A Visualization Model for Concurrent Systems

Mark Astley and Gul A. Agha*

Open Systems Laboratory
Department of Computer Science
1304 W. Springfield Avenue
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
Phone: (217) 244-3087
Fax: (217) 333-3501
Email: {astley | agha}@cs.uiuc.edu

Keywords: Actors, Distributed Systems, Program Visualization

Abstract

Concurrent systems maintain a distributed state and thus require coordination and synchronization between components to ensure consistency. To provide a coherent design approach to concurrent systems, recent work has employed an object-based methodology which emphasizes interactions through well-defined interfaces. The Actor model has provided formal reasoning about distributed object systems. Nonetheless, due to the complex interactions among components and the high volume of observable information produced, understanding and reasoning about concurrent algorithms in terms of simple interactions is a difficult task. Coordination patterns, which abstract over simple interactions, are not biased by low-level event orderings and are the appropriate mechanism for reasoning about concurrent algorithms. We outline a methodology for visualizing coordination patterns in concurrent algorithms which emphasizes observable interactions and causal connections between objects. We introduce *visualizers* as a linguistic mechanism for mapping coordination patterns to visualization. Visualizers are specified separately from algorithm code and thus respect code integrity. Moreover, visualizers may be implemented strictly in terms of object interfaces and thus preserve object encapsulation.

*Author for contact.

1 Introduction

Two distinctive features of today's concurrent systems are their distributed nature and their emphasis on interactions through well-defined interfaces. Because state is distributed in such systems, coordination and synchronization are needed in order to ensure consistency. *Coordination patterns*, which consist of low-level interactions, synchronization, and local state change, drive any distributed computation. However, due to the complex interactions among components and the high volume of observable information produced, attempting to understand and reason about concurrent algorithms in terms of simple interactions is a difficult task. *Program visualization*, the animated display of various aspects of algorithm execution, has been utilized in an attempt to cope with this complexity [7, 17]. In particular, program visualization has been applied to such diverse applications as computer science instruction [5], visual debugging [12], program verification and reasoning [7], and educational software [8]. Typical visual environments use pictorial abstractions to represent program components and their interactions, showing the various stages of a program in execution. Program visualization is particularly important for understanding concurrent applications where the semantic behavior of an algorithm corresponds to many different low-level event orderings. Specifically, visualization may provide abstraction mechanisms which capture high-level behavior whereas typical analysis tools tend to be biased to representing low-level execution details. Thus, a comprehensive environment for visualizing modern concurrent systems requires mechanisms for specifying visualization in terms of communication and coordination patterns. Because they fail to emphasize coordination mechanisms, current visualization environments are inadequate. We present the *causal interaction model* for visualizing concurrent systems which emphasizes coordination behavior. Similarly, unlike most contemporary environments, we demonstrate that the causal interaction model allows a transparent implementation separating visualization design concerns from algorithm code.

Current visualization environments adopt the view that program visualization represents a mapping from computational state to visual representation [14]. Constructing this map involves the following three tasks: first, identifying interesting program states; second, defining visual representations corresponding to these states; and finally, defining a mapping mechanism which links program state to visual representation. We call this the *state-transition approach*. Under the state-transition approach, visualization is synchronized with the transition of a program among computational states. Thus, when used to visualize concurrent execution, the state-transition model requires a global snapshot of algorithm state. Unfortunately, in distributed environments global snapshots are costly due to distributed state and asynchrony, and may not correspond to any state entered by the underlying execution [4]. Moreover, semantically equivalent execution behavior

may yield different state transitions. As a result, the state-transition model is costly to implement and does not effectively abstract over the relevant behavior in distributed systems.

In light of the observations above, we have developed a new model for program visualization, the *causal interaction model*, which synchronizes visualization with coordination patterns. Specifically, our model supports:

Generality. We may visualize sequential components and their interaction patterns in distributed systems.

Consistency. Visualization preserves the causal order of events that it represents.

Flexibility. Events which trigger visualization range from local component interactions to arbitrary patterns involving interactions among distributed components. The set of visualized components may be dynamic.

Transparency. Visualization mechanisms are both specification and execution transparent to the system being visualized:

Specification. Object integrity is preserved. Visualization is specified separately from algorithm code.

Execution. Low-overhead event detection mechanisms are used. Synchronization properties among components are not altered.

We discuss each of these features below.

By allowing generality while ensuring consistency, the causal interaction model encompasses visualization of general distributed systems which preserves the characteristic features of the underlying execution. In particular, reasoning about coordination behavior requires preserving the causal relationships among interacting components. Causal order can be determined succinctly in terms of the partial order of events in a distributed system [9]. Thus, the causal interaction model guarantees consistency by requiring that visualization observe the same partial order of events as that of the algorithm execution. Moreover, as we illustrate in Section 3, visualizing coordination behavior requires flexibility in specifying both the events which trigger visualization as well as the set of components to be visualized. Specifically, we require the ability to specify visualization for a possibly dynamic set of algorithm components and their interaction patterns. Hence, the scope of the causal interaction model is such that abstract patterns of interaction may be visualized over arbitrary (*i.e.* dynamic) groups of components. Finally, an important aspect of a visualization tool is that it not introduce further complication into a system. In particular, visualization should be specifiable over algorithms without side-effects; algorithms should retain approximately the same execution behavior regardless of whether or not they are being visualized. Large performance overhead affects message passing and

may mask race conditions. The causal interaction model naturally separates visualization design objectives from the system under analysis by allowing transparent implementation.

We model distributed systems as systems of Actors [1]. Actors provide a general and flexible object-based model of concurrency. In particular, common concurrent abstractions can be constructed using a system of actors. Actors may coordinate by exchanging messages and updating local behavior according to specific protocols. Typically, a relatively large number of messages are exchanged in order to implement some coordination activity (*e.g.*, message rounds in a commit protocol). The coordination behavior of an actor system may be characterized by three activities: message passing, dynamic object instantiation, and local state change. Thus, the causal interaction model represents coordination behavior by keying visualization to abstract sets of messages, dynamic object instantiation, and local state change. We may provide linguistic support and implementation mechanisms for our model so that only access to actor interfaces is required.

In the remainder of this paper we develop our model of program visualization and demonstrate its applicability by way of expressive language constructs and examples. Section 2 describes the Actor model of concurrent computation. Section 3 presents an example of a computation expressed using actors together with an example of how we might visualize this computation. In Section 4 we develop our model of program visualization. Section 5 discusses implementation issues associated with realizing our model. In Section 6 we contrast our model with other approaches. We summarize our results in Section 7.

2 Computational Model

The applicability of a visualization model depends to a large degree on the applicability of the model of concurrent computation upon which it is built. Thus, to capture the most general and flexible notion of concurrency, we base our model of program visualization on Actors [1]. Actors are encapsulated, interactive, autonomous components of a computing system that communicate by asynchronous message passing. Conceptually, an actor consists of a unique *name*, a *mail buffer* to receive messages, and a *behavior* which determines an actor's response to each message. An actor behavior consists of a list of *acquaintances*, which represent names of other known actors, and a *script* containing method definitions for responding to messages.

Actors compute by serially processing messages queued in their mail buffers. An actor blocks if its mail buffer is empty. Each message invokes a specific method within an actor. Within the body of a method, there are three basic actions which an actor may perform that affect the concurrent computational environment:

- *send* messages asynchronously to known acquaintances,

- *create* actors with specified behaviors, and
- *become* a new behavior which is used to respond to the next message.

Communication is point-to-point and is assumed to be weakly fair: executing a *send* eventually causes the message to be buffered in the mail queue of the recipient although messages may arrive in an order different from the one in which they are sent. Actor names are first class entities which specify the mail address of an actor and may be communicated within messages allowing dynamic reconfiguration of the communication topology. The *create* primitive creates a new actor with a specified behavior. Initially, the new actor is an acquaintance only of the creating actor (*i.e.* only the creating actor knows the name of the new actor). As described above, the name of the new actor may be communicated to other actors. The *become* primitive causes the invoking actor to change behavior. In particular, upon reaching a *become* statement the actor assumes the new behavior and processes the next buffered message.

The three actor primitives defined above provide a simple yet powerful mechanism for expressing concurrency. External concurrency is provided by asynchronous *send* and the ability to *create* new actors. Internal concurrency may be mimicked by creating a new actor to asynchronously process the remainder of the current computation while the original actor begins processing a new message. Actors provide a model of concurrent computation upon which a wide variety of concurrent abstractions can be developed [2]. In particular, actors can be used to model systems of objects communicating through well-defined interfaces. Thus, the causal interaction model, which is defined in terms of actor computation, may be applied trivially to more restrictive object-based models.

3 An Example

We motivate our model of program visualization by way of example. We identify the salient features of concurrency presented in the example and discuss the implications of these features in the next section. The example is presented using a C++-like syntax. Each actor definition contains a collection of methods (*c.f.* C++ class methods). The operator -> invokes an asynchronous message send.

The example, presented in Figure 1, illustrates a distributed implementation of mergesort. This example demonstrates how asynchrony and distributed state can be used to implement a mergesort "server." The server accepts requests for sorting an array of integers and services each request by either sorting the array (if it consists of two elements or less) or creating a merge "worker" and resubmitting the two halves of the

array to itself. The merge worker then receives the result of each of the recursive calls to the server and merges the results to yield the final sorted array. Note that the mergesort server may process several requests from several clients at the same time. That is, it is unnecessary for mergesort to completely sort an array before processing the next sort request.

```
actor Mergesort {
 Sort(array v, actor client) {
  int size=v.size;

  if (size <= 1)
   client->result(v);
  else if (size == 2) {
   if (v[1] < v[2])
    client->result(v);
   else {
    array tmp(2);
    tmp[1]=v[2];
    tmp[2]=v[1];
    client->result(tmp);
   }
  } else {
   actor merger;

   merger=new MergeWorker(client);
   self->Sort(v.copy(1, size/2), merger);
   self->Sort(v.copy(size/2+1, size), merger);
  }
 }
}
```

```
actor MergeWorker(actor client) {
 array one_half;
 bool rec_one=false;

 result(array r) {
  array res;

  if (rec_one) {
   res=Merge(one_half,r);
   client->result(res);
  } else {
   rec_one=true;
   one_half=r;
  }
 }
}
```

Figure 1: *Mergesort Example.* Actors are used to implement each component of the sorter. The syntax is C++-like and the operator -> invokes an asynchronous message send.

The mergesort example is fairly simple. However, it serves to illustrate several key features of concurrent algorithms which we wish to capture in visualization:

- **Distribution:** Because the task of sorting is distributed among several worker components, the "state" of a particular call to mergesort is distributed among the workers which will eventually service the request and messages currently in the server's mail queue.
- **Dynamic creation of resources:** Worker components are created as needed; the creation of components allows load to be distributed.
- **Coordination patterns:** All worker components are identical. However, the context in which they are

used in performing mergesort is defined by the source of their partially sorted arrays and the destination of their merged results. That is, their context is defined by coordination patterns.

Although we are somewhat limited by the medium, it is useful to consider how one might visualize the mergesort example. As suggested by the features we have identified above, it is important to capture how the problem of sorting an array is distributed among groups of worker components. We also wish to identify how different mergesort workers will coordinate to produce a sorted array. Figure 2 shows how one might visualize the mergesort example.

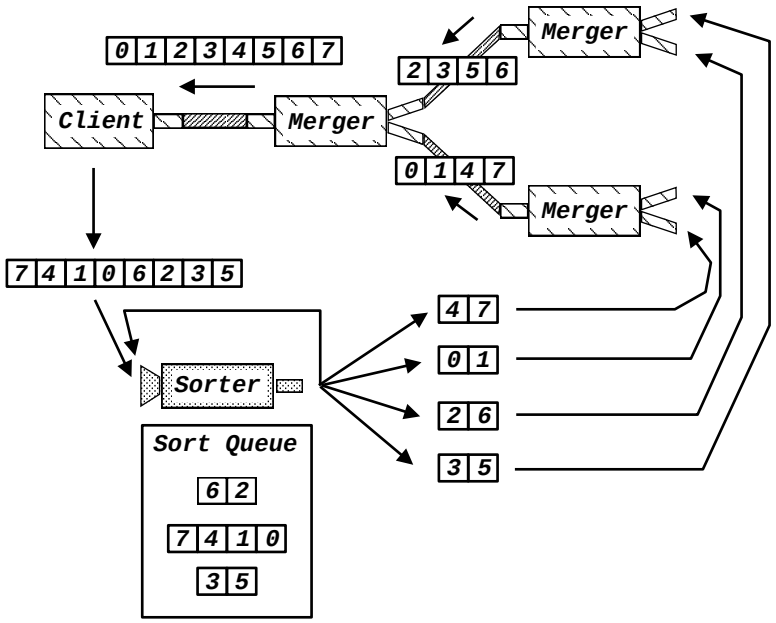

Figure 2: *Visualization of Mergesort.* This figure represents several steps of a program visualization of the mergesort example. Arrows indicate the animation paths of array segments.

In Figure 2, the sorter actor is represented by the structure labeled "Sorter." `MergeWorker` actors are represented by visual abstractions labeled "Merger." Mergers which coordinate to sort an array are all dithered with the same pattern and linked to show how each merge sort request is satisfied. Mergers are created by the sorter and added dynamically to the sorting tree for a particular client. In this visualization, segments of arrays are animated as they are sent from the client (*i.e.* an actor submitting an array to be sorted) to the sorter. Array segments waiting to be sorted are shown in the list below the sorter server. Array segments are animated as they emerge from the sorter and either re-enter the waiting list or are sent

to a merger.

The mergesort visualization captures the three features of concurrent algorithms we identified above. The distribution of each call to mergesort and the dynamic creation of resources is captured by building a sort tree for each request. The animation of array segments moving among components illustrates coordination mechanisms; links between mergers and the client suggest how a group of mergers will coordinate to satisfy a sort request. Although the correspondence between visualization and algorithm execution is fairly obvious, it is not intuitively clear how we may specify visualization of coordination in general. In the next section, we develop the causal interaction model which emphasizes visualization of the key features we have outlined above and thus provides mechanisms for visualizing coordination.

4 Visualizing Concurrent Programs

We formulate visualization of coordination behavior in terms of *actor events*. An actor event may be one of message interaction, local state change, or dynamic creation of new actors[1]. Our model of program visualization relates *visualization events* to *visualization actions* by way of a *visualization mechanism*. Each of these terms is defined below:

Visualization Event: A visualization event corresponds to a pattern of actor events. Visualization events are used to indicate configurations of the system at which visualization should take place.

Visualization Action: A visualization action corresponds to some rendering or animation activity which updates the current display of an algorithm. Typically, a visualization action is parameterized by the visualization event which invokes it.

Visualization Mechanism: The visualization mechanism specifies the relationship between visualization events and visualization actions. In particular, the visualization mechanism is responsible for detecting visualization events and determining the appropriate visualization action to trigger.

Figure 3 illustrates the relationship among each of these components.

Our task is to define our model for program visualization so that each of these components is specified in a manner consistent with the goals stated in Section 1. For expository purposes, we develop the causal interaction model in two stages. We first develop a basic model which supports visualizing actor events

[1] Formally, actor events are only *receive* events [3]. However, for the sake of clarity we abuse terminology here.

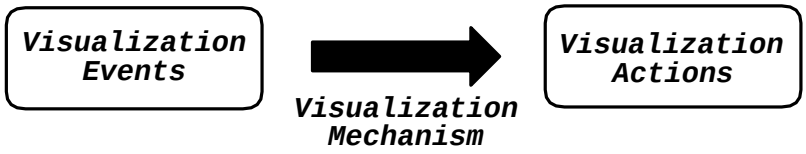

Figure 3: *Program Visualization.* Visualization events are mapped to visualization actions by the visualization mechanism.

and satisfies our requirements for consistency. We then define the causal interaction model by augmenting the actor event model with mechanisms for detecting patterns of actor events in a manner which preserves consistency. Note that we only develop the linguistic constructs and implementation techniques that are necessary for detecting visualization events and triggering visualization actions; we assume the availability of some suitably verbose graphics environment for specifying visualization actions.

4.1 Actor Event Visualization

We define a model for visualizing actor events in terms of mechanisms for event detection and triggering visualization actions. The actor event model captures events by distributing the visualization mechanism and ensures consistency by requiring causal delivery of actor events. Formally, events which trigger visualization in the actor event model satisfy the following definition:

Actor Event: An actor event corresponds to one of the following:

- A message send or reception.
- Local component state change (*i.e.* invoking **become**).
- Dynamic resource creation (*i.e.* invoking **new**).

Note that actor events are completely detectable on a local basis. Moreover, actor events correspond to the relevant local activities associated with coordination among components.

Detecting actor events on a local basis eliminates the necessity of querying global state. Moreover, as we demonstrate in Section 5, actor events may be detected transparently. In particular, we detect actor events by distributing the visualization mechanism so that each actor is observed by an independent *observer*. Observers are objects which filter actor interactions and trigger visualization when specific actor events are detected. However, because observers are distributed entities, visualization actions are triggered in

an asynchronous fashion. Specifically, causally related actor events may trigger visualization out of order. Thus, to guarantee that the basic model can be used to reason about causal behavior we enforce the following restriction on visualization actions:

Causal Connection Restriction: The invocation order of visualization actions must preserve the observable causal order of actor events which trigger them.

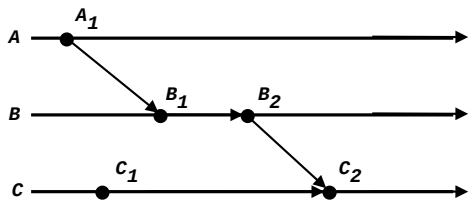

Figure 4: *Event Diagram.* Three actors A, B and C are shown together with visualization events each detects.

Figure 4 illustrates how the causal connection restriction affects visualization. The event diagram displays a total order of actor events for three actors A, B and C. The causal connection restriction states that if two actor events are causally connected, then their associated visualization actions must be invoked in a causally consistent order. Thus visualization actions corresponding to events A_1 and B_1 (which we call $v(A_1)$ and $v(B_1)$ respectively) are causally connected. Moreover, $v(B_1)$ may not be invoked until $v(A_1)$ is. However, there is no causal connection between events A_1 and C_1 thus $v(A_1)$ and $v(C_1)$ do not restrict one another. A more subtle relationship is that between C_1, C_2 and A_1. In this case, $v(C_2)$ must wait for the execution of both $v(A_1)$ and $v(C_1)$ (as well as $v(B_1)$ and $v(B_2)$).

The detection of actor events locally, together with the causal connection restriction completely defines our model for actor event visualization. To summarize:

Actor Event Model:

- Actor events are defined to be message send or reception, state transition and dynamic instantiation events.
- Actor events are detected locally at each component.
- Invoked visualization actions are executed according to the *causal connection restriction.*

Figure 5 illustrates the actor event model. We distribute observers (labeled *Obs*) over each actor to detect actor events and invoke visualization actions. The *visualization monitor* represents the modeling and rendering environment and generates the user display.

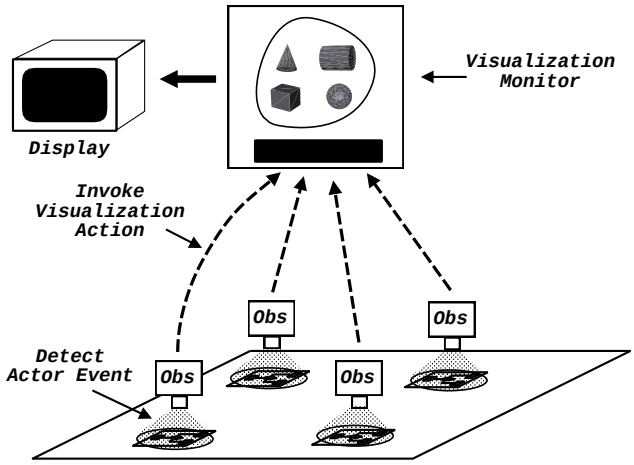

Figure 5: *Actor Event Model.* The visualization mechanism is distributed and only detects local events. Visualization actions are invoked in the visualization monitor in an order consistent with the causal connection restriction.

4.2 Causal Interaction Model

Constraining visualization according to the causal relationships of local events makes the actor event model simple and elegant but also rather limited. In particular, the actor event model is only useful for visualizing the simplest coordination schemes among individual actors. Recall that because tasks are distributed in the mergesort example, *groups* of actors may represent single algorithmic abstractions; coordination patterns among groups of actors establish the context of their behavior. Thus, we wish to extend the actor event model to allow visualization actions to be invoked based on abstractions which represent patterns of actor events. In particular, we need to define mechanisms so that visualization events, which are represented by patterns of actor events from possibly distributed actors, can be detected and trigger visualization.

A visualization event is defined inductively over actor events by adding guards or conjoining visualization events. Specifically, a visualization event is either an actor event or consists of:

Guarded Events: Visualization events guarded by a predicate.

Set of Visualization Events: A finite set of visualization events.

For example, if two actors participate in message passing, the send event at one actor and the receive event at the other may be composed to form a single visualization event.

In order to capture group behavior we organize actors into *visualization groups* which specify visualization events in terms of patterns of member actor events. Moreover, visualization groups associate state with local event patterns to facilitate temporal and guarded visualization events. Formally:

Visualization Group: A visualization group is defined as the actors over whom a set of visualization events are specified. A visualization group maintains state which may be referenced and modified by visualization actions triggered by visualization events specified over the group. In particular, predicates in guarded events are evaluated over visualization group state.

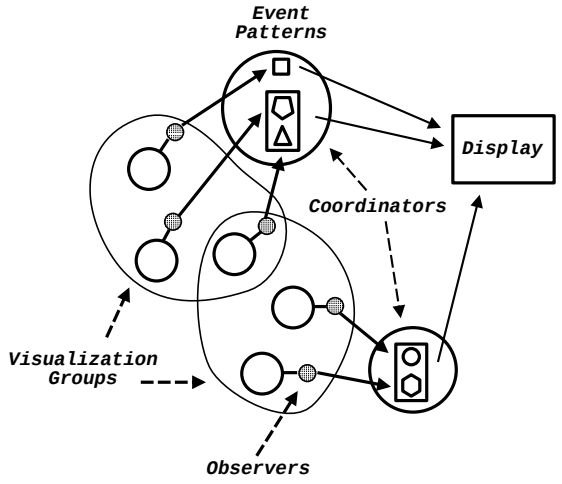

Figure 6: *Visualization Groups.* Visualization groups specify visualization events over possibly overlapping groups of actors.

Figure 6 illustrates the functionality of visualization groups. We augment the actor event model with visualization groups by introducing *coordinators* which are centralized components of the visualization mechanism. Coordinators receive actor events from observers and trigger visualization actions when visualization events are detected. We call this augmented version of the actor event model the *causal interaction* model which is defined as follows:

Causal Interaction Model:

- Actors are organized into visualization groups.
- Visualization events are defined as patterns of actor events over the set of member actors.
- Visualization events are detected at coordinators.
- Visualization actions are invoked according to the *causal connection restriction.*

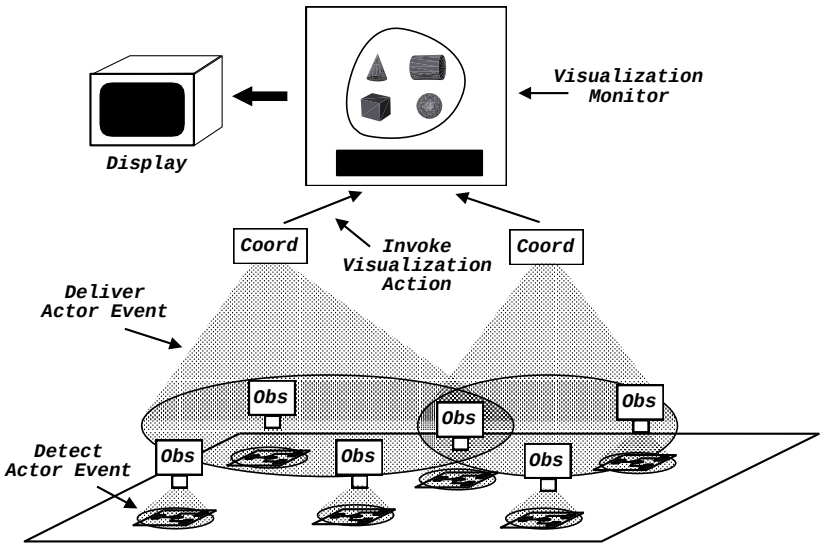

Figure 7: *Causal Interaction Model.* Centralized components of the visualization mechanism called *coordinators*, are introduced to form *visualization groups.* Local events are combined into patterns to form abstract visualization events.

Figure 7 illustrates the organization of the causal interaction model. The causal connection restriction is enforced by coordinators when invoking visualization actions. That is, visualization actions triggered by visualization events are invoked in an order consistent with the causal order of the underlying algorithm execution. However, since visualization events are specified in terms of local actor events, we must also preserve the causal order of actor events when reporting them to coordinators. Thus, observers causally deliver actor events to coordinators.

The definition of visualization events in terms of visualization groups and the requirement that causal orders be preserved in visualization provides a coherent model for reasoning about coordination. By forcing visualization actions to preserve the partial order of the visualization events which invoke them we may use visualization to reason about the causal interactions among components. The resulting emphasis on causal

connections shifts the focus of visualization to communication and coordination which serves as the driving mechanism in any distributed computation.

5 Implementation Issues

A significant consequence of triggering visualization based on patterns of actor events is that we may treat actors as black boxes for the purpose of visualization. Specifically, the causal interaction model allows detection of visualization events by way of monitoring component interactions rather than component internals. In Section 5.1 we introduce a mechanism by which actors may be observed transparently. Moreover, we specify how this same mechanism may be used to implement the causal connection restriction. In Section 5.2 we develop linguistic support for specifying visualization separate from component code.

5.1 Observing Interactions

Observers are the primary mechanism for implementing visualization groups over a set of actors. Intuitively, observers are actors that monitor actor interactions locally and deliver appropriate actor events to *coordinators*. As a special case of actor interaction, we view invocations of **new** and **become** primitives as calls to a run-time library which we may observe. Thus, the visualization mechanism may be installed transparently by creating an *observer* for each algorithm component. Figure 8 illustrates this notion.

Observers act as conduits for all interaction between an observed actor and the rest of the system. Implementing observers requires modifying the communication behavior of observed actors so that all communication is filtered through an observer. More importantly, observers must view interactions in the *same order* as the component they observe. This suggests an implementation in which components and their observers are indistinguishable entities. The transparent implementation of protocols for guaranteeing fault-tolerance requires similar constructs [19]. In particular, reflective language attributes have been used to modify communication behavior [2].

Recall that preserving the causal order of actor interactions is necessary in order to create visualization which can be used to reason about global behavior. Thus, we require implementation mechanisms which maintain a representation of the causal order of events and ensure that invoked visualization actions are *causally delivered*. Causal delivery can be guaranteed by implementing a vector clock protocol [4]. In particular, we wish to guarantee that invocations of visualization actions are causally delivered to the visualization monitor and hence executed in causal order.

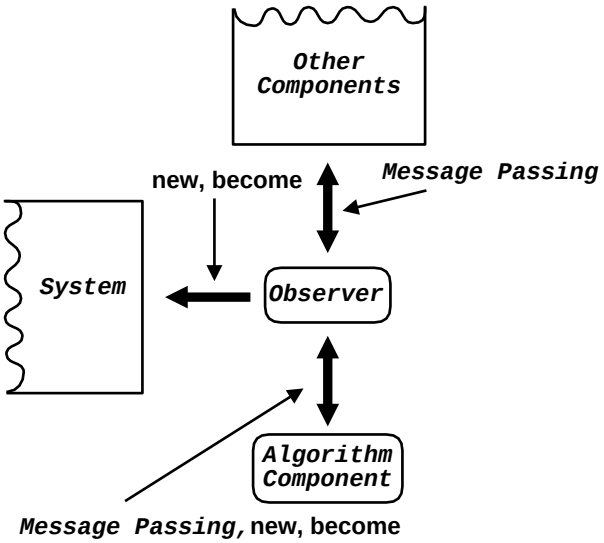

Figure 8: *Observers.* Observers filter interactions between actors.

Note that because observers are installed on each actor, they are the natural choice for transparently implementing a vector clock protocol. In addition to filtering interactions, each observer is also responsible for annotating outgoing messages with vector clock information. When a component executes some activity which corresponds to a local actor event, the observer updates its vector clock and sends the event to the appropriate coordinator. At the coordinator, actor events are causally delivered according to the vector clock protocol maintained by the observers. Since several coordinators may exist (*e.g.* corresponding to separate visualizers), a separate vector clock protocol is implemented among coordinators for delivering visualization actions to the visualization monitor.

Figure 9 presents an example illustrating the effect of causal restrictions between observers and coordinators. Four actor events occur in the figure corresponding to the send and reception of messages m_1 and m_2 (denoted $S(m_1)$, $S(m_2)$, $R(m_1)$, and $R(m_2)$ respectively). A_1, A_2 and B_1 represent observers, and A and B represent their corresponding coordinators. The partial order of each actor event is shown in the figure. Events $A_1(S(m_1))$ and $A_2(R(m_1))$ reported by the observers are combined in A to form the visualization event $A(S(m_1), R(m_1))$ which represents a message exchange between the two communicating algorithm components. Event $B_1(R(m_2))$ represents the reception visualization event of message m_2. Causal delivery guarantees first that events are reported to A in the appropriate order, and second that visualization actions invoked by A and B are executed in the appropriate order.

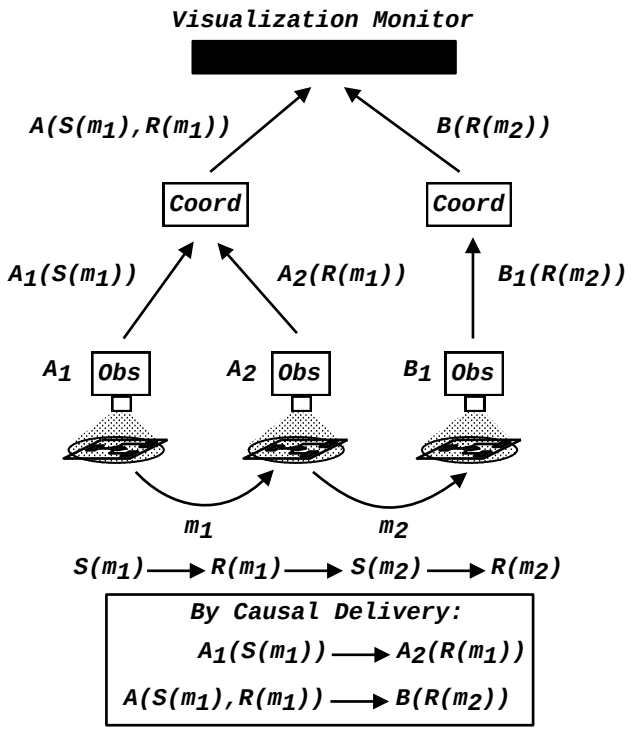

Figure 9: *Causal Delivery of Visualization Actions.* Visualization events are causally delivered for invoking visualization actions. Visualization actions are executed in causal order within the visualization monitor.

Note that requiring that causal connection information be maintained entails both performance and storage overhead. Vector clocks are only one implementation and may not always be the most suitable. A variety of different techniques exist for guaranteeing causal delivery in various contexts [13]. However, as long as the number of actors remains modest, vector clocks provide suitable functionality to demonstrate the implementation.

5.2 Linguistic Support for Specifying Visualization

Visualization groups represent the key abstraction for specifying visualization over a group of actors. We define visualization groups linguistically in terms of *visualizers*. Visualizers are language constructs similar to classes in an object-oriented language. That is, visualizers capture a particular visualization paradigm for a group of actors. Visualizers define visualization events over a group of actors by specifying interaction rules. An interaction rule is matched and invoked when a particular interaction pattern is detected over members of a visualizer. We demonstrate our constructs by specifying a portion of the mergesort visualization presented

in Section 3.

```
visualizer Visualizer Name {
  Local State Variables

  begin enter
    Membership rules invoked when components are added
  end enter

  begin exit
    Membership rules invoked when components are removed
  end exit

  begin actions
    Message actions invoked according to patterns of messages
  end actions

  begin create
    State-change rules invoked when members execute new
  end create

  begin become
    State-change rules invoked when members execute become
  end become
}
```

Figure 10: *Visualizer Specification*

Figure 10 provides a template describing the specification of a visualizer. Visualizers maintain a list of members, local state, and specify rule blocks which define the visualization events detected over member actors. Note that members of a visualizer may be added or removed dynamically. Moreover, a single component may be a member of more than one visualizer; visualizer member lists may overlap. Rule blocks define three types of rules: membership rules, message actions, and state change rules. When a member of a visualizer exhibits an appropriate behavior, each rule in the related rule block is evaluated in order until a rule matches or the list of rules is exhausted. Only the first matching rule is invoked.

Membership rules are used to invoke visualization actions in response to membership changes in a visualizer. State change rules are invoked when member components change state using **become**, or when some member of a visualizer instantiates a new actor using **new**. State change rules invoked when new actors are created may specify the visualizer membership of the new actor. Both membership rules and state change rules define visualization in response to the dynamically changing computational environment. In particular, membership rules establish relationships between graphical abstractions and the components (or groups of

components) they represent. Similarly, state change rules allow visualizers to maintain these abstractions in response to new system components.

Message action rules are used to invoke visualization actions in response to the detection of message patterns. A message action specifies a message pattern to detect and a corresponding visualization action to invoke. Message patterns are specified using basic patterns which represent the interaction of two components. More complex patterns are created from basic patterns using guards and conjunction of patterns. Because rules are evaluated in the order they are specified, detecting a disjunction of patterns is implicit.

To illustrate how visualizers are specified syntactically, consider the mergesort visualization from Section 3. Due to space limitations and to enhance clarity, we only describe a visualizer which specifies visualization events for sorter interactions. The sorter interacts with three types of component: clients, mergers, and itself. For example, clients interact by sending `Sort` requests. We want to describe a visualization event and a corresponding visualization action for each of these interactions. In particular, we need to describe the following aspects of the sorter visualizer:

- Rules for creating a visual representation when a sorter is added to the visualizer.
- Rules for invoking visualization triggered by interactions with other components.
- Rules for invoking visualization triggered by creating new `MergeWorker` components.

Figure 11 gives the abstract code for a sorter visualizer. For brevity, visualization actions are specified in italicized pseudo-code. We create a visual representation for the sorter when it is added to the `Sorter` visualizer by creating a membership rule. The rule

```
on Mergesort do
  Create representation for sorter object and sort queue
end
```

matches when a component of type `Mergesort` is added to the `Sorter` visualizer. The corresponding visualization action creates an appropriate visual representation for the sorter.

To handle interactions with other components, we need three action rules. The rule

```
me ← client : Sort(anArray, aClient) do
  Add visual representation of anArray to sort queue
```

```
visualizer Sorter {
 begin enter
  on Mergesort do
   Create representation for sorter object and sort queue
  end
 end enter

 begin actions
  me ← client : Sort(anArray, aClient) do
   Add visual representation of anArray to sort queue
  me → client : result(anArray) do
   Remove top element of sort queue. Animate delivery
    of result to client.
  (me → me : Sort(a1,dest1) and
   me → me : Sort(a2,dest2)) where dest1==dest2 do
   Remove top element of sort queue. Animate delivery
    of arrays to ourself.
 end actions

 begin create
  on MergeWorker(client) from me do
   join ClientMerger
  end
 end create
}
```

Figure 11: *Visualizer for Sorter*

matches when a member of the visualizer (*i.e.* a sorter) receives a `Sort` request. The ← indicates a reception event (a → would indicate a send event). The identifier `me` is bound to the member of the visualizer which receives the request while the identifier `client` is bound to the sender of the request. `Sort` indicates the name of the message and the identifiers `anArray` and `aClient` are bound to the appropriate arguments in the message. When this rule matches, the visual representation of the sort queue is updated to indicate the new request. The second action rule is similar to the first and matches when a member of the visualizer sends a result message back to a client.

The third action rule specifies an abstract visualization event which is triggered when a sorter recursively sends itself two new sort requests. This is necessary because when a sorter decides to divide an array and resubmit the requests it uses two messages to do so. The keyword **and** joins the two basic `Sort` events into an abstract event which only occurs if both basic events occur. The **where** keyword specifies a guard over the abstract event which requires the client arguments of both sort requests to be equal in order for the event

to be triggered. This rule demonstrates how abstract events may be created using guards and conjunction.

Finally, to invoke visualization when new `MergeWorker` components are created we specify a state change rule. The rule

```
on MergeWorker(client) from me do
 join ClientMerger
end
```

is invoked when a member of the visualizer creates a new actor with behavior `MergeWorker`. The identifier `client` is bound to the instantiation parameter used when creating the new actor. The identifier `me` is bound to the actor which caused the instantiation. The expression

```
join ClientMerger
```

specifies that the newly instantiated component should be added to the `ClientMerger` visualizer. Thus, this state change rule simply adds the newly created component to another visualizer which handles `MergeWorker` interactions.

Recall from Figure 2 that each new sort request causes a new `MergeWorker` to be created which corresponds to a "Merger" in the visualization. For large sorting requests, this could lead to a cluttered display. This case can be handled easily, however, by mapping several `MergeWorkers` to a single visual abstraction. Then a single visual element can be used to indicate the status of each sort request. A membership rule of the form

```
on MergeWorker(client) do
 if (client is a MergeWorker) then
   Update the appropriate existing display element
 else
   Create a new "Merger" display element
end
```

in the `ClientMerger` visualizer with appropriate action rules would provide this functionality.

The specification of the sorter visualizer reiterates how visualization may be specified within the causal interaction model without requiring access to component internals. Moreover, visualizers allow a straightforward implementation atop observers. In particular, state change and action rules are detected locally by observers installed on each member of a visualizer. Actor events detected by local observers are causally delivered to a coordinator component created for each visualizer. The coordinator combines actor events in

order to detect visualization events and invoke appropriate visualization actions. Moreover, the coordinator arbitrates access to visualizer state and moderates visualizer membership changes. Thus, by building on observers, visualizers may be installed transparently on concurrent algorithms.

6 Discussion

The field of program visualization is still relatively young. As a result, most recent efforts have concentrated on visualizing sequential program execution. Nonetheless, it is still useful to contrast and compare the causal interaction model with sequential systems in order to reveal differences in expressiveness and specification techniques.

Representative sequential environments include BALSA [5] and its descendent ZEUS [6], and TANGO [16]. Technically, these environments are not restricted to visualizing sequential programs. However, none of the named systems includes explicit mechanisms for dealing with concurrency. The strength of these systems tends to lie not in their visualization event detection mechanisms, but rather in their expressiveness in terms of visualization actions. We have tabled the discussion of visualization actions in this paper in favor of developing usable visualization event detection mechanisms. However, any realization of the model defined herein should include mechanisms for defining and animating arbitrary three dimensional shapes. BALSA and ZEUS provide perhaps the most complete mechanisms in terms of specifying arbitrary visual layouts. TANGO, on the other hand, contributes a natural and flexible animation facility using the notion of *path transitions* [15].

The sequential systems named above all use code annotation to identify visualization events (in BALSA these are called *interesting events*). In effect, visualization state and computational state are intermingled. As a result, visualization is produced as a side effect of algorithm execution. In contrast, the causal interaction model supports transparent realizations and requires no explicit code modification. Moreover, visualizers are specified separately from system components and hence respect object integrity. Although code annotation is undesirable from a software engineering perspective, overall it provides the most flexibility and allows the finest control of when to signal visualization actions. However, we have argued that coordination behavior is the most relevant attribute in concurrent systems. Our techniques demonstrate that code modification is not necessary to capture synchronization and coordination. Moreover, annotated code biases the resulting visualization to a particular execution history. By emphasizing patterns as a basis for visualization events, visualizers avoid bias and provide an abstraction mechanism for viewing interactions.

Of the few environments which do exist for visualizing concurrency, POLKA [17] and PAVANE [14] are the most relevant. POLKA is a descendent of TANGO intended for animations of programs executing on parallel architectures. POLKA is a relatively straightforward extension of the sequential model of TANGO for a concurrent setting; the notable addition is the support of concurrent, overlapping animation and more modular constructs for creating visual representations. However, a code annotation approach is still utilized and thus suffers from the various trade-offs mentioned above. Moreover, the designers do not specify how synchronization issues are handled in the system. Thus, it is not clear if and how the resulting visualization can be used to reason about program execution.

The PAVANE system represents a relatively coherent approach to visualizing concurrent program execution. Moreover, PAVANE has been designed explicitly to aid programmers in reasoning about program execution. In this system, concurrent algorithms are specified in a *tuple-space* environment. A configuration of tuple-space represents the current state of an algorithm execution. Visualization event detection follows a rule-based approach where visualization rules match based on the contents of tuple-space and create graphic representation tuples in a separate *visualization* space. Animation may be created by annotating tuples in visualization space with animation information. Note that PAVANE enjoys all the advantages of a rule-based approach. In particular, visualization rules do not interfere with algorithm code and are completely reusable.

The main differences between the causal interaction model and the PAVANE system are the model of concurrency and the expressiveness of the visualization event detection mechanism. The PAVANE model of concurrency is completely synchronized, thus global program state is readily obtainable. Changes to tuple-space in PAVANE are synchronized according to groups of executing "processes." Thus tuple-space (*i.e.* program state) may be sampled after each process group has completed execution. The causal interaction model, on the other hand, specifies visualization for distributed environments. Moreover, abstraction is difficult to define using the PAVANE mapping approach because transitions among computational states correspond directly to transitions among visualization states. In particular, abstractions expressed using temporal relations are difficult to describe. Visualization groups, on the other hand, maintain state making temporal relationships easy to detect.

From a somewhat different tradition than program visualization, event diagrams have been a prevalent mechanism for visualizing actor computation. Augmented Event Diagrams were used by Manning in the Traveler observatory to support the debugging of actor programs [10]. In a related fashion, predicate transition nets have been used to visualize actor computation [11]. However, both approaches suffer from two key weaknesses: there are no coordination abstraction mechanisms; and, representations rather than models

are generated. Event diagrams do not abstract over low-level execution details and tend to be unnecessarily complex. Predicate transition nets do not retain the history of the computation and only visualize actor behavior change. Moreover, both approaches fix the visualization mechanism and limit flexibility. The causal interaction model provides a foundation upon which explicit views of concurrent computation may be developed; visualization groups may be used to create both event diagram and predicate transition net representations.

7 Conclusion

The successful design and implementation of complex concurrent systems relies in large part on the ability to understand and detect errors in interactions among components. To cope with this issue, we have forwarded the concept of developing program visualizations of concurrent algorithm execution which can be used to reason about causal behavior and coordination.

We have advanced a model which emphasizes distributed detection of visualization events and captures coordination activity with minimal overhead. The causal interaction model distributes the visualization mechanism, but enforces a causal connection constraint on visualization actions to allow the resulting program visualization to be used to reason about system behavior. We introduce *visualization groups* as a technique for defining visualization events according to interactions over groups of actors. Visualization groups provide appropriate abstraction mechanisms for capturing both spatially and temporally defined coordination patterns.

In order to demonstrate the transparent realization of our model, we have presented *Observers* as a mechanism for detecting local actor events. An observer is installed on each actor and detects local actor events by observing interactions involving the base actor. Observers provide *execution transparency* by way of low-overhead filtering of interface invocation of the base actor. We guarantee the causal connection restriction by forcing observers to causally deliver visualization events when triggering visualization actions. Observers are the foundation for implementing *Visualizers* which are linguistic constructs that capture the functionality of visualization groups. Specifically, visualizers are group abstractions which maintain local state and isolate visualization events according to specific visualization paradigms. Visualizers are *specification transparent* in that they need only refer to the interfaces of member actors and hence may be specified completely separately from algorithm code.

Currently, we are in the process of implementing our model atop BROADWAY, a proto-type environment for

developing actor systems [18]. We have concentrated on specifying visualization event detection mechanisms rather than specifying explicit graphics modeling features. A more complete specification of a visualization model would include mechanisms for defining visualization actions as well. We leave this topic for future work.

Acknowledgments

We thank past and present members of the Open Systems Laboratory who aided in this research. Extra gratitude is extended to Daniel Sturman and Brian Nielsen whose comments were particularly useful in developing this work. Lastly, we thank the editors and referees for their insightful comments. The research described has been made possible in part by support from the Office of Naval Research (ONR contract numbers N00014-90-J-1899 and N00014-93-1-0273), by an Incentives for Excellence Award from the Digital Equipment Corporation, by Hitachi, and by the National Science Foundation (NSF CCR 93-12495).

8 References

[1] G. Agha. *Actors: A Model of Concurrent Computation.* MIT Press, 1986.

[2] G. Agha, S. Frølund, W. Kim, R. Panwar, A. Patterson, and D. Sturman. Abstraction and modularity mechanisms for concurrent computing. *IEEE Parallel and Distributed Technology*, May 1993.

[3] G. Agha, I. Mason, S. Smith, and C. Talcott. Towards a theory of actor computation. In *The Third International Conference on Concurrency Theory (CONCUR '92)*, pages 565–579, Stony Brook, NY, August 1992. Springer Verlag. Lecture Notes in Computer Science No. 630.

[4] Ö. Babaoğlu and K. Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. In S. Mullender, editor, *Distributed Systems*, chapter 4, pages 55–96. ACM Press, New York, NY, 1994.

[5] M.H. Brown. Exploring algorithms using balsa-ii. *IEEE Computer*, May 1988.

[6] M.H. Brown. Zeus: A system for algorithm animation and multiview editing. In *Proceedings of the IEEE Workshop on Visual Languages*, pages 4–9, 1991.

[7] G.-C. Roman et. al. Pavane: A system for declarative visualization of concurrent computations. *Journal of Visual Languages and Computing*, 3(2):161–193, June 1992.

[8] K. Kahn. Toontalk^{TM} – an animated programming environment for children. In *Proceedings of the National Educational Computing Conference (NECC'95)*, 1995.

[9] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[10] C. Manning. Traveler: the actor observatory. In *Proceedings of European Conference on Object-Oriented Programming*, January 1987. Also appeared in LNCS (276).

[11] S. Miriyala, G. Agha, and Y. Sami. Visualizing actor programs using predicate transition nets. *Journal of Visual Languages and Computation*, 3(2):195–220, June 1992.

[12] S. Mukherjea and J.T. Stasko. Toward visual debugging: Integrating algorithm animation capabilities within a source level debugger. *ACM Transactions on Computer-Human Interaction*, 1993.

[13] M. Raynal, A. Schiper, and S. Toueg. The causal ordering abstraction and a simple way to implement it. *Information Processing Letters*, 36(6):343–350, 1991.

[14] G.-C. Roman and K.C. Cox. A taxonomy of program visualization systems. *IEEE Computer*, December 1993.

[15] J.T. Stasko. The path-transition paradigm: A practical methodology for adding animation to program interfaces. *Journal of Visual Languages and Computing*, 1(3):213–236, September 1990.

[16] J.T. Stasko. Tango: A framework and system for algorithm animation. *Computer*, 23(9):27–39, September 1990.

[17] J.T. Stasko and E. Kraemer. A methodology for building application-specific visualizations of parallel programs. *Journal of Parallel and Distributed Computing*, 18:258–264, 1993.

[18] D.C. Sturman. Fault-adaptation for systems in unpredictable environments. Master's thesis, University of Illinois at Urbana-Champaign, January 1994.

[19] D.C. Sturman and G. Agha. A protocol description language for customizing failure semantics. In *Proceedings of the 13th Symposium on Reliable Distributed Systems.* IEEE Computer Society Press, October 1994.