shaping the conceptional model on which Pavane is based.

(1) A broad class of concurrent computations had to be specifiable in a language that has an associated assertional-style proof logic. Pavane uses the language Swarm [9], a computational model which extends UNITY [2] and its proof logic to include content-based accessing to data, dynamic statements, and dynamic partial synchrony. Swarm is particularly attractive due to its simple and uniform representation of both data and control states: a single set of tuples called the dataspace.

(2) Visualizations had to be easily specified and altered. This is accomplished by adopting the *declarative visualization* method proposed in [8]. Visualization is treated as a mapping from program states to a three-dimensional world of geometric objects. The latter is rendered in full-color and may be examined freely by a viewer who is allowed to navigate through the geometric world as it changes in response state transitions in the underlying computation. The state-to-geometry mapping may be defined as a composition of several simpler mappings. All mappings are specified using a rule-based notation; rules may be added, deleted, and modified at any time during the visualization.

(3) Real parallelism as exhibited by programs executing on multiprocessors and networks must eventually replace the simulated parallelism of the current Swarm implementation. The particular visualization mapping adopted by Pavane allows a computation which is not formulated using Swarm to supply (incrementally via Ethernet) abstract tuple-like representations of its state changes without losing the elegance and flexibility of declarative

visualization. Moreover, the mapping is decomposed in such a way so as to facilitate a future high-performance pipelined implementation of the visualization process.

(4) Full animation capabilities had to be provided without sacrificing the advantages of declarative visualization. At the time we first proposed treating visualizations as mappings from program states to geometric objects, it was not at all obvious how one might accommodate the animation of simple events (e.g., highlighting of objects involved in some state transition, smooth movements from one state representation to the next). Pavane preserves the notion of state to object mapping but permits the detection of visual events (changes in the configuration of the world of geometric objects) which may be mapped into animation sequences by employing the same rule-based notation.

The requirement for an assertional-style proof logic is motivated by our desire to place the process of constructing the visualization of a given concurrent program on some solid technical foundation. The particular approach we are currently exploring could be characterized in general terms as a *proof-based visualization*—the visualization focuses on abstract formal properties of the computation and not on structural or operational properties of the program. 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 [8] 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.

The remainder of this paper examines Pavane, its design, and the visualization methodology it supports with the help of two sample programs: one for detecting termination in diffusing computations and another for bagging groceries in accordance with a prescribed set of rules. Section 2 introduces the diffusing computations program, some Swarm notation required to capture the program state, and several important formal properties of the algorithm. Section 3 describes the process of specifying a complete visualization for the diffusing computations program as the composition of a series of simple mappings whose domains and ranges are sets of tuples. Special attention is given to highlighting the methodological considerations that led to the particular mappings adopted by Pavane. Section 4 introduces the bagger program and explores the heuristics we developed for deriving graphical representations and animation scenarios from key formal properties of the program, i.e., from those safety and progress assertions which turn out to be important in verifying its correctness. Section 5 discusses the design of Pavane and its current capabilities. A summary and 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. 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, cites articles that discuss Swarm [9] 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 representing passive data; the *transaction space*, a finite set of *transactions* representing program actions; and the *synchrony relation*, a mechanism for specifying that selected actions are to be executed either synchronously or asynchronously (a feature unique to Swarm). Every element of the dataspace has a 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 single transaction is modeled as a transition between dataspaces. The 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. (A synchronic group consists of all transactions present in the dataspace and related to each other via the reflexive transitive closure of the synchrony relation.) 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 and Scholten [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 acknowledgment 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 acknowledgments for every message sent. The correctness of this solution relies on the following two invariants:

(1) Considering active processes as vertices and parent process information as directed arcs from child to parent, 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 acknowledgments 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 the solution above is defined by the local state of the environment, of each process, and of each 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 parent process; by convention, we make the parent process of an inactive transaction the environment (this has no effect on the computation). 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 acknowledgments 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 (acknowledgment) and the number of messages (acknowledgments) in transit. In Swarm, the four tuple types introduced so far are declared as follows:

```
environ( messages_to_send, pending_acks );
process ( id, status, pending_acks, parent, );
msg( source , destination , in_transit );
ack( source, destination, in_transit )
```

All the tuple components are natural numbers except the *process id*, which is a natural in the range 0 to N, and the *status*, which is a member of 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 acknowledgment 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 acknowledgments 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 transaction instances: *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 acknowledgments 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, 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, 0 ), ack( I, p, k+1 )
     ||   TRUE
               →  PassAck( I )
```

TRUE is one of a class of *special query types* which may be used; **TRUE** is a query which always succeeds, so each of the above transactions re-creates itself when it is executed. Some of the other special query types succeed or fail depending on the success or failure of the other subtransactions of the synchronic group; for example, the special query **NOR** succeeds if and only if none of the other subtransactions succeed.

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, inactive, 0, 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

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

i.e., that the absence of the *Env()* transaction in the dataspace indicates that the computation has terminated. In this particular case, the Swarm simulation of the computation continues (but does not change the dataspace) since the *SendMsg*, *GetMsg*, and *PassAck* transactions are still present in the dataspace.

3. Specifying Visualizations

In systems like BALSA [1] and Cedar's Real-Time Visualization system [11], visualizations are built by identifying in the program code places where significant events occur and by annotating the code with calls to procedures that manage the visualization. In effect, these procedures provide a low level constructive method for mapping computational events into complex sequences of visual events. By contrast, visualizations in Pavane are specified abstractly, as mappings from computational states to images on the screen. This *declarative* approach to visualization is attractive because visualizations are specified and modified easily and because visualization is decoupled from the program code. Pavane is not the first system to employ declarative visualization. PROVIDE [7] and PVS [5], for instance, make use of simple mappings from variables to icon attributes. Difficulties associated with specifying formally and elegantly the state of programs written in traditional block structured languages may explain why declarative methods have failed to play (so far) a significant role in the visualization community.

Because Swarm combines expressive power with a simple uniform representation of the program state (tuple-like entities stored in the dataspace), Pavane is the first system to exploit to the fullest extent the power of declarative visualization. However, the potentially complex mappings appearing in Pavane require the use of a compositional approach to defining mappings, i.e., a complex mapping is given as a composition of several simpler ones. Besides helping manage complexity, compositional approaches facilitate the formulation and refinement of visualization methodologies (as shown in the remainder of this section) and are expected to play a key role in constructing high-performance systems for program visualization. This could be accomplished by restructuring and restricting the mappings in a manner that is best suited for a particular underlying architecture (e.g., pipelining, dataflow, etc.).

In order to retain the advantages of a single uniform representation, all the mappings used in Pavane are defined over sets of tuples called *spaces*. The mappings used in Pavane fall in one of three categories: simple, history sensitive, and differential mappings. A simple mapping is a function which given an input space produces an output space. A history sensitive mapping computes the output space by examining both its (primary) input space and the previous version of its output space. This enables one to accumulate historical data, e.g., count the number of passes through the visualization mapping. Finally, a differential mapping is one that computes the output space by examining the current and the previous configuration of the input space. This allows one to check for changes, i.e., to detect events, if necessary.

Mappings are combined by identifying the output space of one as the primary input space for the next. Since each mapping has a single primary input space and a single output space, the result is a linear application of a series of mappings originating with the Swarm dataspace and ending in images on the screen. Conceptually, each atomic transformation of the dataspace, i.e., computational event, triggers an instantaneous recomputation of the entire mapping, leading to appropriate changes to the image. In order to accomplish this, a visualization system which is not fast enough to keep up with the computation must slow down the computation using some control flow protocol. This is possible on experimental platforms such as Pavane but it is not reasonable on a production system. The latter needs to employ high performance implementations (e.g., specialized hardware, pipelined algorithms) and must reduce the amount of information presented on the screen by exploiting the abstraction capabilities of state-based declarative visualization.

We specify each mapping as a collection of rules. For a simple mapping, each rule may be seen as a logical relationship between the input and output spaces given in terms of a predicate Q(v) over the input space and a predicate P(v) over the output space, with the restriction that P(v) is simply a test whether or not a specific list of tuples is present in the output space:

$$\forall v : Q(v) \Rightarrow P(v)$$

If a mapping is defined as a finite set of rules, the output space is, by definition, the smallest set satisfying all the relationships required by the set of rules. Operationally, one may treat Q(v) as a query over the input space and P(v) as a list of tuples contributed by the particular rule to the construction of the output space. The result of applying a mapping (a collection of rules) is simply the union of all sets of tuples produced by the rules of the mapping.

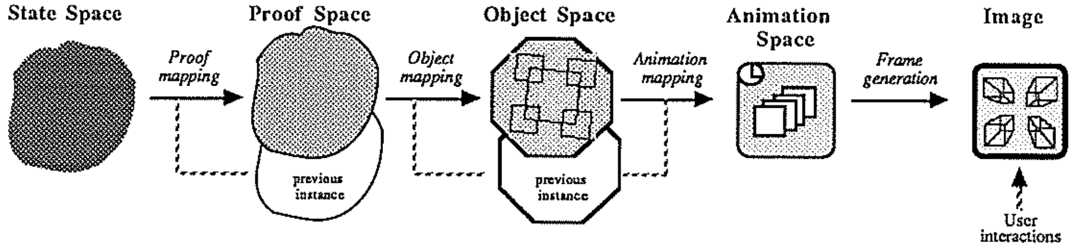


Figure 1: Visualization mapping decomposition in Pavane.

For notational convenience, the notation employed by Pavane drops the universal quantification allowing the visualization rules to be simply stated as

$$v : Q(v) \Rightarrow P(v)$$

Thus, the notation for the rules is similar to the Swarm subtransaction notation, except that ⇒ is used in the rule instead of the Swarm → to separate the query part from the list tuples produced by the rule. Despite the syntactic similarity, one must remember that subtransactions have an implicit *existential* quantification while the visualization rules have an implicit *universal* quantification.

Under the simplifying assumption that no two spaces contain tuples of the same type, no additional notation is required for history-sensitive mappings. The query (predicate) Q(v) is merely permitted to make reference to tuples in the previous output space and the tuples produced are contributed to the current output space. The distinction is made from the context, i.e., Q(v) or P(v). New notation is required, however, in differential mappings where Q(v) must refer to both the current and previous input space. We choose to prefix tuples from the previous input space configuration with the qualifier *old*.

Figure 1 depicts the visualization mapping decomposition employed in Pavane. The *state space* refers to the computational state of the program being visualized, i.e., the Swarm dataspace in the current implementation; the *proof space* is an abstraction of the program state obtained by eliminating irrelevant details and by accumulating certain historical data; the *object space* defines a three dimensional world of geometric objects; finally, the *animation space* contains graphical objects (objects which may be painted on the screen) with time-dependent attributes. The *proof mapping* and *object mapping* are history-sensitive. The *animation mapping* is differential; it detects visual events and translates them into sequences of image changes interpreted by the *frame generation mapping*. The frame generation mapping is effectively a simple mapping, but is not under the control of the composer of the visualization. In the remainder of this section we justify and illustrate this particular decomposition using the program introduced in Section 2.

3.1. The proof mapping—extraction of key properties

As a methodological foundation for visualization, we have made the assumption that those properties of the program which are important in verifying its correctness are also properties which should be visualized. We have constructed a number of visualizations using this methodology, and have found it to be quite successful as a method for constructing meaningful visualizations. Accordingly, in Pavane we refer to the first stage of visualization as the *proof mapping*. Its function is to extract from the program state only those aspects relevant to visualization. The output space of this mapping is called the *proof space*.

State extraction is obviously of great importance in program visualization. Our choice of Swarm as a computational model greatly simplifies the problem of extraction—all components of the state, both data and control, are contained in the Swarm dataspace and easily accessible. When visualizing computations which are not expressed in Swarm, some mechanism must be available for monitoring the program state and for communicating it to Pavane. In this context, the proof mapping can be considered as a *specification* of what information about the state must be extracted and how the information should be presented to later stages of the visualization.

Originally the proof mapping was intended to be a simple mapping. However, several considerations led us to choose a history-sensitive mapping. One consideration which led to this decision was the postulated proof-based methodology for constructing visualizations. It is often the case that program proofs rely on the use of auxiliary variables which maintain information about the computation. Therefore, 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.

We use as a starting point for our visualization of diffusing computations one of the key invariants required in the correctness proof: *the parent information in those process tuples whose status is active defines a directed tree rooted at the environment.* Note that at this point, the term "tree" refers to a graph-theoretic entity and not to a geometric representation. However, the correspondence between a graph-theoretic tree and its depiction is a natural one and serves as the basis for our visualization.

Not all tuples and transactions appearing in the diffusing computations program are essential to capturing this invariant. 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 describe the connectivity of the tree (edge definitions). In addition to extracting the parent-child relations from the state space, we also use the proof space to compute the distance of each node from the tree's root; this information is implicitly contained within the tree structure, but to generate a graphical representation of the tree we need to have explicit values. This illustrates another use of the proof space, the calculation of values required to support later stages of the visualization.

The connectivity and distances from the root are represented in the proof space by *vertex* tuples containing for each node in the graph 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.* 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:

```
    Env()
⇒   vertex( 0, 0, 0 )
```

Technically the environment's parent is undefined; by convention we will use 0 for the parent. The second rule creates tuples representing active processes. 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 ), vertex( p, pp, dist )
⇒   vertex( id, p, dist + 1 )
```

For every instantiation of the variables *id*, *ack*, *p*, *pp*, and *dist* such that a tuple *process(id, active, ack, p)* can be found in the current state space and a tuple *vertex(p, pp, dist)* can be found in the previous proof space, a tuple *vertex(id, p, dist + 1)* is contributed to the object space.

3.2. The object mapping—abstraction of properties

The next stage is the materialization of properties contained in the proof space into a three-dimensional geometric representation. The *object mapping* accomplishes this task, producing a collection of tuples called the *object space*. The inputs to the object mapping are the current proof space and the previous instance of the object space. The tuples of the object space represent *abstract* geometric objects. They define an abstract three-dimensional world of objects which lacks a concrete visual representation, because the abstract objects are not necessarily in one-to-one correspondence with the graphical objects that make up the final image. Each tuple in the object space may eventually be mapped to several graphical objects of the final image, or a single graphical object may be described by parameters drawn from several tuples of the object space. This flexibility is an important part of our model, as it allows the constructor of the visualization to work with concepts that are most suitable for the task, without premature commitment to a particular visual representation.

Returning to our example, we construct the tree in three dimensions. (We are now using "tree" to refer to a geometric construction.) Assume that we have some layout of the underlying communications graph in the plane—this is not to say that the graph is planar, merely that we have some arrangement of its vertices in the plane. The X- and Y-coordinates of the nodes in our three-dimensional tree are obtained from the corresponding coordinates in this graph, while the Z-coordinate corresponds to the distance of the node from the root (environment). Pavane provides mechanisms for the definition and use of functions. Assuming three functions which produce the needed coordinates:

nodex : ProcessIds → Reals
nodey : ProcessIds → Reals
nodez : Naturals → Reals

the tree is constructed using two types of components: the nodes and the links between nodes. The following rules generate these two types of abstract objects, without yet indicating how they might be represented (three-dimensional coordinates are represented by triples enclosed in brackets):

```
     vertex( 0, 0, 0 )
⇒   node_object( [ nodex(0), nodey(0), nodez(0) ] )

id, pid, dist, ppid, pdist :
     vertex( id, pid, dist ), vertex( pid, ppid, pdist )
⇒   node_object( [ nodex(id), nodey(id), nodez(dist) ] ),
     link_object( [ nodex(id), nodey(id), nodez(id) ], [ nodex(pid), nodey(pid), nodez(pdist) ] )
```

All that we currently specify is that the nodes are abstract objects associated with a particular point and the links are objects characterized by two points. The first rule generates the root node at the coordinate of the environment. The second generates a node for each non-root node (i.e., those which have a parent node) and a link between the node's coordinate and that of the node's parent.

3.3. The animation mapping—production of graphical objects

The third stage of the visualization mapping involves the translation of abstract objects produced by the object mapping into a form suitable for rendering. We call the objects which can be so rendered *primitive graphical objects*. The current Pavane "vocabulary" of primitive graphical objects includes points, lines, polygons, circles, and spheres; this vocabulary can be extended by augmenting the capabilities of the rendering engine. Production of the primitive graphical objects is accomplished by the *animation mapping* which uses the object space to produce an *animation space*. In the latter, each tuple represents exactly one primitive graphical object. Actually, the animation mapping is a differential mapping. A simple mapping cannot accommodate smooth transitions between the images representing successive states; the final result would have a jerky appearance and would be difficult to follow, as objects simply appeared and vanished in response to changes in the object space.

Full animation capability is 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 visual events. For example, 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 detect this visual event (the exchange of objects) and decompose it in a sequence of simpler visual events involving synchronized movement along some prescribed trajectories. In this manner we are able to preserve the formality of state-based mapping without sacrificing aesthetics.

To accomplish this, the tuples of the animation space represent primitive graphical objects positioned in a 4-D space, the three spatial dimensions plus time. 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*. Attributes may be time-dependent, with time being measured in terms of number of display frames starting at 0. A collection of functions are provided to generate time-dependent values. For example, the function $ramp(t_0, v_0, t_1, v_1)$ generates a linear interpolation between times t_0 and t_1, with the value of the interpolation being v_0 at time t_0 and *v1 at* time t_1 Mechanisms for composing the effects of the various functions are also provided.

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])

represents a *line* object whose endpoints are at coordinates (0,0,0) and (5,0,0). The line would be colored white (the default for that attribute).

In the diffusing computations example, we can represent each node by a sphere and each link by a line. Using a simple mapping, spheres and lines representing the tree simply appear and disappear. Two rules which accomplish this are:

```
coord :
    node_object( coord )
⇒   sphere( center = coord, radius = sphererad )

ncoord, pcoord :
    link_object( ncoord, pcoord )
⇒   line( from = ncoord, to = pcoord )
```

It should be noted that we permit the use of unification in our rules; thus, the variables in the above rules are unified with coordinates (lists of three elements). This is obviously more convenient than requiring the user to use separate variables for the X-, Y-, and Z-coordinates and provide a list of three variables as a pattern. In addition, note the use of the constant *sphererad*; Pavane permits constants to be defined and used in the same way as functions.

We can improve the appearance of the visualization by animating the addition and removal of the spheres and lines. 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 separate rules for each of these cases. Consider first the addition of a node to the tree, resulting in the addition of a sphere and (usually) a line to the image. Assume we want to animate this as follows: the line grows out from the parent sphere until it reaches the position of the new sphere; the new sphere then expands outward from a point to its full radius. This is accomplished by the following rules (recall that when using a differential mapping we distinguish between the current and the previous instances of the input space by attaching the prefix *old* for patterns that are to match the previous instance and omit the prefix on patterns that match the current instance):

```
ncoord, pcoord :
    link_object( ncoord, pcoord ), ¬old.link_object( ncoord, pcoord )
⇒   line( lifetime = [ 1, 10 ], from = pcoord, to = ramp( 1, pcoord, 5, ncoord ) )
```

```
coord :
     node_object( coord ), ¬old.node_object( coord )
⇒    sphere( lifetime = [ 5, 10 ], center = coord, radius = ramp( 5, 0.0, 10, sphererad ) )
```

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 *pcoord*); 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 *sphererad* 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, and could be represented by named constants like *sphererad.* Our ability to rapidly change the visualization by changing such constants is quite helpful in development.

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

```
ncoord, pcoord :
     link_object( ncoord, pcoord ), old.link_object( ncoord, pcoord )
⇒    line( from = pcoord, to = ncoord )

coord :
     node_object( coord ), old.node_object( coord )
⇒    sphere( center = coord, radius = sphererad )
```

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

3.4. The frame generation mapping—producing the final image

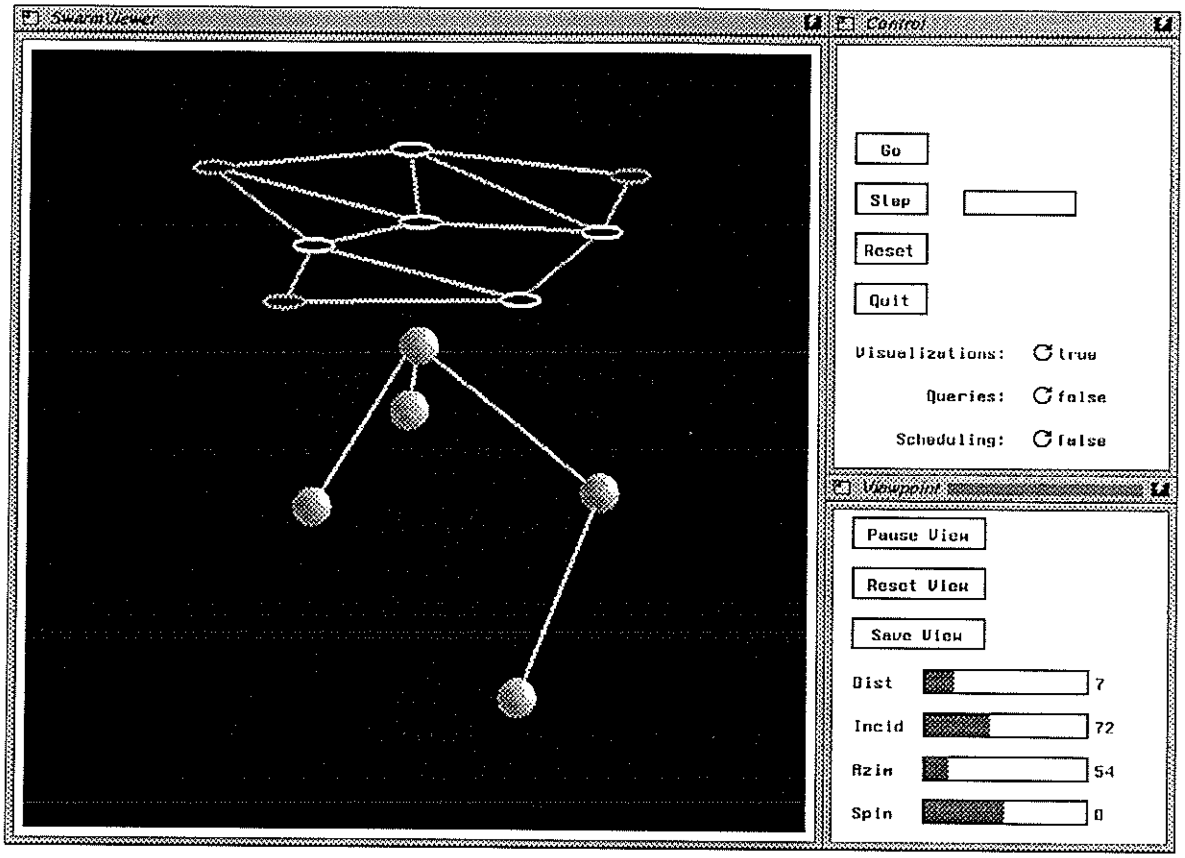
The animation tuples are interpreted and rendered by the *frame generation mapping*, which given a collection of animation tuples produces a sequence of *frames*. Each frame is a "slice" through the 4-D animation space and consists of a collection of primitive graphical objects in 3-D space (the three spatial dimensions). The frame generation mapping is performed by an interpreter which makes direct use of the rendering capabilities of some graphics workstation (the Personal Iris in our case) to convert each successive frame as an image. The rapid sequencing from one frame to another produces the animation effects.

The behavior of the frame generation mapping is not under the control of the constructor of the visualization; no rules need to be written to specify its behavior. We therefore provide only a brief discussion of the operational behavior of our particular implementation of the frame generation mapping, which we call *SwarmView*. In addition to performing the frame generation mapping, SwarmView handles user manipulation of the viewpoint, performs lighting effects, and allows recording and playback of completed animations.

Given an animation space, SwarmView first determines how many frames must 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 $[t_{min}, t_{max}]$ which indicates that the object is to be produced only between frames t_{min} and t_{max}). 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 graphical object. This is done by evaluating the value assigned to each attribute (which is either a constant or a time-dependent function) at the "current" time, i.e., the frame count. For example, the animation space tuple

```
line( from = [ 0, 0, 0 ], to = ramp( 0, [2, 2, 2], 4, [4, 6, -2] ) )
```

produces, for frames 0 through 4, the primitive graphical objects



Visualizations: ○ true

Queries: ○ false

Scheduling: ○ false

Dist 7

Incid 72

Azim 54

Spin 0

Figure 2: One frame from a visualization of the diffusing computations problem.

This frame is viewed from a position slightly above the X-Y plane; the positive Z axis is directed upward. The graphical tree representing the computation's process-activation history and termination-detection algorithm is located in the middle of the figure; the node associated with the computation's environment is the one with the greatest Z-coordinate. By examining the tree, we can see that the environment has activated three other processes, one of which has activated a fourth process.

The visualization rules presented in this paper have been augmented with additional rules to produce the graph above the tree, which captures the state of the processes and communication channels. Active processes are represented as green circles, while inactive processes are represented as red. The addition of a message or signal to a communication channel is represented by an animation in which a small colored segment traverses the line representing the channel from the sender to the receiver; this cannot be properly illustrated in the print medium.

The state graph is constructed at a uniform Z-coordinate somewhat greater than that of the environment; the X and Y coordinates of each of the circles in the graph are the same as the coordinates of the corresponding elements of the tree. The two portions of the representation are thus visually connected, and this connection can be perceived in this image by comparing the green (active) circles of the state graph with the spheres in the tree. The viewer of the visualization can change the viewpoint (for example, by rotating the entire image about the Z-axis) to better emphasize this relationship.

A 7-minute video tape entitled *Diffusing Computations* contains the complete animation and is available on request.

frame 0:	line with endpoints [0, 0, 0] and [2, 2, 2]
frame 1:	line with endpoints [0, 0, 0] and [2.5, 3, 1]
frame 2:	line with endpoints [0, 0, 0] and [3, 4, 0]
frame 3:	line with endpoints [0, 0, 0] and [3.5, 5, –1]
frame 4:	line with endpoints [0, 0, 0] and [4, 6, –2]

Once the entire collection of tuples has been evaluated, the image corresponding to each frame is generated. This is simply a matter of setting up (under user control) the viewpoint and lighting and then, for each of the primitive graphical objects, performing whatever function calls are necessary to produce the object.

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 visually explore 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. Figure 2, for instance, captures one frame from a video animation in which the operational details of the network computation and the abstract representation of the termination detection protocol are captured together. The video was generated by adding new rules needed to capture the state of processes and channels and by modifying the animation rules above to improve their visual impact. (Spheres drop from the network representation while growing to the desired size and attach themselves to the parent sphere by sprouting the connecting line.) 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 flow of messages.

4. Visualization Methodology

In the previous section we showed how one can declaratively specify complex visualizations. We turn our attention to a closer examination of the process by which we decide to select one particular visual representation over others and by which we identify minimal animation requirements, e.g., movement, highlighting, etc. The basis for our approach is the hypothesis that *program properties which are significant in correctness proofs must be captured by any meaningful visual representation of the program.* The strategy involves an analysis of the program proof (or proof outline) and the application of a number of heuristics that foster the transfiguration of (program-wide) computational properties into appropriate visual counterparts. In this section we apply this approach to a second program; we do not list the explicit rules involved in producing the final image.

The computation which we wish to visualize is the *Bagger* problem [12]. This problem is a variant of bin-packing in which a collection of items having various positive integer weights are placed into one or more bags; the items in each bag are ordered, and items within a bag must be placed from largest to smallest by weight. The bags all have a capacity indicating how much weight the bag can hold; this capacity is the same for all bags and is identified by the symbol H. A number of solutions to this problem have been developed; we use the Swarm solution and proof presented in [10]. In this solution, the only tuple type is the *item* type; a tuple *item(id, weight, bag, position)* indicates that the item with unique identifier *id* has the given *weight* and is located in the *bag* at the indicated *position.* Item and bag identifiers are positive integers, with item identifiers assuming all values between 1 and N, the number of items. By convention, the bag identifier 0 is used for items that have not yet been placed in a bag. Note that there is no explicit representation of a bag; rather, a bag is defined by the items that are placed in it.

The Swarm solution to Bagger (Figure 3) has three transaction types. The transactions *Track_Weight* and *Make_Bag* work together to generate new bags as needed. Specifically, *Track_Weight* detects the condition where the heaviest unbagged item does not fit in any of the existing bags and creates a *Make_Bag* transaction. In turn, *Make_Bag* places the first item into the new bag and creates a *Track_Weight* and a *Bag_In* transaction. At all times there is either a *Track_Weight* or a *Make_Bag* transaction (but not both). The *Bag_In* transaction adds items to a particular bag until the bag is full (i.e., none of the remaining unbagged items fit in the bag). There may be several *Bag_In* transactions at any time, one for each bag that is not yet full.

Both *Track_Weight* and *Bag_In* exhibit tracking behavior. One of the parameters of the transaction type is an item weight, and the transaction queries the dataspace for an item of that weight. If such an item is found, the transaction takes one action (creating a *Make_Bag* tuple in the case of *Track_Weight* and adding the item to the bag in *Bag_In*); if the item is not found, the transaction reduces the weight by one and searches for the new weight. This comparison between item weights and bag capacities is key to the program—a number of proof properties refer to it—and, therefore, plays a major role in constructing the visualization.

We are now ready to begin development of the visualization. The final visualization will be composed of a number of stages (the mappings discussed in the previous section); however, we start by creating a specification of the overall mapping and only later develop the rules needed to implement the specification. The first step is to determine what aspects of the computation need to be represented and to select appropriate graphical representations of these aspects.

Using our methodological guidelines, we examine the correctness properties of the proof; in particular, we focus on *safety* properties (**unless**, **invariant**, **stable**, **constant**—see [2]) which indicate what the program is allowed and is not allowed to do. It is among these properties that we find clues to what aspects of the program state must be represented by geometric objects. In the case at hand, we find a number of invariants related to items and bags. For example, we find

- There is exactly one *item* tuple for each of the item identifiers.
- The weight associated with a particular item identifier is constant.
- Once an item is placed into a bag, neither the item's bag nor its position in the bag change.
- No two items can be placed in the same position in a single bag.
- All bags have a constant capacity (which is the same for all bags).
- The sum of the weights of all items in a bag does not exceed the bag's capacity.

These properties indicate that some representation of the items and the bags is essential to the visualization. In itself, this is hardly interesting. However, we can now use the properties to select appropriate graphical representations.

Since there is exactly one item for each identifier at all times, and each item has a specific constant weight, we should select as our representation of the items some graphical object with an attribute to which the weight may be mapped. Almost any scalar attribute could be used; we could, for example, use a color spectrum, with blue hues representing the smaller items and red the larger. However, the last property listed above indicates that the sum of weights is significant and we should therefore choose a property amenable to summation—color would not be good, because the concept that "blue + blue = yellow" is hardly natural. A more appropriate measurement would be a dimension, which is amenable to summation; indeed, dimensions can be readily summed by positioning the objects such that the dimensions in question form segments of a line. Accordingly, we might select rectangles to represent the items and make one dimension of the rectangle proportionate to the item's weight.

A similar representation is suggested for bags, with the corresponding measurement being the bag's capacity. However, the bag's situation is somewhat more complex. In the Swarm formulation of the Bagger program, bags are *abstract* entities. There is no explicit representation of bags in the Swarm dataspace; instead, bags exist if (and only if) items have been placed within them. Thus, whereas the *item* tuples are in one-to-one correspondence with the rectangles representing the items, the rectangles representing bags must be generated with somewhat more care. In addition, we probably want to emphasize the aggregate nature of bags; since a bag is represented by the items contained within it, we may want to position the representations of the items and the representations of the bags in such a way as to emphasize this relationship. The ability to capture and represent such abstract entities and their properties is one of the major strengths of our declarative approach to visualization.

```
Program Bagger (H, N, weight : natural(H), natural(N), weight[1..N] of natural)

tuple types
    [I,w,B,n : natural(I), natural(w), 0 < w ≤ H, natural(B), natural(n) : item(I,w,B,n)]

transaction types
    [B,c,n,w : natural(B), natural(c), 0 ≤ c ≤ H, natural(n), natural(w), 0 < w ≤ H ::
        Track_Weight(B,w) ≡
                I,B',c',n',w' : item(I,w,0,0), Bag_In(B',c',n',w'), (w ≤ c') →
                                        Track_Weight(B,w)
            ||  : [∀ I : ¬item(I,w,0,0)], (1 < w) → Track_Weight(B,w-1)
            ||  : NOR,  (0 < w) → Make_Bag(B,w);
        Make_Bag(B,w) ≡
                I :  item(I,w,0,0)† →
                     item(I,w,B,1), Bag_In(B,H-w,2,min(H-w,w)), Track_Weight(B+1,w);
        Bag_In(B,c,n,w) ≡
                I :  item(I,w,0,0)† →
                     item(I,w,B,n), Bag_In(B,c-w,n+1,min(c-w,w))
            ||   : NOR, (0 < c), (1 < w) → Bag_In(B,c,n,w-1)
    ]

initialization
    [I,w : 0 < I £ N :: item(I, weight(I),0,0) ], Track_Weight(1,H)
```

Figure 3: Swarm solution to the *Bagger* program.

All data is encapsulated in the *item* tuple type. The four components of this tuple give the unique item identifier, the item's weight, the bag containing the item, and the position of the item in the bag. The special bag identifier 0 is used to indicate the item is unbagged; this is reflected in the **initialization** section, where each item is created with the third and fourth components set to 0.

Three transaction types are used in the solution. The *Track_Weight* transaction type has two parameters which are the number of the next new bag and the weight which is being tracked. The *Make_Bag* transactions two parameters are the number of the bag being created and the weight of the first item to be placed in the bag. *Bag_In* has four components: the bag number, the remaining capacity of the bag, the position in the bag of the next item to be packed, and the item weight that the bag is tracking.

Track_Weight determines when new bags must be created. The first subtransaction of *Track_Weight* succeeds if an unbagged item and bag can be found such that the item is of the weight being tracked and can fit in the bag; the effect of this subtransaction is simply to re-create the *Track_Weight* transaction. The second subtransaction succeeds if there are no items of the tracked weight, and creates a *Track_Weight* which searches for an item with weight one less than the current search weight. The third transaction succeeds if neither of the previous two do, i.e., if there is some unbagged item that will not fit into any of the bags; this transaction creates a *Make_Bag* transaction.

Make_Bag locates an unbagged item of the indicated weight, puts it into the selected bag, creates a *Bag_In* transaction to work on the new bag, and creates a *Track_Weight* transaction. The *min* expression computes the search weight; it will be either the original search weight or the remaining capacity of the bag, whichever is smaller.

Bag_In uses two subtransactions to add items to a bag. The first subtransaction attempts to find an unbagged item of the search weight; if one is found, it is put in the bag. The *min* expression is again used to compute the next search weight. The second subtransaction creates a *Bag_In* with a search weight one less, provided the bag still has some capacity and the new search weight will be at least 1.

Summarizing our representation decisions, both *bags* and *items* will be represented in the final image by rectangles. Bag rectangles have some (arbitrary) width, and a length proportionate to the bag capacity. Item rectangles fall into two groups, those representing unbagged items and those representing bagged items; this distinction will probably be made in the proof space and maintained throughout the visualization, since the two types require different treatment. All item rectangles will have the same width; the length of a rectangle will be proportionate to the weight of the item represented, using the *same* proportionality constant as is used for the bag lengths. The position of the item rectangles depends on the item type. Unbagged items can be placed more-or-less arbitrarily (this decision will be modified soon!). Bagged items are aligned (visually "summed") within the representation of the bag; the position of a particular item depends both on the position of the bag containing that item and on the weights of the items positioned "before" that item in the bag.

One possible arrangement is illustrated in Figure 4. We give each item rectangle a distinct, arbitrary color (represented by gray tones in the figure). This is not prompted by any of the correctness properties, but is imposed by the requirement that the rectangles be easily distinguished. This representation simultaneously captures three of the correctness properties listed above: once placed, items remain in the same position in the bag; no two items are at the same position in the same bag; and the sum of the item weights does not exceed the bag capacity.

We will eventually define functions that compute the positions of the rectangles representing bags and items. The discussion in the previous paragraph, coupled with the invariants relating to bagged item positions and weights and with the desire to visually link the representation of a bag with the representations of the items contained within the bag, lead us to develop certain constraints that apply to these functions. Predicates over the program state are thus reshaped as geometric constraints. Below is an example of how this happen.

Let us assume that rectangles are represented by the coordinate of one corner and the X and Y dimensions. Furthermore, let $bag_{XP}(B)$ represent the function that produces the X-coordinate of a bag B, $bag_{YP}(B)$ be the function that produces the Y-coordinate of a bag B, and $bagged_{XP}(I)$ and $bagged_{YP}(I)$ be the corresponding functions for the positions of bagged items. Assume also that the length of a rectangle is determined by multiplying the item weight or bag capacity by the constant K. Then the following constraints on these functions reflect the invariants we considered and our choice of layout in the following manner:

$$I \text{ is in } B \Rightarrow bagged_{XP}(I) = bag_{XP}(B)$$
$$I \text{ is in } B \text{ at position } P \Rightarrow bagged_{YP}(I) = bag_{YP}(B) +$$
$$K \times (\Sigma\, i : i \text{ is in } B \text{ at position } p, p < P, \text{ and } i \text{ has weight } w : w)$$

In other words, the rectangle for a bagged item has the same X-coordinate as its bag, while its Y-coordinate is the Y-coordinate of the bag plus K times the sum of all weights of items which appear "before" the item in the bag. The latter positioning constraint is prompted by the last invariant listed earlier and by the decision to represent this invariant by visually summing lengths.

The safety properties listed above suggest one more characteristic of the final visualization. The first two properties indicate that the number of items and the weight of each individual item are *constant*. Imagine for a moment the final visualization; one important change in the image takes place when an unbagged item becomes bagged. We could cause this to occur instantaneously; the rectangle representing the item would disappear and reappear within one of the bags. However, the two properties indicate that in this case such "teleportation" would be undesirable. These properties require that the items and their weights are constant; we should therefore exploit *visual continuity* to convey the sense of constancy. Rather than teleportation, a steady movement from the old position to the new is indicated; this allows the viewer to instantly perceive that the number of rectangles and the length of each rectangle are constant.

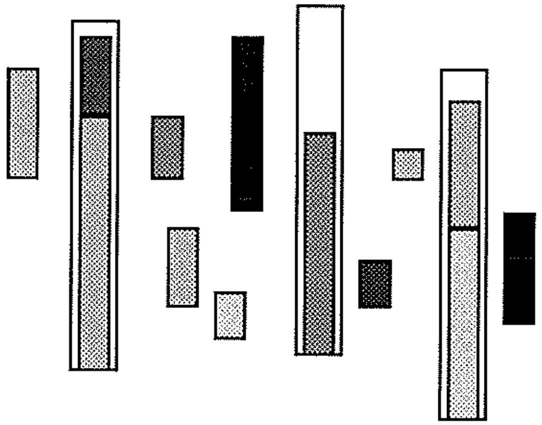


Figure 4: An initial layout of the Bagger visualization.

The representations of items and bags (filled and unfilled rectangles, respectively) have no particular arrangement. The rectangles representing bagged items are stacked within the bags, thus visually illustrating both the ordering of the items and the summing of their weights.

We can now consider several other properties of the computation and see what they indicate about the representation:

- The number of unbagged items is a non-increasing function of time which eventually reaches 0.
- The weight which the *Track_Weight* transaction is tracking is always greater than or equal to the largest weight of all unbagged items.
- A new bag is created if and only if *Track_Weight* detects an unbagged item whose weight is greater than the remaining capacities of all bags for which a *Bag_In* transaction exists.
- The weight which each *Bag_In* transaction is tracking is always less than or equal to the remaining capacity of the transaction's bag.
- No item has a weight which exceeds the weight which a *Bag_In* transaction is tracking and is also less than the remaining capacity of that transaction's bag.

The first of these is a progress property; the others are safety properties which constrain the "tracking" behavior discussed above. The progress property indicates that the decrease in the number of unbagged items is significant. This indicates a flaw in the layout of Figure 4, where bags and unbagged items are indiscriminately positioned; we need to separate the unbagged items from the bags so that a decrease in the number of unbagged items is easily observed. One simple way of doing this is to make the X-coordinate of the objects a function of the identifier, with unbagged items mapping to negative X-coordinates and bags (and bagged items) mapping to positive coordinates.

The tracking properties indicate that *comparison* between the weights of the unbagged items and the remaining capacities of the active bags is of primary importance to the algorithm. Both of these concepts are visually represented by lengths; the properties indicate that the lengths should be positioned in the image in such a way that the relationship can be instantly perceived. One possible arrangement is indicated in Figure 5, where the rectangles representing the unbagged items are aligned with the tops of the rectangles representing the bags.

The program properties we are currently considering also suggest that we need to represent additional aspects of the computation; specifically, we need to know what weight the *Track_Weight* transaction is tracking, and for each active *Bag_In* we need the same information. This can be represented simply enough by adding marks (lines or arrows) as illustrated in Figure 5. The operation of the *Track_Weight* transaction is represented by the segment extending the length of the unbagged items, while the *Bag_In* activity is represented by the arrows next to each bag. These marks function as gauges, moving upward to indicate the measure that is being tested.

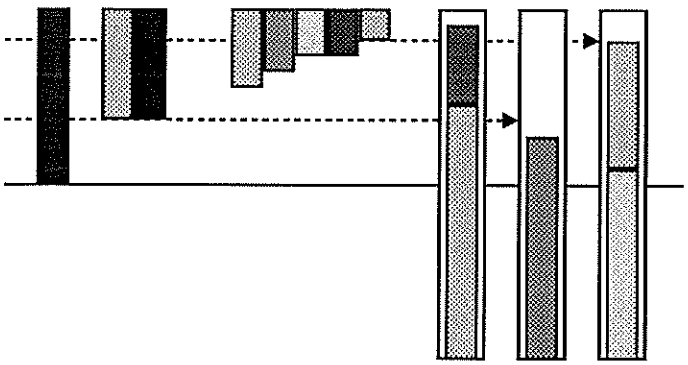


Figure 5: Modified Bagger visualization.

The weights of unbagged items and the capacities of bags can be visually compared, paralleling the comparison performed in the algorithm. The solid horizontal line shows the weight which *Track_Weight* is examining, while the dotted arrows show the weights which each of the active *Bag_In* transactions is examining.

The *Bag_In* marks also capture some progress properties: the movement of the marks indicates the search behavior of each transaction, while the marks disappear as the *Bag_In* transactions complete their operation. Progress is here captured by a steady movement toward a boundary (the end of the bag); we have found such steady motion to be a good representation of progress conditions.

The choice of suitable boundaries for these marks is somewhat difficult. The second and third invariants above indicate that *Track_Weight* is simultaneously comparing *all* the weights of unbagged items and *all* the remaining bag capacities, so appropriate bounds for this line would extend the entire length of the visualization (i.e, spanning the representation of all the bags and all the unbagged items). Similarly, the fourth and fifth invariants indicate that the mark for each active *Bag_In* transaction should extend from the transaction's bag across all the unbagged items. Including all these marks results in a somewhat confusing image. We can mitigate this somewhat by making the *Bag_In* arrows "weaker", for example by using a faint color for the arrows (indicated by dotted lines in the figure). This permits visual comparison of the values without cluttering the image.

Addition of these marks brings out a minor problem with this representation. The layout of Figure 5 has the property that when we reduce the capacity of a bag, the reduction is visually represented by the removal of space from the "wrong" end of the remaining space, as represented diagrammatically in Figure 6. Ideally, we want a layout that permits easy comparison of all four elements shown there; that is, the representations of the item weights, of the bag capacities, of the *Bag_In* and *Track_Weight* tracking, and of the space removed by adding an item to the bag should all be visually aligned.

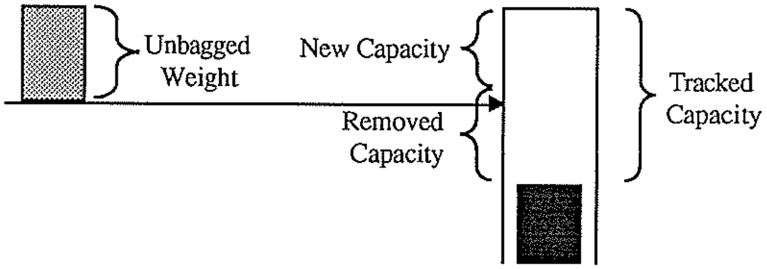


Figure 6: Elements which should be aligned to permit easy visual comparison.

When the unbagged item show at left is added to the bag, the item is placed at the lower end of the remaining capacity being tracked by the *Bag_In* transaction. The tracking mark then moves upward to the level indicated by *New Capacity*.

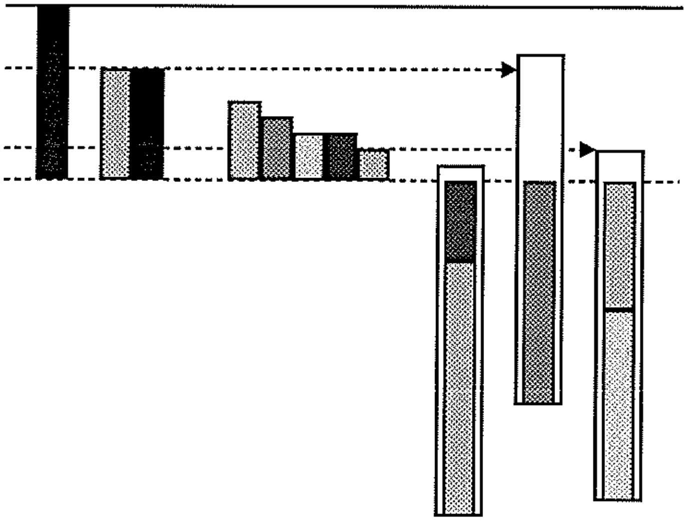


Figure 7: Final layout of the Bagger visualization.

As unbagged items are placed into the bags, the representation of the bag, its contents, and the tracking arrow all sink downward. All four lengths from Figure 6 are aligned in this layout.

Such a layout is obtained in Figure 7. In this layout of the elements, the lengths to be visually compared are all arranged so as to extend upward from some arbitrary horizontal line; this line is included in the layout as a dashed line (color would serve to distinguish the line in the final visualization). Note that the visual alignment of the lengths does come at some cost: We must arrange the visualization so that the representations of the bags as well as all the bagged items "sink" downward when new items are added to the bag. This involves a certain extra complexity in the rules, but more significantly the visual confirmation that all bags have the same total capacity is not as clear as in Figure 5. Here again we rely on visual continuity to reassure the viewer that the total capacity does not change; smooth movement of items and bags allows the viewer to perceive readily that they do not change size.

The development of the visualization is now essentially complete; all that remains is to write the rules that will produce the desired effects. As mentioned in the beginning of this section, we omit the presentation of the rules; however, we outline below the information produced by each of the mappings.

The proof mapping will produce five basic types of information, which will be captured by five distinct tuple types. For unbagged items, we require the item identifier and weight. The information about bagged items includes the item identifier, weight, bag, and the sum of the weights of all items located before the item in the bag (the last is required for positioning the item). Bag information includes the bag identifier and the sum of the weights of all items in the bag; note that this information is of use in positioning the bag, but can also be used when computing the position of the bagged items. Finally, we have two types of information involving the marks: one for the *Track_Weight* and one for each active *Bag_In*. In both cases, we need to know the value being tracked; in addition, for the *Bag_In* we need to know the identifier of the bag.

The object mapping will generate abstract geometric objects for the various elements. We will retain five basic types of information, corresponding to the five types contained in the proof space. However, the information for each type may be gathered from several tuples in the proof space; for example, the positioning of a bagged item depends both on the individual attributes of the item and on the attributes of its bag, since in our layout the position of the bags changes. Suitable functions will be defined to compute the coordinates of the layouts; these functions must obey certain restrictions as discussed above.

The animation mapping detects changes in the object space and generates the final animation space. Both for reasons of visual continuity and to improve the appearance of the visualization, we wish to capture several different visual events. Among these are the movement of unbagged rectangles to bagged rectangles, the "sinking" of bags (and bagged rectangles) when new items are added, and the movement of the tracking marks. In order to detect these types of transitions, we need a way to uniquely identify particular abstract objects. In our case, we will use the item and bag identifiers from the original program.

The final visualization captures most of the correctness properties of the computation and also shows the algorithm's behavior (including the operations of transactions, which are captured by the marks used to show the tracking invariants). For this reason, the final visualization is useful both for pedagogical (explaining the algorithm's operation), and for monitoring (examining and verifying the operation) purposes.

5. The Pavane Environment

The Pavane system consists of three primary components. *SwarmParse* is the Pavane compiler; its function is to translate programs and visualizations written in Swarm into executable form. *SwarmExec* is a run-time package which executes compiled Swarm programs and visualization rules and produces animation traces (representations of instances of the animation space). *SwarmView* reads animation traces (from SwarmExec, or from any other source which produces the traces) and renders the images described by the traces. In our current implementation of Pavane, SwarmParse and SwarmView are written in C and SwarmExec is written in Prolog. SwarmParse and SwarmExec both run on a Macintosh IIfx; SwarmView runs on a Silicon Graphics Personal Iris. SwarmExec and SwarmView execute concurrently, communicating via an Ethernet connection between the machines.

Both SwarmView and SwarmExec have been designed to be capable of interfacing with other programs. SwarmView can accept and display animation traces from any source which observes the correct protocols. In the SwarmExec package, the section which performs the visualization rules and transmits the results is self-contained and can perform these functions for any underlying computation which produces a Swarm dataspace. For example, we could read information over the Ethernet from another machine, assert it into the Prolog database, apply the visualization rules, and transmit the results to SwarmView. This ability to "patch in" at any point in the visualization pipeline promises to be quite useful; we have already taken advantage of this by writing a replay facility which simply connects to SwarmView and sends the contents of a previously-saved file of animation traces.

5.1 SwarmParse

SwarmParse, the Pavane compiler, uses the LL(1) parsing method to translate programs written in Swarm (extended by function definitions and visualization rules) into Prolog files suitable for use by SwarmExec. We refer to the output language as SwarmProlog, since (although it is simply Prolog) the SwarmProlog elements require the framework provided by SwarmExec to be executed. Three main components of the extended Swarm language are represented in SwarmProlog: the Swarm dataspace and visualization spaces, the Swarm transaction definitions, and the visualization mappings. The next subsections describe how SwarmParse represents each of these components.

Dataspaces. Swarm, as extended with visualization capabilities, uses several distinct dataspaces (collections of tuples) representing the computation state. These spaces include the *tuple* space, the *transaction* space and the *synchrony relation* of Swarm, and the *proof*, *previous proof*, *object*, *previous object*, and *animation* spaces of the visualization extension.

Each of these spaces is represented by a collection of facts in the Prolog database. Each fact has the form

```
space_name( tuple_type( tuple_parameters ) ).
```

where `space_name` is one of `tuple`, `proof`, etc., `tuple_type` is the Swarm tuple type, and `tuple_parameters` is the parameter list (if any) of the particular tuple. SwarmParse makes appropriate syntactic adjustments to accommodate Prolog syntax; for example, the `tuple_type` and all other atoms must begin with small letters, since capital letters are variables in Prolog.

To create the initial contents of the dataspace, a Prolog goal `initialize_dataspace` is provided as part of the completed SwarmProlog program. The function of this goal is to generate the initial contents of the dataspaces, using the Prolog `assert` goal. SwarmParse translates the initialization section of the Swarm code directly into the `initialize_dataspace` goal.

Transaction definitions. A Swarm transaction consists of a collection of parallel subtransactions, each consisting of a list of variables, a query part, and an action part:

$$\begin{array}{ll} \multicolumn{2}{l}{\text{transaction(parameters)} \equiv} \\ & \text{variables}_1 : \text{query}_1 \rightarrow \text{action}_1 \\ & \quad \ldots \\ \| & \text{variables}_n : \text{query}_n \rightarrow \text{action}_n \end{array}$$

SwarmParse translates such a transaction into a Prolog goal which is executed by the SwarmExec run-time package. The goal consists of a sequence of sub-goals, each the translation of one of the subtransactions. The variables are represented by Prolog variables. SwarmParse automatically generates a Prolog variable for each Swarm variable. Some care must be taken with this process, since the scope rules of Swarm and Prolog differ—in Swarm the variable scope is confined to its subtransaction, while in Prolog, the variable's scope is the entire goal.

The translation of the query and action is quite direct; for example, to determine if the tuple *t(1)* is in the dataspace, the Prolog goal `tuple(t(1))` is used, while to insert such a tuple the goal `insert_tuple(tuple(t(1)))`. Expressions occurring within a tuple must be pulled out and separately translated so that the result of evaluating the expression can be determined (in Prolog, 1 + 1 is not 2, but a structured object having the functor +). SwarmExec also provides goals to perform the more complex types of queries and actions, including goals to perform quantifiers, summations, generators, and so forth.

Visualization mappings. A visualization consists of three distinct mappings (collections of visualization rules), the *proof* mapping, the *object* mapping, and the *animation* mapping. Each mapping transforms one or more input spaces to an output space; for example, the proof mapping transforms the state space (the Swarm dataspace) into the object space. A visualization rule consists of a list of variables, a query over one or more spaces, and a list of tuples to be produced:

$$\text{variables} : \text{query} \Rightarrow \text{production}$$

The interpretation of such a rule is as follows: For all instantiations of the variables such that the query is true, include in the output space the list of tuples appearing in the production. This is exactly the same definition as a Swarm generator, except that queries over the dataspace are not permitted in a Swarm generator; however, the same translation mechanism suffices.

In addition to the rule translations, the collection of rules which make up each mapping must be represented. This is achieved by providing lists of all the rule names generated by SwarmParse.

5.2 SwarmExec

SwarmExec, the Pavane run-time package for the execution of SwarmProlog programs, is written in AAIS Prolog. To use SwarmExec, the user enters the Prolog environment, consults the file containing the SwarmExec code, and then consults the file(s) containing the SwarmProlog produced by SwarmParse. The user then executes the goal `swarm_run`, which begins the interpretation cycle. (A more convenient interface for starting SwarmExec and consulting user programs is contemplated.)

SwarmExec simulates Swarm execution in a serial fashion; the implicit parallelism of Swarm, in which two transactions can be executed simultaneously provided they do not interfere, is not accommodated. However, SwarmExec does provide for explicit parallelism through the use of the synchronic group; all transactions in a synchronic group will be executed in effective parallelism.

Primary control. The `swarm_run` goal is executed following loading of the files containing the SwarmProlog goals. All remaining execution is controlled by `swarm_run`. The `swarm_run` goal first initiates communication with the Silicon Graphics Personal Iris (used to render the visualizations) over the Ethernet. As a result of initiating this connection, SwarmView begins running on the Personal Iris. The "control panel" of the combined SwarmExec/SwarmView system appears on the Personal Iris screen; the user controls the execution of the simulation from the Iris while examining the results of the visualization.

`swarm_run` then creates the initial dataspace, using the `initialize_dataspace` goal generated by SwarmParse. After this initialization is completed, the goal `swarm_step` is repeatedly called. `swarm_step` succeeds only if the execution of SwarmExec finishes (because the user stops execution) and fails otherwise. When `swarm_step` succeeds, execution terminates and the connection with the Personal Iris is closed.

The goal `swarm_step` interacts with the Personal Iris to receive a command from the user. The commands include performing a single atomic action of the Swarm program, restarting the Swarm computation from the beginning, and ending the computation (the last is the only case in which the `swarm_step` goal succeeds).

Execution of an atomic action. When an atomic action is performed, a transaction is first selected in a fair manner. This is currently achieved by selecting the first transaction found in the Prolog database, and adding new transaction instances to the end of the database; we are contemplating several other techniques which will provide fairness and a greater degree of nondeterminism. If no transaction can be found, the atomic action is completed; note that this does not terminate execution of SwarmExec and SwarmView, but the fact that the computation has completed is signaled to SwarmView and displayed for the user.

If a transaction is found, the synchronic group to which the transaction belongs is determined. This is done by performing a search from the initial transaction through the closure of the synchrony relation. The synchronic group is represented as a Prolog list of transactions with parameters. The group may contain transactions which are not actually present in the dataspace; by the Swarm definition, only those transactions which are actually present will be executed.

The next step is to prepare for execution of the group. The `sw_readyupdate` goal performs this task. The most important function of this goal is to initialize a collection of counters which are used in the processing of the special Swarm queries such as **NOR**, **NAND**, and so forth. These counters store the total number of local subtransactions (those not involving special queries) which are executed and the number of these transactions which succeeded and failed.

Execution of the group is performed by traversing the list of transactions making up the group. If a transaction in the group is also present in the dataspace, it is executed by the `sw_exectrans` goal. This goal first asserts into the Prolog database a fact of the form

```
sw_deletion(true,trans(Trans)).
```

The semantics of Swarm requires that all queries involved in a single atomic operation (execution of a synchronic group) must be performed before the dataspace is altered. We therefore record all dataspace modifications, such as the deletion of a transaction instance when the transaction is executed, in the Prolog database using the `sw_deletion` and `sw_insertion` tuples. Only after execution of the group is completed are these tuples processed to update the dataspace. The first component of a `sw_deletion` or `sw_insertion` tuple is a tag which gives the type of subtransaction which resulted in the modification (i.e., `local`, `true`, `nor`, etc.); the second component is the element to be removed or added.

After the fact that the transaction should be removed from the dataspace is recorded, the transaction is executed using the `call` goal. The Swarm transaction consists of a sequence of subtransactions, each represented in the form

```
setquerytype( type ),
    (  translation of query,  !,  translation of action,  itsucceeded ; itfailed )
```

The `setquerytype` goal makes a temporary record of the type of the subtransaction; this is then referenced when generating `sw_deletion` or `sw_insertion` tuples. The query translation is performed; if it succeeds, the action is performed (possibly generating `sw_deletion` and `sw_insertion` tuples) and the goal `itsucceeded` is executed. If the query fails, the second part of the alternative construct, i.e. the goal `itfailed`, is executed.The `itsucceeded` and `itfailed` goals modify as appropriate the counters of the total number of local subtransactions and the number which succeeded and failed.

After all transactions in the synchronic group are executed, the dataspace is updated by the `sw_doupdate` goal. The first action of this goal is to determine what *types* of special transactions are successful; for example, the **OR** subtransactions succeed if any of the local subtransactions succeeded, i.e. if the count of successful local subtransactions is greater than zero. Local subtransactions are always successful. The `sw_deletion` and `sw_insertion` tuples are then processed, with the deletions performed before the insertions (again, this is Swarm semantics). Only tuples which have been tagged with a successful type are processed; the others are discarded.

After a synchronic group has been processed (whether or not a group was actually found and performed), the visualization rules are applied to the state space and the resulting animation space is transmitted to the Personal Iris. This process begins by copying the current proof space and object space to the previous proof and previous object spaces respectively, then clearing the proof, object, and animation spaces. Each of the three mappings—proof, object, and animation—is then applied in turn to generate the spaces. Applying the rules is simply a matter of finding the list of rule names, then calling each of the rules.

Interpretation of generators. The Swarm generator construct has the form [variables : predicate : list], which indicates the operation "for each instantiation of the variables such that the predicate is true, create all the objects in the list". The generator is particularly useful in the initialization part of Swarm programs. As mentioned previously, the behavior of the visualization rules is quite similar. Two additional Swarm constructs, the subtransaction generator (which is used in transactions to specify a collection of subtransactions) and the universal quantifier (∀ v : p : q) are also similar in behavior. In all four cases, we want to determine all instantiations of a collection of variables such that a particular predicate (or query) is true, then perform some action for all such instantiations.

Two Prolog goals, `forall` and `forall_worker`, each with two clauses, suffice to provide all four of these constructs. `forall` uses the built-in Prolog goal `bagof` to find all instantiations of a variable list such that a goal (corresponding the the predicate part of the generator) can be satisfied. The list of all such instantiations is provided to `forall_worker`, which recursively processes the list and performs the action for each variable instantiation in the list.

As a general rule, each invocation of `forall` will be accompanied by definitions of two additional goals used for the predicate and action goals. Because of the behavior of `bagof`, some extra syntactic elements must be included in invocations of `forall`. These elements serve to existentially quantify the variables of the variable list for the `bagof` goal, ensuring all possible instantiations will be found.

A similar `thereexists` goal is also provided, with the same syntactic form (which is not actually required in this case; however, keeping the forms similar simplified some aspects of SwarmParse). `thereexists` succeeds if it can find any instantiations of the variables for which both predicates succeed. Finally, several goals are provided to produce sums, products, and so forth. These are all similar in form and closely related to the generator, except that for each instantiation of the variables an expression is evaluated and some operation is applied to the list of values so obtained.

5.3 SwarmView

The SwarmView system actually consists of two parts, which together act as the server in a client-server communications model using the Ethernet. The first part is a server daemon which runs continuously. The daemon responds to requests for service by initiating the SwarmView program proper. SwarmView then completes the connection and begins to interact with the client.

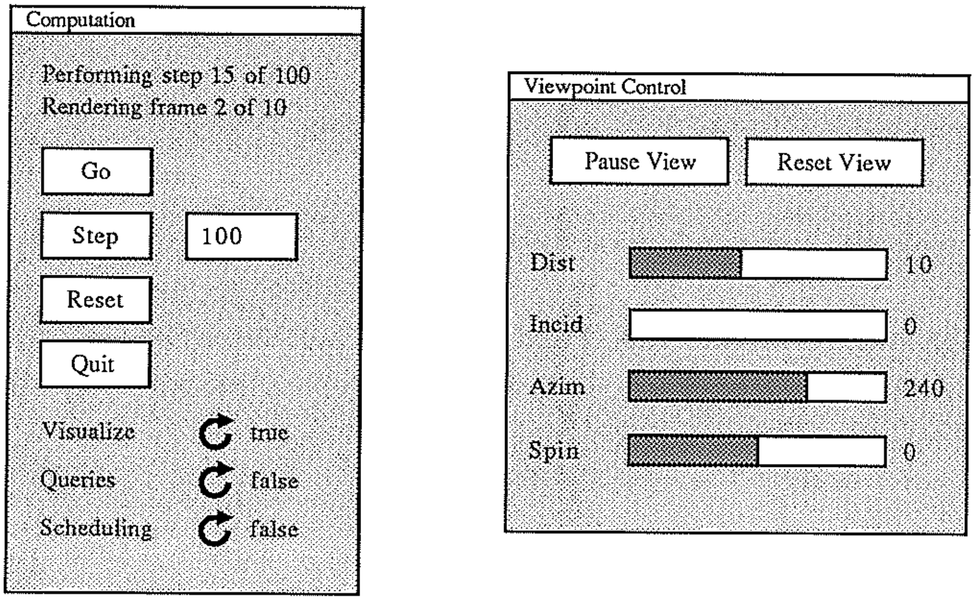
In our current configuration, the client is the Macintosh running SwarmExec; however, the system can service requests from any source. Both the daemon and SwarmView are implemented in C running under Silicon Graphics' IRIX operating system. In addition, SwarmView uses the Silicon Graphics graphical library for the rapid definition and display of graphics. The daemon is a standard example of its type and will not be discussed further.

SwarmView has four major functions. The most important of these is the interaction with the user; this interface provides both control of the computation (through the interface with SwarmExec) and control of the user viewpoint. In addition, SwarmView reads and stores collections of animation tuples from the client process. SwarmView also interprets the stored animation tuples to produce sequences of frames consisting of collections of graphical objects in three-dimensional space. Finally, SwarmView renders these graphical objects in a window on the Silicon Graphics screen.

User interactions. SwarmView interacts with the user in two ways: computation control and viewpoint control. The "control panels" which provide the viewer interface to these facilities are illustrated in Figure 8.

Control of the computation is provided by four buttons and three rotary switches; the viewer uses the Silicon Graphics mouse to manipulate these controls. Each of the buttons sends a command to SwarmExec. After sending the command, SwarmView waits for a response from the client; depending on which button was selected, the response might be a simple acknowledgment or might be a new instance of the animation space.

The *Quit* button sends a command to terminate the computation; when the response is received, SwarmView closes the connection to SwarmExec and exits. The *Reset* button sends a command to restart the computation from the initial dataspace; the response is the initial animation space. *Step* indicates one or more atomic actions of the computation should be performed; the response is the new animation space. The number of steps which are to be performed can be entered by the user and appears in the box to the right of the *Step* button. *Go* performs an unbounded series of steps. When either *Go* or *Step* is performed, the *Go* button's label changes to *Pause*; this button halts the sending of step commands.



Performing step 15 of 100
Rendering frame 2 of 10

Step 100

Visualize true
Queries false
Scheduling false

Dist 10
Incid 0
Azim 240
Spin 0

Figure 8. The SwarmView control panels.

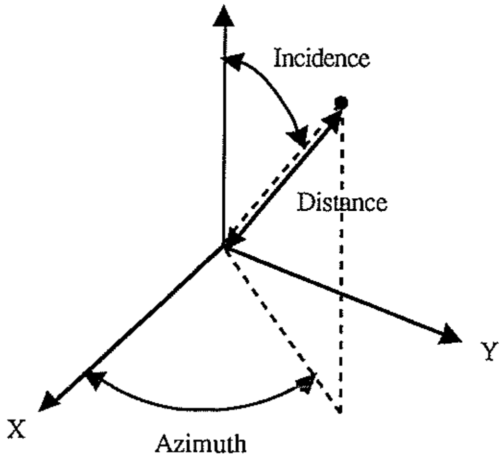


Figure 9. The polar viewing model used by SwarmView.

The viewer's eye is located by a distance from the origin and two angles. By changing these parameters and the location of the coordinate origin, the viewer can observe any part of the visualization.

The three rotary switches below the buttons are used to determine some parameters of the computation's behavior. *Visualize* determines whether visualization rules will be applied and the animation space transmitted; it is normally true. *Queries* and *Scheduling* are included for future expansions. If *Queries* is true, SwarmExec will perform certain additional queries and display the results (for example, listing all *item* tuples in the dataspace). If *Scheduling* is true, SwarmExec will select a synchronic group based on user-provided scheduling rules.

Viewpoint control is provided by two buttons and four sliders. Once SwarmView has read and stored an instance of the animation space, it begins generating and rendering the frames described by that animation space. The *Pause View* button stops the animation clock, allowing the viewer to examine a single frame in greater detail. While the view is paused, the remaining viewpoint controls can still be used.

The four sliders control the actual viewpoint, i.e., the location of the viewer's eye in three-dimensional space. SwarmView uses a polar viewing system as illustrated in figure 9; the user's eye is located by a distance from the origin, an incidence angle measured from the Z-axis, and an azimuthal angle measured in the X-Y plane. The *Dist*, *Incid*, and *Azim* sliders are used to set the values of these three parameters. The *Spin* slider can be used to cause the azimuthal angle to steadily change; this causes the image to rotate as the user's eye circles the Z-axis.

The location of the coordinate origin is controlled directly through the image. By "clicking and dragging" in the image, the user shifts the center of the coordinate system in a corresponding manner. The *Reset View* button restores the origin and viewpoint to their original values.

Animation space input and storage. The animation space tuples are transmitted from the client as strings in response to a step or reset command. These strings are passed through a lexical analyzer written in LEX and a parser written in YACC. YACC produces an LALR(1) parser with associated actions; in this case, the actions are chosen such that each tuple parsed results in the creation of a dynamic data structure of type `SV_Object` and storage of this object in a list which represents the collection of tuples.

An `SV_Object` is actually a pointer to a C structure which includes a number of components. One component is a pointer to the master description of the object's type, which contains information about the object parameters, the function used to render the object, and so on. Another component is an array which stores the expressions assigned to each of the object parameters; this array is initialized with default values during the parsing process and the entries are replaced with new values as the attributes are parsed. Each entry is either a function or a constant value.

Functions are represented by objects of type `SV_Func`, which is again a pointer to a structure. The information stored in this structure includes which of the (pre-defined) animation functions is represented and the values of each of the function arguments. Values are represented by objects of type `SV_Value`; values can be integers, double-precision floating point numbers, the special animation constant *t_max*, or lists of any of the above.

Animation tuple interpretation. After a new animation space is read and stored, SwarmView begins producing the frames represented by the animation space. Production of frames continues until all frames described are produced (i.e., from frame counts 0 through the special number *t_max*). At this point, SwarmView is once again ready to issue a step command and read a new animation space; if the user does not choose to issue a step command, SwarmView will continue to produce the last frame of the animation.

When a frame is produced, the linked list of `SV_Object` structures is processed. For each object, every parameter is examined. If a function is defined for the parameter, the function is evaluated at the current frame count (time) using the appropriate C routine and the resulting value is stored in the parameter. The result of the processing is a list of objects, each having a value for each parameter. The objects are then ready for rendering.

Rendering. Rendering refers to the transformation of the object descriptions to graphical form. The process of rendering in SwarmView is greatly simplified by the hardware capabilities of the Personal Iris and the software capabilities of the graphical library. The Personal Iris hardware provides many useful capabilities in hardware, including: three-dimensional calculations; clipping and perspective transformations; rendering of lines and filled polygons; lighting calculations; and Z-buffering, which automatically performs hidden-surface removal.

Each type of graphical object has an associated C routine which renders the object. Each object in the list is rendered by the appropriate routine, which extracts the calculated parameter values from the object and uses the graphics library functions to add the object to the image. Use of the Z-buffer means that the objects do not have to be sorted (into, for example, back-to-front order) for rendering; object occlusions and intersections are properly handled by the hardware.

6. Conclusions

This paper 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. The model was used as the basis for constructing an experimental program visualization environment called Pavane. Our research and experimentation with Pavane brings forth additional evidence that declarative visualization is a viable and attractive paradigm. In particular, this paper shows that a strict declarative approach, based on mapping states to images, is able to handle elegantly both history-dependent attributes and special effects essential to effective visual communication. The decomposition of the visualization mapping is shown to be useful not only as a complexity control tool but also as a means of capturing important aspects of the visualization methodology. For instance, in this paper the decomposition favors a process which first abstracts the program state up to the point where a three dimensional world of abstract objects is conceptualized; this world is gradually materialized in term of a concrete visual representation; and, finally, visual events are explicated through the use of highlighting, movement, etc. Additional guidelines are supplied by heuristics that substitute geometric, temporal and other image properties for properties of the underlying computation. The approach described in this paper is but the first step toward transforming the art of program visualization into a disciplined intellectual exercise.

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 Parallel and Distributed Systems* Vol. 1, No. 3, pp. 365-376 (July 1990).

[4] Dijkstra, E. and Scholten, C.S. , "Termination Detection for Diffusing Computations," *Information Processing Letters* Vol. 11, No. 1, pp. 1-4 (August 1980).

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

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

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

[9] Roman, G.-C. and Cunningham, H.C., "Mixed Programming Metaphors in a Shared Dataspace Model of Concurrency," *IEEE Transactions on Software Engineering* Vol. 16, No. 12, pp 1361-1373 (Dec 1990).

[10] Roman, G.-C., Gamble, R. F., and Ball, W. E., "Seeking Concurrency in Rule-based Programming," Technical Report WUCS-91-17, Department of Computer Science, Washington University in St. Louis (January 1991).

[11] Sharma, Sanjay, "Real-Time Visualization of Concurrent Processes," CSRD Report No. 999, Center for Supercomputing Research and Development, University of Illinois (September 1990).

[12] P.H. Winston, *Artificial Intelligence, 2nd Edition,* Addison-Wesley Publishing Company, Reading, MA (1984).