ONLINE EVENT-BASED PROGRAM VISUALIZATION FOR DISTRIBUTED SYSTEMS

BY

MARK CHRISTOPHER ASTLEY

B.S., University of Alaska, Fairbanks, 1993

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1996

Urbana, Illinois

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. In this thesis, we present a methodology for visualizing coordination patterns in concurrent algorithms which emphasizes observable interactions and causal connections between objects. We introduce *visualization groups* as an intuitive notion for mapping coordination patterns to visualization. Visualization groups are specified linguistically using *visualizers*. 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. We describe the implementation of STAGEHAND, a prototype environment which supports visualizers for the purpose of specifying visualization over actor computations implemented on the actor platform BROADWAY.

Acknowledgements

First and foremost, I would like to thank my advisor, Gul Agha, for providing the foundational ideas behind much of my research. In addition, I would like to thank Helena Mitasova, Bill Brown, and the other fine folks at the U.S. Army Construction Engineering Research Laboratory for providing a source of funding and tolerating my moonlighting as a graduate student. I would also like to thank the other members of the Open System Laboratory, both past and present, for their input and contributions. In particular, I would like to thank Dan (Anybody want some coffee?) Sturman, Brian (How much coffee should I make?) Nielsen, Shangping (I don't want any coffee, honest) Ren, and Rajendra Panwar for their invaluable contributions to this work. I would like to thank Suzanne (you know who you are) for keeping me well fed and stress free for the duration. Lastly, I would like to thank my brother Scott and my parents for being there through good times and bad.

Table of Contents

Appendix

List of Figures

Chapter 1

Introduction

1.1 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 point-to-point 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. Moreover, conventional sequential tools do not readily extend to distributed systems.

As a simple example of how complexity in distributed algorithms establishes the need for special analysis tools, consider a distributed protocol such as two-phase commit. Two-phase commit may be expressed algorithmically as follows [6] (see Figure 1.1):

1. A *coordinator* sends the message **vote** to each component participating in the protocol.

2. Upon receiving **vote**, each component sends either the message **commit** or **abort** to the *coordinator*.

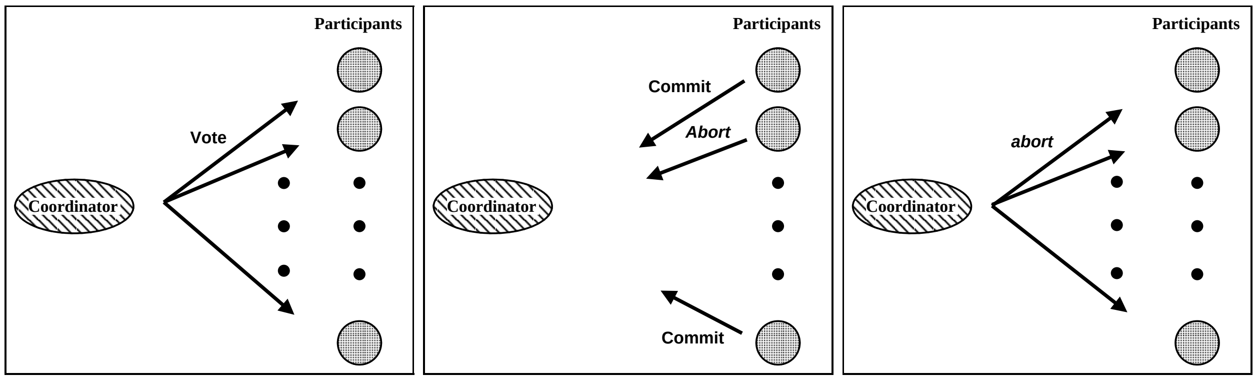

Figure 1.1: **Two-phase Commit.** The coordinator collects votes and decides on an action. All participants execute the same action.

3. After collecting all responses from all participating components, the *coordinator* broadcasts the message **commit** if all components voted to commit, and **abort** otherwise.

Two-phase commit proceeds in two stages delineated by rounds of message passing. Understanding and reasoning about two-phase commit requires the ability to correlate component state with phases of the algorithm. However, during the voting procedure, the *coordinator* can determine only when a stage has ended but can not establish the exact state of the protocol in the middle of a phase. Similarly, no participant can establish protocol state based solely on its interactions. Hence typical sequential analysis techniques which emphasize component-centric views are not sufficient. Such systems lack a *multi-component context* which correlates state over multiple components with their causal relationships established by interactions. Thus any analysis tool for distributed systems must be based on the *combined* state of all participating components and their interactions.

Similarly, components in a distributed system often require multiple interaction protocols. Typically, protocols will overlap (see Figure 1.2). As a result, interactions at a single component may consist of intermingled interactions involving several protocols. Thus, an analysis tool must provide abstraction mechanisms for identifying and separating interactions according to their semantic content. Moreover, large complex systems generate a large volume of observable

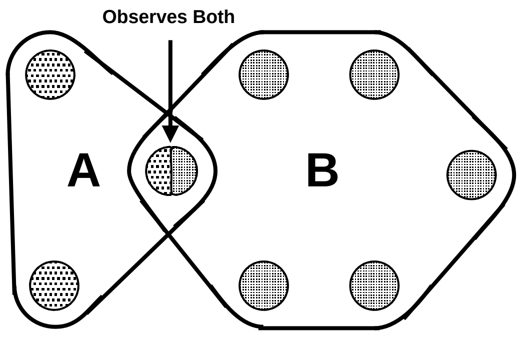

Figure 1.2: **Overlapping Protocols.** The components participating in protocol **A** overlap with those participating in protocol **B**.

information. Abstraction mechanisms must therefore exist for varying granularity and filtering interactions.

Program visualization, the animated display of various aspects of algorithm execution, has been utilized in an attempt to provide appropriate analysis tools [27, 32]. In particular, program visualization has been applied to such diverse applications as computer science instruction [9], visual debugging [21], program verification and reasoning [27], and educational software [16]. Typical visual environments use pictorial abstractions to represent program components and their interactions, showing the various stages of a program in execution. Note that a visual-based analysis tool provides a multi-component context. Moreover, extending such systems with an appropriate specification language allows analysis which filters specific interactions. That is, visualization may provide abstraction mechanisms which capture high-level behavior whereas typical analysis tools, such as trace-based systems, tend to be biased to representing low-level execution details.

Current visualization environments adopt the view that program visualization represents a mapping from computational state to visual representation [26]. 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 approach 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 [23]. Moreover, semantically equivalent execution behavior may yield different state transitions. As a result, the state-transition approach is costly to implement and does not effectively abstract over the relevant behavior in distributed systems.

A more natural model for visualizing distributed algorithms is one in which visualization is triggered according to the occurrence of *events* at participating objects. An event consists of a basic transition which effects global state; message passing and the dynamic creation of new objects are specific examples. We call such a visualization model an *event-based* approach. By considering patterns of local events, an event-based approach emphasizes coordination patterns and hence captures the salient behavior of distributed systems. Due to asynchrony and distributed state, only a partial order of events is available in a distributed system [17]. Thus in order for event-based visualization to be meaningful, some form of consistency must be enforced between the ordering of events and the visualization they trigger. In particular, correctly characterizing the causal relationships of the underlying execution is critical to providing a tool for reasoning about distributed execution [28].

In this thesis we develop an event-based environment for specifying and implementing visualization of distributed object-based systems. In our environment, visualization is synchronized with causally ordered coordination patterns among objects. Moreover, granularity is not fixed in our environment; visualization may be specified over simple low-level interactions as well as complex interaction patterns over a dynamically changing set of objects. Unlike most contemporary environments, our model allows a transparent implementation separating visualization design concerns from algorithm code. We demonstrate the utility of our model by way of STAGEHAND, a specification language and supporting run-time mechanisms for specifying *online* program visualization over a distributed computation. Online means that visualization is generated in real-time in response to events in the underlying execution. Using STAGEHAND, users define the set of interactions which trigger a particular visual transition as well as the specific manipulations of visual modeling entities. Specifically, STAGEHAND embodies the following set of design goals:

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.

By allowing generality while ensuring consistency, our environment 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 [17]. Thus, our environment guarantees consistency by requiring that visualization observe the same partial order of events as that of the algorithm execution. Moreover, 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 our environment 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. Our environment naturally separates visualization design objectives from the system under analysis by allowing transparent implementation.

The remainder of this thesis is organized as follows. This chapter discusses background information and related work: Section 1.2 discusses the Actor model of computation; Section 1.3 discusses the concept of reflection; Section 1.4 discusses related work in program visualization. In Chapter 2 we describe how our visualization model is defined over a distributed computation. In particular, we develop an event-based model for visualizing actor-based systems. In Chapter 3 we describe the specification language portion of STAGEHAND for defining visualization. Chapter 4 describes the implementation of STAGEHAND. We conclude in Chapter 5.

1.2 Actors

Our goal is to visualize encapsulated, possibly distributed, objects which interact via message passing. Actors [1] provide a general and flexible model of concurrency which captures all such systems, thus we base our visualization mechanisms on actor-based computation. Actors are encapsulated, interactive, autonomous components of a computing system that communicate by asynchronous message passing. Conceptually, an actor encapsulates a state, a thread of control, and a set of procedures which manipulate the state. Actors provide an interface which may be used to invoke encapsulated procedures. Procedures which are accessed through this interface are called *methods* (see Figure 1.3).

Actors coordinate by asynchronous message passing. Each actor has a unique *mail address* and a *mail buffer* to receive messages. Actors compute by serially processing messages queued

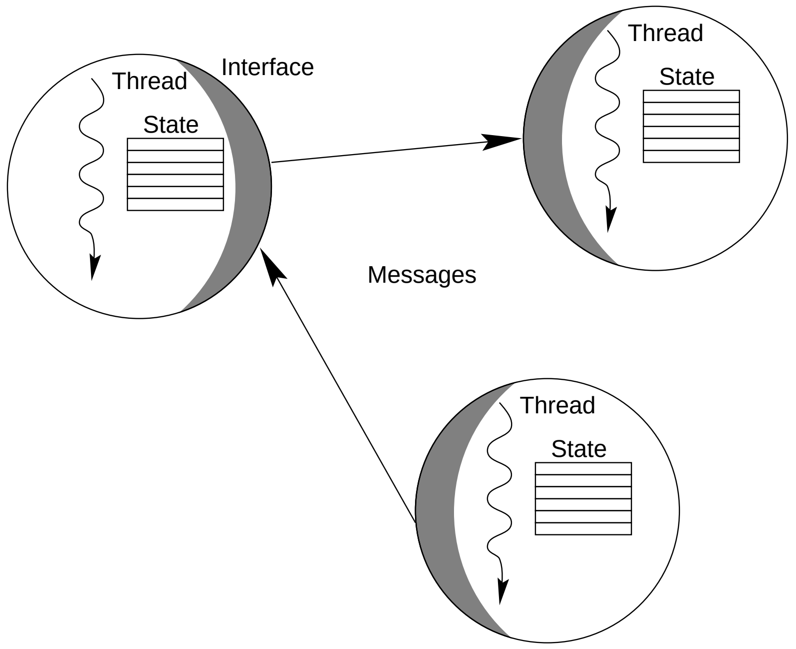

Figure 1.3: **Actor Model.** Actors encapsulate state and a thread of control. The interface is comprised of public methods which operate on the state.

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 two basic actions which an actor may perform that affect the concurrent computational environment[1]:

- *send* messages asynchronously to acquaintances, and
- *create* actors with specified behaviors.

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

[1]Those familiar with the Actor model may notice that **become** is missing. In recent work [5], **become** has been replaced by a continuation passing style transform and the specification of a replacement behavior. We assume a similar convention in this thesis.

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 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]. Hence, the visualization mechanisms described in this thesis readily extend to computational models supporting synchronous communication, remote procedure call, migrating processes, and so on. More importantly, actors provide a uniform view of concurrency: every object, including system-level objects, are actors. Our visualization mechanisms may be installed on any actor in a system, hence visualization may be used to monitor computations at any arbitrary level of abstraction ranging from system-level to high-level application specific interactions.

For the purposes of this thesis, we model actors as residing in one of two states: *ready* or *processing.* The *ready* state simply indicates that an actor is ready to process the next message in its mail queue. An actor is in the *processing* state when it is processing a message. These states serve no other purpose than to allow us to define two basic actor transitions. A *method dispatch* corresponds to the transition from the *ready* state to the *processing* state. Intuitively, the actor gets the next message off its mail queue and begins processing. A *method completion* corresponds to the transition from the *processing* state to the *ready* state. Intuitively, the actor has finished processing the current message and is ready for the next message. We make no assumptions as to when these transitions occur other than that an actor may only begin processing a message by being in the ready state and performing a method dispatch, and that after processing a message an actor must eventually transition to the ready state by performing a method completion. We also assume that all actors initially start in the ready state.

1.3 Reflection

Reflection refers to the ability of an object to manipulate a causally connected description of itself [18, 29]. Causal connection implies that changes to the description have an immediate effect on the described object. The reflective capabilities of a language are referred to as the *meta-architecture* of the language and are embodied by *meta-objects* which customize specific attributes of their *base-object.* For example, in the case of actors, a meta-object may be used to control how the mail buffer functions. Another meta-object might be used to control how new actors are created. Using reflection, such meta-objects may be customized at run-time and replaced with user defined meta-objects yielding a dynamically reconfigurable computation environment.

From the perspective of program visualization, reflection has the advantage of allowing flexibility while respecting object integrity. In particular, meta-level objects may be manipulated without requiring access to base-object internals. This feature will allow us to transparently install visualization on objects. Specifically, we reflectively manipulate the meta-level objects describing communication, method dispatch, and actor creation. Conceptually, we encapsulate these meta-level objects into an *observer*, a meta-level object which reports the occurrence of events we are interested in for visualization purposes. BROADWAY supports a limited form of reflection which allows the run-time customization of communication-related attributes using compiled objects. STAGEHAND specifications are compiled into appropriate observers which are installed on visualized actors at run-time.

1.4 Related Work

Most of the work related to the visualization of parallel and distributed programs has concentrated on performance analysis and instrumentation. The ParaGraph [15] and Pablo [25] systems are representative of this work. These approaches tend to emphasize largely application independent performance issues. In this thesis we focus our attention on *application-specific* program visualization. That is, we are primarily concerned with user-definable abstraction mechanisms which aid in the comprehension of concurrent algorithms. As such, we will not discuss visualization for tuning performance.

The majority of work on program visualization has been concerned with visualizing sequential program execution. Although we are concerned with visualizing parallel and distributed programs, it is still useful to contrast and compare with these sequential systems in order to reveal differences in expressiveness and specification techniques.

Representative sequential environments include BALSA [9] and its descendent ZEUS [10], and TANGO [31]. 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 specification mechanisms, but rather in their support of flexible and expressive modeling primitives for creating complex imagery. 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* [30]. The emphasis of this thesis is on specification techniques rather than modeling and rendering visualization. Hence STAGEHAND contains a rather basic set of primitives for generating images. Nonetheless, any complete visualization system should include a comprehensive modeling and rendering environment. The systems named above are prime examples of the type of flexibility which should be supported.

The sequential systems named above all use *code annotation* to identify visualization events (in BALSA these are called *interesting events*). That is, the programmer indicates, using special syntax, where in the source code visualization should take place. As a result, visualization is produced as a side effect of algorithm execution. In contrast, STAGEHAND supports transparent realizations and requires no explicit code modification. Moreover, visualization is implemented reflectively over source components and hence respects 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 trigger visualization. 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, STAGEHAND specifications avoid bias and provide an abstraction mechanism for viewing interactions.

The concept of synchronizing visualization with the causal relationships of the underlying execution has been recognized as critical to understanding concurrent algorithms. Turner and Cai have described a visualization mechanism based on the logical clock traces of interacting processes [38]. Similarly, the PVM [35] distributed computing environment has been extended with PVaniM [36] to support timestamp based visualization. The Conch [8] system has been extended in a similar fashion in [37]. All of these environments are based on *post-mortem* visualization. That is, events are recorded in a log during execution and visualized after the fact. Thus, a shallow difference between these systems and STAGEHAND is that STAGEHAND allows visualization during system execution. Note that many important and interesting distributed systems never "terminate". STAGEHAND was designed to be online in the interest of allowing visualization to be added dynamically at run-time. The latter two systems both employ POLKA [32] as a visualization front-end. 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 most notable addition is the support of concurrent, overlapping animation and more modular constructs for creating visual representations. The visualization support provided by POLKA is more complete than that supplied by STAGEHAND. However, the two systems described above provide no mechanisms for abstracting over low-level interactions. Such abstraction must be implemented as part of the visualization mechanism in POLKA. Furthermore, PVaniM utilizes a code annotation approach, the tradeoffs of which we have already discussed; whereas the approach described in [37] modifies system level routines to record events. In the latter case, there are no convenient mechanisms for filtering events, thus every potentially interesting event must be recorded. STAGEHAND represents a significant step in the direction of integrating language support for specifying visualization with the run-time mechanisms necessary to implement this support. In particular, STAGEHAND provides modular visualization constructs which respect object integrity and allow the user to specify the granularity of events. Furthermore, by utilizing a reflective architecture, STAGEHAND constructs are transparently integrated with applications, obviating the need for code annotation.

The concurrent visualization environments we have described above all employ a rather traditional view of process-based concurrent computation. The PAVANE [26] system represents a coherent approach to visualizing concurrent program execution based on a tuple-space environment similar to Linda [11]. Moreover, PAVANE has been designed explicitly to aid programmers in reasoning about program execution. In PAVANE, a configuration of tuple-space represents the current state of an algorithm in 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 STAGEHAND and PAVANE 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. STAGEHAND specifications, on the other hand, specify 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. STAGEHAND specifications, on the other hand, allow event patterns which depend on mutable 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 [19]. In a related fashion, predicate transition nets have been used to visualize actor computation [20]. 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. STAGEHAND provides a foundation upon which explicit views of concurrent computation may be developed; STAGEHAND specifications may be used to create both event diagram and predicate transition net representations.

Chapter 2

An Event-Based Model for Program Visualization

We formulate visualization of actor systems in terms of *actor events*. We are primarily interested in interaction patterns. Thus, we define actor events in terms of those transitions observable externally to each actor. Specifically, an actor event may be either a message send, a method dispatch, a method completion, or dynamic creation of new actors[1]. We disallow events based on actor internals in order to preserve object integrity. This design tradeoff is discussed in some detail in Chapter 5.

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.

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

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 2.1 illustrates the relationship among each of these components.

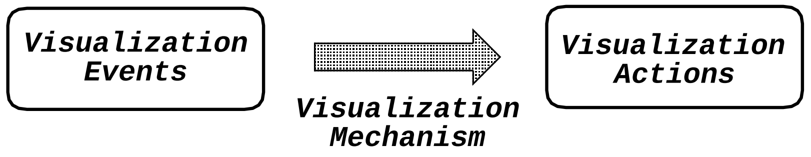

Figure 2.1: **Program Visualization.** Visualization events are mapped to visualization actions by the visualization mechanism.

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 Chapter 1. In Section 2.1 we consider actor events in more detail. In particular, we discuss the features of the visualization mechanism necessary to preserve consistency in the resulting visualization. A model based solely on local actor events is satisfactory for visualization but is too low-level for practical use. In Section 2.2, we introduce *visualization groups*, an abstraction mechanism for encapsulating interaction patterns in the spirit of abstract data types. Finally, in Section 2.3 we develop the architecture of the visualization mechanism required to implement our model.

2.1 Events

Actor events represent the most fine-grain elements which may trigger visualization. As a result, some care must be taken in their definition. In particular, it is expected that a large volume of actor events will be generated by an executing system. Thus, actor events must be defined so that their detection incurs little overhead without sacrificing expressive ability. Moreover, we require that actor events capture the low-level interactions used to express coordination. Formally, actor events are defined as follows:

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

- A *message send.*
- A *method dispatch.*
- A *method completion.*
- The *creation* of a new actor.

A message send corresponds to the invocation of the **send** actor primitive. A method dispatch corresponds to the transition of an actor from the *ready* state to the *processing* state. Similarly, a method completion corresponds to the transition of an actor from the *processing* state to the *ready* state. Finally, a creation event corresponds to the invocation of the **create** actor primitive. Each actor event corresponds to an externally observable interaction between the actor and the underlying run-time environment. For example, an actor invoking **send** must call the interprocess communication library supported by the system. Hence, each actor event may be detected on a local basis while preserving object integrity. Moreover, actor events correspond to the relevant local activities associated with coordination among components: in two-phase commit, for example, coordination was expressed using *vote-reply-decision* message patterns.

Detecting actor events on a local basis eliminates the necessity of querying global state. Moreover, as we demonstrate in Chapter 4, actor events may be detected transparently. In particular, we detect actor events by distributing the visualization mechanism so that each actor is monitored by an independent *observer*. Observers are objects which filter actor interactions and trigger visualization when specific actor events are detected.

Visualization triggered by actor events serves to visually identify changes in local state in response to interactions with other actors. However, because actors are distributed entities, visualization is triggered in an asynchronous fashion. From the perspective of reasoning about programs, this is an undesirable feature since it is not clear how the visualization characterizes the underlying execution. The causal relationships between interacting components in a distributed system are a critical feature for reasoning about distributed interactions [28, 7]. In particular, causal relationships indicate a chain of interactions which corresponds to the progress of a distributed algorithm. To capture this feature in the resulting visualization, we require a visualization mechanism which ensures that visualization actions *characterize* the causality of the underlying execution. This requirement may be stated in terms of the following restriction on visualization actions:

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

Note that under the casual connection restriction, the resulting visualization always corresponds to a *consistent cut* [23] of the triggering events in the underlying execution.

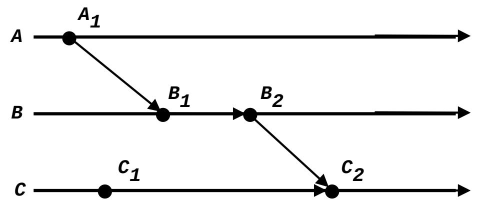

Figure 2.2: **Event Diagram.** Three actors A, B and C are shown together with their visualization events.

Figure 2.2 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 a model of visualization for actor-based systems:

Actor Event Model:

- Actor events are detected locally at each component.
- Invoked visualization actions are executed according to the *causal connection restriction.*

In the actor event model, there is a one-to-one correspondence between actor events and visualization. However, as discussed above, actor events represent the most fine-grain element of visualization. Specifically, we anticipate a large volume of actor events. Unfortunately, the actor event model provides no mechanisms for abstracting over these fine-grain elements. In particular, we are forced to view every actor event. In the next section, we consider constructs for defining abstractions over patterns of actor events.

2.2 Visualization Groups

The actor event model provides a foundation for generating visualization and ensuring consistency but lacks constructs for abstracting over low-level events. In order to capture patterns of events, we organize actors into *visualization groups* which specify visualization events in terms of patterns of actor events. Moreover, visualization groups associate state with 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 defined for the group. In particular, predicates over group state may be used to guard visualization events.

Figure 2.3 illustrates the functionality of visualization groups. Recall that observers are entities associated locally with each actor. Conceptually, actors in a visualization group are monitored by observers and a *coordinator*, which manages state for the group. Observers detect actor events and report them to the coordinator. The coordinator collects local actor events

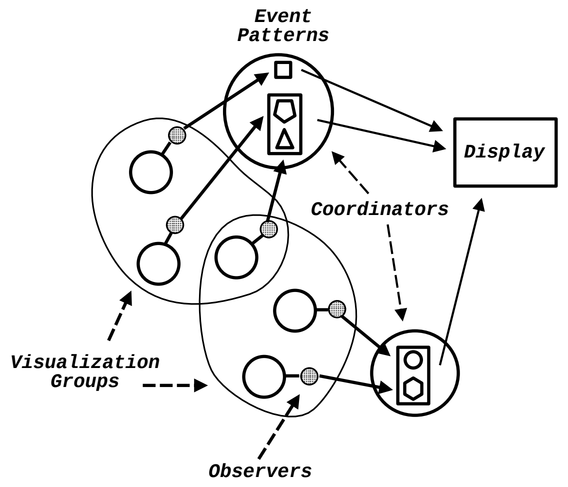

Figure 2.3: **Visualization Groups.** Visualization groups specify visualization events over possibly overlapping groups of actors.

and determines if the current set of events matches a visualization event pattern. When a pattern is matched the corresponding visualization action is invoked. Note that shared state for a visualization group is maintained within the group's coordinator.

The motivation for organizing actors into visualization groups is in the spirit of encapsulating data within an abstract data type. Specifically, we would like to encapsulate interaction patterns within a single abstraction and consider the behavior of the visualized actors strictly in terms of those patterns. In essence, a visualization group filters actor events considering only those relevant to some appropriate behavioral metaphor. Moreover, we wish to allow events which depend on visualization group state as well as patterns of actor events.

Combining the notions of state dependent events and abstraction over sets of actor events, we define visualization events inductively over actor events by adding guards or conjoining visualization events. Specifically, a visualization event for a visualization group is either an actor event on a member of the group or consists of:

Guarded Events: Visualization events guarded by a predicate.

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

For example, for two actors participating 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 addition to visualization events, visualization groups give rise to several *meta-events*. For example, an actor may move from one visualization group to another. Strictly speaking, these transitions are not visualization events but may affect the visual metaphor being presented. Thus, in order to provide flexibility, the following events may also be used to trigger visualization:

Group Event: A group event corresponds to one of the following:

- A *group enter* event.
- A *group exit* event.

A group enter event corresponds to the addition of an actor to a visualization group. A group exit event, on the other hand, corresponds to the removal of an actor from a visualization group. Actors are added and removed from visualization groups dynamically at run-time in one of two ways: explicitly using the language constructs **joinGroup** and **leaveGroup**; or, implicitly as a result of the visualization action associated with a *create* actor event. We discuss these two mechanisms for manipulating group membership in more detail in Chapter 3. Note that actors may only modify their own visualization group membership or the group membership of actors they create at the *time* of creation.

As an example of how visualization groups can be used to specify visualization, consider the two-phase commit example from above where the coordinator for the commit protocol is also a participant in a primary backup protocol. The following example illustrates how we might define visualization groups for the participants of these two protocols.

Example: Two-Phase Commit and Primary Backup. In order to visualize these two protocols we create two visualization groups, a PrimaryBackup group and a TwoPhaseCommit group (see Figure 2.4). For the PrimaryBackup group we will only be interested in **update** messages which are periodically sent by the coordinator to the backup. For visualization purposes, we will denote the coordinator with a circle when both the coordinator and its backup

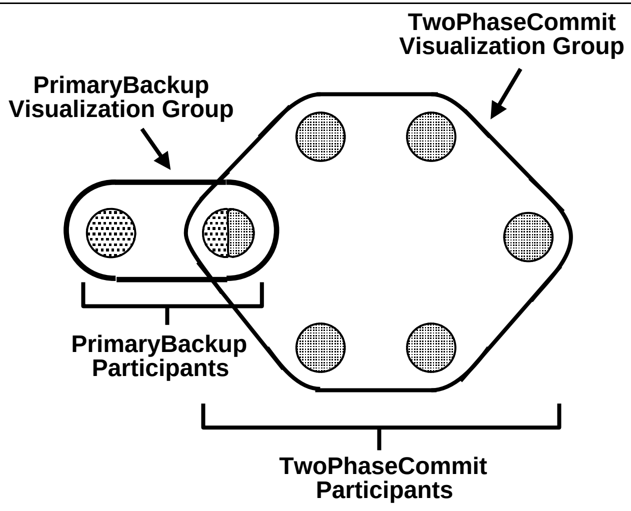
TwoPhaseCommit Visualization Group
PrimaryBackup Visualization Group
PrimaryBackup Participants
TwoPhaseCommit Participants

Figure 2.4: **Group Organization.** Organization of primary backup and two-phase commit participants.

are consistent, otherwise we will denote the coordinator with a square. The backup actor will not have a visual representation.

For the TwoPhaseCommit group will be interested in each of the message rounds of the protocol. Each participant in the protocol will have an explicit visual representation (a square for this example). We will visualize the message rounds as follows. When the coordinator broadcasts the **vote** message we will generate a "funnel" originating from the coordinator. When a participant replies with a **commit** or **abort** and the coordinator receives the reply we will draw a line from the participant to the coordinator. Finally, when the coordinator broadcasts the decision in the final stage of the protocol we will draw another funnel. The shading of the funnel will indicate whether the decision was to **commit** or **abort**. Figure 2.5 shows several frames from the resulting visualization. Notice that the visualization captures such dynamics as the fact that some participants may reply before all participants have received the vote request, but the coordinator may only reply after receiving all votes.

We will delay a detailed discussion of how visualization events are specified for this example until Chapter 3. However, we can provide the following intuitive description. The visualization event for processing **update** messages consists of a send event followed by a method dispatch

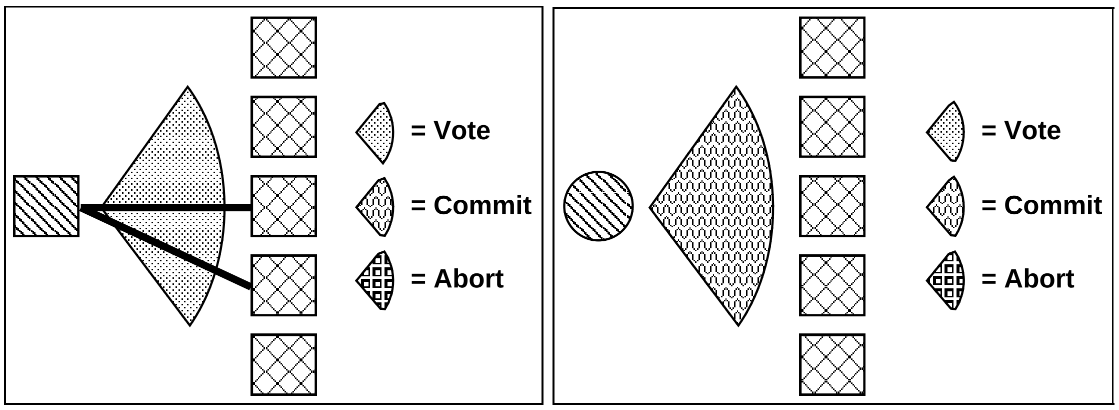

Figure 2.5: **Visualization.** On the left, the coordinator has broadcast a **vote** message, two participants have replied, and the backup is not consistent with the coordinator. On the right, the coordinate has broadcast the **commit** decision, and the backup and coordinator are consistent.

event guarded by the state variable consistent defined in the PrimaryBackup visualization group. When the coordinator sends the **update** message (detectable via a send event) we set consistent to **true**. For any message received by the coordinator we set consistent to **false** and change the coordinator's visual representation to be a square. When the backup processes an **update** message (detectable via a method dispatch event) we check the consistent flag. If consistent is **true** we change the coordinator's visual representation to be a circle, otherwise we do nothing. The visualization events for the TwoPhaseCommit group are specified in a similar fashion. The visualization event for the broadcast of the **vote** and decision messages consists of the set of all message sends from the coordinator to each participant. Replies to a vote request are just the send/method dispatch pairs originating at participants. □

As the example above demonstrates, visualization groups provide the appropriate abstraction mechanism for filtering actor events to isolate interesting behavior. However, visualization groups require a two-layered approach in order to implement visualization: actor events are first detected locally at each actor, then patterns are detected at the group coordinator. Introducing a new layer between the observed actors and the resulting visualization implies new consistency requirements for the visualization mechanism. In the final section we develop the architecture of the visualization mechanism necessary for implementing this multi-layered approach.

2.3 Visualization Mechanism

Visualization groups are constructs which allow for *abstract events*. That is, events which are defined as patterns of actor events. As a result, we need to refine the notion of causal relation between events. For actor events, causal relationships were well defined because actor events are indivisible relative to one another. That is, actor events are *atomic*. Abstract events, on the other hand, may consist of multiple actor events and may share multiple causal relationships with other abstract events. In particular, it is not clear what the "right" causal relationship is between abstract events. The purpose of this section is to develop the visualization mechanism for the event-based model, and, in doing so, define the causal relationship between abstract events. Specifically, we define how events are detected and how they are used to trigger visualization.

Figure 2.6 illustrates the architecture of the event-based model. The *visualization monitor* represents the modeling and rendering environment and generates the user display. In this figure, we have identified observers, coordinators, and the display as separate objects although they need not be implemented in this manner. The causal connection restriction guarantees that observers deliver events in causal order to all coordinators. Coordinators, in turn, collect events and determine if any visualization event for the visualization group they manage is satisfied. Satisfied visualization events cause the invocation of their corresponding visualization action.

Actor events invoke visualization actions exactly in the manner defined above. Abstract events, on the other hand, may require multiple actor events to be satisfied (*i.e.* a *set* of actor events) and may be guarded by a predicate over group state (*i.e.* a *guarded* event). As a result, actor events delivered to a coordinator must be held by that coordinator until they may be used to satisfy an abstract event or it is determined that they may never be used. Moreover,

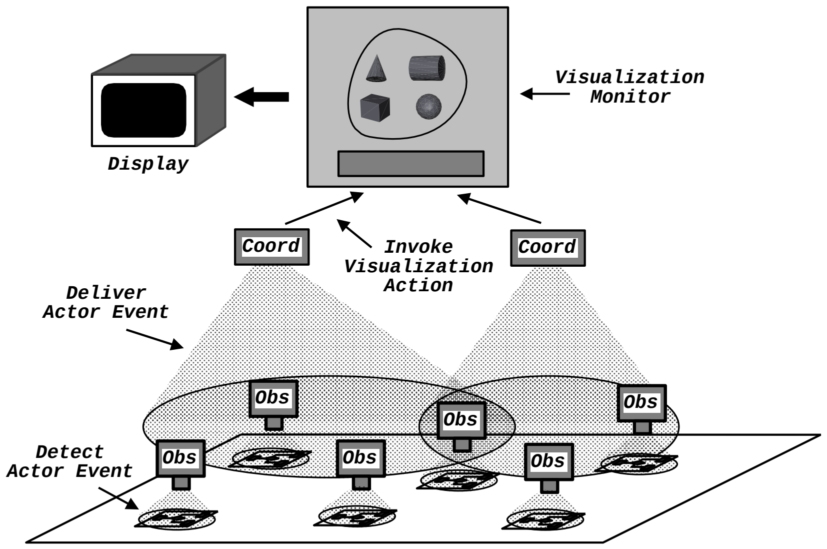

Figure 2.6: **Model Architecture.** Observers causally deliver actor events to coordinators. Coordinators determine if a visualization event is satisfied and invoke appropriate visualization actions.

if an actor event may be used to satisfy multiple abstract events, some resolution mechanism must exist to determine which abstract event is satisfied.

Preserving the causal order of the underlying execution is critical for presenting visualization which may be used to reason about distributed algorithms. Moreover, having coordinators maintain lists of actor events indefinitely is undesirable both from an implementation perspective as well as in terms of presenting understandable visualization. Thus, given a set of actor events which may be used to trigger a visualization event we require that the visualization mechanism implement the following policies:

Use Earliest Policy: An actor event may only be used to satisfy a visualization event if no other causally preceding actor event within the same group may be used. If more than one causally unrelated actor event may be used, one is chosen non-deterministically.

Use Once Policy: An actor event may only be used to satisfy one visualization event within each visualization group.

The use earliest policy guarantees that actor events won't be used in a single visualization group in a manner contradictory to the underlying causal order of the execution. That is, visualization *within a group* is always generated according to a consistent cut of actor events. The use once policy guarantees that as soon as an actor event is used to satisfy a visualization event it will be discarded. By enforcing these policies it is unambiguous at the time of invocation which actor events were used to satisfy a visualization event. However, neither policy places any restriction on the relation between visualization events in separate visualization groups.

As a final consideration, note that visualization actions may modify state. Thus we require that the visualization mechanism execute visualization actions atomically so that each action has a consistent view of visualization group state. We may summarize the event-based model as follows:

Event-Based Model:

- Actors are organized into visualization groups.
- For each visualization group, visualization events are defined as patterns of actor events over the set of member actors.
- Actor events are detected locally and delivered in causal order to coordinators.
- Coordinators invoke visualization actions by matching visualization events using the "use earliest" and "use once" policies.
- Visualization actions are executed atomically.

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.

Chapter 3

Specifying Visualization

In this chapter we discuss linguistic support for specifying visualization of actor-based systems. Our goal is to develop unambiguous specification mechanisms which do not require access to actor internals, and which support visualization activated by the event structure we described in the previous chapter. There are two aspects to specifying visualization: defining visualization events, and defining visualization actions which are triggered by these events. In this chapter we only consider language constructs for defining visualization events. Developing linguistic support for specifying visualization actions depends to a large degree on the modeling and rendering library in use as well as the demands of the user. However, the degree of support provided and the syntactic mechanisms required for specifying visualization actions is not germane to the development of event detection language support. Moreover, it is our intention to provide specification mechanisms which are not constrained by the computer graphics support available in the system. Thus, we will leave issues related to specifying visualization actions until Chapter 4. Note that none of the constructs defined here rely on graphics library specific support.

For the purposes of this chapter, we will assume a simple C-like syntax for the base actor language. Note that none of the constructs we define are inextricably tied to this base language. All that we require is a syntax for identifying an actor's behavior, and the methods and message contents used in message passing. We will identify an actor's behavior with a simple text string.

For example, an actor implementing the Fibonacci function would be labeled Fibonacci. When we refer to the *type* of an actor, we are referring to its behavior. Methods within an actor will be defined much in the same fashion as a C function prototype without a return value. For example, the function sort which takes as arguments an array and its boundaries would be identified as:

sort(**Array** A, **Integer** lower, **Integer** upper)

We may also refer to the same method as just sort when we are not interested in the arguments. When we refer to the *address* of an actor we are referring to the unique identifier used to send messages to the actor.

3.1 Visualizers

The notion of a visualization group is captured by the Visualizer language construct. Visualizers are defined in a fashion similar to classes in the object-oriented sense. Specifically, visualizers maintain a list of members (instances if you like), local state, and a set of rule blocks which define visualization events. Members of a visualizer are added or removed dynamically using syntax described below and in Chapter 4. Our language defines rule blocks for three classes of events:

Membership Change. Each visualizer maintains a list of member components. Visualization may be triggered when an actor joins the group or when a current member leaves.

Actions. An action is defined to be either a message send or a method dispatch and is meant to indicate a direct interaction between two actors. Visualization may be triggered according to patterns of actions which may depend on visualizer state and the contents of messages.

Dynamic Behavior. Dynamic behavior corresponds to the creation of new actors (*i.e.* invoking **create**) or method completion. Visualization may be triggered when members of a visualizer create new actors or complete the processing of a method (*i.e.* coarse grain state change).

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. The rules in a visualizer are evaluated independently for each member when specific behavior is detected. Figure 3.1 provides an abstract syntax for visualizers.

```
visualizer ::= visualizer name {
                          local_variables
                          initialization
                          membership_rules
                          action_rules
                          dynamic_behavior_rules
                          }
```

Figure 3.1: **Visualizers.** Rule blocks are used to organize visualization actions according to the type of visualization events which trigger them.

The specification of a visualizer may include any, all, or none of the rule blocks identified in Figure 3.1. When a member of a visualizer exhibits one of the behaviors specified above, that object is referred to as the *target*. In general, rules defined in a visualizer match based on conditions defined on the target. For example, membership rules match based on the type of the target and whether the target is joining or leaving the visualizer. Some rules may have multiple targets. For example, actions may specify communication patterns involving multiple objects. Where it is unambiguous, the address of the target is always bound to the identifier **self**. Otherwise, specific identifiers must be introduced to identify separate components within a rule. Note that because actor addresses are unique, **self** may be used to identify individual actors within a rule.

The specification of a visualizer allows for the definition of both local variables (encapsulated within the visualizer) as well as an initialization block which is executed when the visualizer is created. In general, the syntax of these two sections will depend on the base actor language in use. Thus we will not provide a syntax for specifying local variables or their initialization until Chapter 4. For this chapter, assume that local variables may simply be referenced by name when necessary.

```
membership_rules ::= begin enter                  behavior_rule ::= on behavior do
                         {behavior_rule}*                               vis_action
                     end enter                                      end
                     begin exit
                         {behavior_rule}*
                     end exit
```

Figure 3.2: **Membership Change Syntax.** Abstract syntax for visualization events which match membership change.

3.1.1 Membership Rules

Membership rules are used to invoke visualization actions in response to membership changes in a visualizer. In particular, two types of rules may be defined. An *enter* rule specifies a visualization action triggered when an actor becomes a member of the visualizer. Similarly, an *exit* rule is triggered when an actor leaves the visualizer. Both rules are matched based on the type of the actor. Moreover, the identifier **self** is bound to the address of the actor within each visualization action which is triggered. Figure 3.2 gives an abstract syntax for membership rules. The meaning of the syntax is described below:

- $\boxed{\textit{vis\_action}}$ specifies a visualization action.
- $\boxed{\text{on } \textit{behavior} \textbf{ do} \ldots}$ defines a behavior rule. If the target actor has type matching *behavior* then the associated visualization action is invoked.

3.1.2 Action Rules

Action rules are used to invoke visualization actions in response to the detection of message patterns. An action rule 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. Because rules are evaluated in the order they are specified, detecting a disjunction of patterns is implicit. Figure 3.3 gives an abstract syntax for defining actions.

$$
\begin{array}{lll}
\textit{action\_rules} & ::= & \textbf{begin action} \\
 & & \quad \left\{ \begin{array}{l} \textit{pattern} \ \{ \ \textbf{where} \ \textit{expr} \ \} \ \textbf{do} \\ \quad \textit{vis\_action} \\ \textbf{end} \end{array} \right\}^{*} \\
 & & \textbf{end action} \\
 & & \\
\textit{pattern} & ::= & \textit{msg\_spec} \\
 & | & \textit{msg\_spec}_1 \ \textbf{and ... and} \ \textit{msg\_spec}_n \\
 & & \\
\textit{msg\_spec} & ::= & \{\textbf{local}\} \ \textit{id}_t \leftarrow \textit{id} : \textit{method} \\
 & | & \{\textbf{local}\} \ \textit{id}_t \rightarrow \textit{id} : \textit{method} \\
 & & \\
\textit{method} & ::= & \textit{method\_name} \\
 & | & \textit{method\_name}(\textit{arg\_list})
\end{array}
$$

Figure 3.3: **Actions Syntax.** Abstract syntax for actions.

Actions are structured according to individual component-to-component interactions. Actors communicate by invoking named methods on other actors. *Method* identifies a method of an actor and specifies the *method_name* and, optionally, an argument list specified by *args*. Recalling our example from above, sort is a valid *method* and so is sort(**Array** A, **Integer** lower, **Integer** upper). The purpose of this syntax is to be able to define actions based on the messages passed among actors as well as the contents of those messages. Note that by convention, **self** is *never* defined within an action rule. The remainder of the syntax is defined as follows:

- $\boxed{\{\textbf{local}\}\ id_t \leftarrow id{:}method}$ and $\boxed{\{\textbf{local}\}\ id_t \rightarrow id{:}method}$ specify a *msg_spec* which matches a single message interaction. A $\leftarrow$ specifies a method dispatch. A $\rightarrow$ specifies a message send. The keyword **local**, if present, requires that both participants in the interaction be members of the visualizer within which this *msg_spec* appears. *Method* specifies the message passed between the two participants as described above. A *msg_spec* is said to *match* an interaction if:

  1. The optional **local** keyword is satisfied.
  2. The interaction is a send or method dispatch as appropriate.
  3. The passed message satisfies the specification of *method*.

If the interaction is matched, then id_t is bound to the address of the member of the visualizer sending or dispatching the method in the interaction, and *id* is bound to the address of the other participant in the interaction. If *method* includes an argument list, then each argument is bound appropriately according to the contents of the message. Note that messages in an actor's mail queue will not match a method dispatch *msg_spec* until the actor is about to begin processing the method.

- $\boxed{\textit{msg\_spec}_1 \textbf{ and} \ldots \textbf{and } \textit{msg\_spec}_n}$ defines a pattern which matches if and only if $\textit{msg\_spec}_1$ through $\textit{msg\_spec}_n$ may be satisfied.

- $\boxed{\textit{pattern } \{ \textbf{ where } \textit{expr } \}}$ specifies the complete visualization event for an action rule. The **where** *expr* syntax indicates an optional guard specified as a predicate over the state of the visualizer and the bindings of any argument list in a *msg_spec* in *pattern*. If no guard is present, then the event is matched when *pattern* is satisfied. If a guard is specified, then the event is matched when *pattern* is satisfied and the guard evaluates to true.

As mentioned above, patterns are created by assembling *msg_specs* using conjunction and guards. Actions are unique in that they may require multiple interactions before triggering a visualization action. As a result, each time a member of a visualizer is involved in a message interaction, the status of each action rule is updated in order. If an action is completely matched, the appropriate visualization action is invoked. Recall from Chapter 2 that visualizers are required to implement both the *use earliest* and *use once* policies. That is, a message may be used to satisfy a *msg_spec* only if no causally preceding message may be used in its place, and a message may be used to match only one action rule. Until an interaction is used in rule matching it is available to any action rule.

3.1.3 Dynamic Behavior Rules

Dynamic behavior rules are invoked when member components complete the processing of a message (called a *become* rule for historical reasons), or when a member of a visualizer instantiates a new actor using **create** (called a *create* rule). Create rules, in addition to triggering a visualization action, may specify the visualizer membership of the new actor. Dynamic behavior rules define visualization in response to the dynamically changing computational environment. In particular, create rules allow visualizers to update visual abstractions in response to new system components, while become rules can be used to indicate changes in these abstractions in response to coarse grain state change. Figure 3.4 gives an abstract syntax for dynamic behavior rules.

```
dyn_beh_rules ::= begin create
                      {create_rule}*
                  end create
                  begin become
                      {become_rule}*
                  end become

create_rule   ::= on behavior from id {join vis_1,...,vis_n} do
                      vis_action
                  end

become_rule   ::= after {local} id : method do
                      vis_action
                  end
```

Figure 3.4: **Dynamic Behavior Syntax.** Abstract syntax for dynamic behavior rules.

Create rules have two targets, namely, the newly created actor and the member of the visualizer which created it. Become rules have a single target: the member of the visualizer which just completed processing a method. For create rules, **self** is bound to to the address of the newly created actor. For become rules, **self** is bound to the address of the member of the visualizer which just completed the method. In addition, for create rules, an identifier may be bound to the address of the creating actor and the initial visualizer membership of the created actor may be specified. Similarly, for become rules, an identifier may be bound to the address of the actor which invoked the method. In addition, the arguments of the completed method are accessible within the visualization action triggered by a become rule. The meaning of the syntax is described below:

- $\boxed{\textbf{on } \textit{behavior} \textbf{ from } \textit{id} \ \{\textbf{join } vis_1, ..., vis_n\} \textbf{ do } ...}$ defines a create rule. A create rule matches if the newly created actor has type matching *behavior*. When the rule matches, **self** is bound as described above and *id* is bound to the address of the member of the visualizer which created the new actor. The optional expression **join** $vis_1, ..., vis_n$ specifies a list of visualizers to which the new actor should be added after it has been created. Note that adding actors in this fashion causes an enter event to be triggered in each visualizer.

- $\boxed{\textbf{after } \{\textbf{local}\} \ \textit{id} : \textit{method} \textbf{ do}}$ defines a become rule. When the rule matches, **self** is bound as described above and *id* is bound to the actor which invoked the method which has just completed. The optional keyword **local**, if specified, requires that the invoking actor must be a member of the visualizer in which the rule is defined. *Method* refers to a valid method specification as defined for *action* rules above.

3.2 Example

The linguistic constructs we have described above are suitable for detecting the types of events we defined in Chapter 2. In particular, visualizers completely encompass the notion of visualization groups. Our approach has two key strengths:

- Our linguistic constructs respect object integrity and do not provide any functionality which breaks object encapsulation

- Action rules are particularly powerful for visualizing coordination as they may trigger visualization actions based on both message patterns and their contents.

Moreover, any identifiers bound in membership, dynamic behavior or action rules remain bound within the scope of the triggered visualization action. Thus, our linguistic constructs effectively parameterize the visualization which they trigger.

To illustrate how visualizers are specified syntactically, consider the visualization of primary backup and two-phase commit from Chapter 2. We defined two visualization groups, a PrimaryBackup and a TwoPhaseCommit visualization group. Naturally, we will define two corresponding visualizers which capture the appropriate interaction patterns. Figure 3.5 gives the complete code for the PrimaryBackup and TwoPhaseCommit visualizers.

The PrimaryBackup visualizer captures interactions between the coordinator and the backup. We use the consistent flag to determine when the two are consistent. In particular, if the backup dispatches an Update message before the coordinator receives any further messages, the two are consistent. We capture this behavior by setting consistent to **true** when the coordinator sends an update, and testing the value of consistent when the backup receives the update. In a similar fashion, the TwoPhaseCommit visualizer captures interactions between the coordinator and each participant. We capture the broadcast of a vote request or decision by creating a conjunctive event consisting of all the individual message sends to each participant. Note that by the *use once* policy, each message used to satisfy the conjunctive event must be unique. Thus the event is only satisfied when all messages in the round have been sent. We capture the individual replies from each participant using a simple message specification event.

3.3 Discussion

The specification of the PrimaryBackup and TwoPhaseCommit visualizers reiterates how visualization may be specified within the event-based model without requiring access to component internals. In particular, note that none of the event specifications required access to actor internals. All that we require is a specification of the actor's interface. However, the above example also illustrates some of the limitations of the specification language we have provided. For example, it would be more convenient to express the broadcast of a Vote message using a "wild-card" operator which captures multiple messages invoking the same method. In our specification language, such patterns may only be expressed by explicitly listing each of the messages involved.

In general, the language we have provided should be extended with more powerful combinators for expressing interaction patterns. Nonetheless, our goal in this chapter was to provide a basic specification language which could be used to develop visualization for the event-based model. The language constructs we have illustrated above completely satisfy this requirement. Furthermore, within each event, the participants and the interactions involved are readily discernible from the description of the event. This allows an unambiguous description of the relationship between execution behavior and the resulting visualization. In the next chapter, we describe the implementation of STAGEHAND, a prototype visualization environment which implements the event-based model and completely supports the language described above.

```
visualizer PrimaryBackup {
  begin var
    bool consistent;
  end var

  begin enter
    on Coordinator do
      Draw circle for coordinator
    end

    on Backup do
      // Assume consistency initially
      consistent = true;
    end
  end enter

  begin action
    // Coord sends update
    Coord → Back : Update do
      consistent = true;
    end

    // Coord receives message
    Coord ← Part : VoteCommit do
      consistent = false;
      Draw square for coordinator
    end

    Coord ← Part : VoteAbort do
      consistent = false;
      Draw square for coordinator
    end

    // Backup is consistent
    Back ← Coord : Update where consistent do
      Draw circle for coordinator
    end

    // Backup is not consistent
    Back ← Coord : Update where !consistent do
      // Do nothing, discards event
    end
  end action
}
```

```
visualizer TwoPhaseCommit {
  begin enter
    on Participant do
      Draw square for participant
    end
  end enter

  begin action
    // Coord sends vote
    Coord → P1 : Vote and
    Coord → P2 : Vote and
    Coord → P3 : Vote and
    Coord → P4 : Vote and
    Coord → P5 : Vote do
      Draw vote "funnel"
    end

    // Coord sends commit decision
    Coord → P1 : Commit and
    Coord → P2 : Commit and
    Coord → P3 : Commit and
    Coord → P4 : Commit and
    Coord → P5 : Commit do
      Draw commit "funnel"
    end

    // Coord sends abort decision
    Coord → P1 : Abort and
    Coord → P2 : Abort and
    Coord → P3 : Abort and
    Coord → P4 : Abort and
    Coord → P5 : Abort do
      Draw abort "funnel"
    end

    // Coord receives a reply
    Coord ← Part : VoteCommit do
      Draw connection to participant
    end

    Coord ← Part : VoteAbort do
      Draw connection to participant
    end
  end action
}
```

Figure 3.5: **PrimaryBackup and TwoPhaseCommit Visualizers.** Visualization events are defined for Backup, Coordinator, and Participant actors. Visualization actions are represented by italicized pseudo-code.

Chapter 4

Implementation

STAGEHAND is a visualization environment designed to extend actor-based systems in order to support online event-based program visualization. In particular, STAGEHAND implements the event-based model described in Chapter 2 and supports the specification of visualization using the language constructs defined in Chapter 3. In order to test our mechanisms, we chose to extend BROADWAY [34], a prototype environment for building actor-based systems. The accessibility and strict object-oriented design of BROADWAY make it an ideal platform for testing our ideas.

In this chapter, we provide a detailed discussion of the implementation mechanisms used in building STAGEHAND. Section 4.1 provides an overview of the structure of BROADWAY. A more detailed description of the design methodology which inspired BROADWAY is available in [34]. In Section 4.2 we describe the overall architecture of STAGEHAND visualizers. In particular, we provide a general description of how visualizers are implemented and how BROADWAY actors are added to visualizers. The implementation of visualizers can be roughly divided into two components: actor event detection and delivery, and rule matching and execution. In Section 4.3 we describe the STAGEHAND mechanisms added to BROADWAY in order to implement event detection and causal delivery. In Section 4.4 we describe implementation mechanisms for rule matching and visualization action execution. In Appendix C, we provide several fully coded examples and screen shots using STAGEHAND.

4.1 Broadway

BROADWAY is a prototype programming and run-time support environment implemented in C++ for developing actor-based applications. In particular, BROADWAY provides support for distributed actor programs including asynchronous communication, dynamic actor creation, and scheduling of actors. Basic actor functionality is augmented with support for migration, exception handling [4], synchronization constraints [13], and modular specification of interaction policies [34]. The platform currently runs on Ultrix for DEC MIPS workstations, on Solaris for SUN Sparcstations, and IRIX 5 for SGI workstations.

Broadway supports basic actor functionality using a multi-thread scheduler, distributed name service, and platform independent communication service. These facilities are implemented using class hierarchies that simplify adding new features to the system. For example, the visualization mechanisms we describe below were implemented as a subclass of the standard actor behavior.

Each actor is implemented as a C++ object: the state and methods of the actor are the state and methods of the C++ object. Each actor also maintains a mail queue to buffer incoming messages. When an actor is ready to process its next message, the scheduler invokes the correct method in the C++ object as a new thread. With one notable exception — replies from RPC invocations — only one method may be active for a single actor: there is no internal concurrency.

In addition to run-time functionality, BROADWAY includes a library of system actors. These actors include an i/o and file system interface, a failure detector, and a migration controller. Application actors interact with the system actors using standard asynchronous message passing.

4.1.1 Reflection in Broadway

In addition to supporting basic actor application development, BROADWAY is designed to support a limited form of reflection using compiled objects. Reflection has two manifestations in

BROADWAY: a *dynamic* form of reflection which allows compositional modifications to the actor communication mechanism, and a *static* form of reflection which allows transparent interaction with superclass structures which all actors contain as part of their instantiation. The former mechanism is *dynamic* because it may be customized on a per-actor basis at run-time. The latter mechanism is *static* because the superclass structure of all actors is fixed at run-time. In the next section, we discuss how STAGEHAND utilizes the static reflective capabilities supplied by BROADWAY for the purpose of local actor event detection[1].

4.2 Visualizer Architecture

To provide for a straightforward implementation, we organize visualizers within a single BROADWAY actor called a VisManager. Upon creation, the VisManager calls the initialization block of each visualizer. The VisManager is responsible for managing the membership of all visualizers, collecting actor events and distributing them to the appropriate visualizer, ensuring that visualizers see events in the correct causal order, and providing an interface to the modeling and rendering library. In general, STAGEHAND visualization proceeds in three interacting stages: first, actor events are detected locally at each application actor and delivered in causal order to the VisManager; second, the VisManager distributes each actor event to the appropriate visualizer; and third, each visualizer is given an opportunity to determine if any rules have matched and to trigger appropriate visualization actions. Figure 4.1 illustrates the relationship between application actors and the VisManager.

In BROADWAY, each application actor is developed as a specialization of the class AClass which encapsulates an actor superstructure. The actor superstructure consists of a "fat" class hierarchy using multiple inheritance to allow the customization of several features of the actor run-time system. Figure 4.2 illustrates this superstructure. AClass provides an interface for standard actor functionality (send, create, etc.) to all subclasses (*i.e.* application actor classes).

[1]Interested readers should refer to [34] for applications of the dynamic form of reflection for the modular specification of interaction policies.

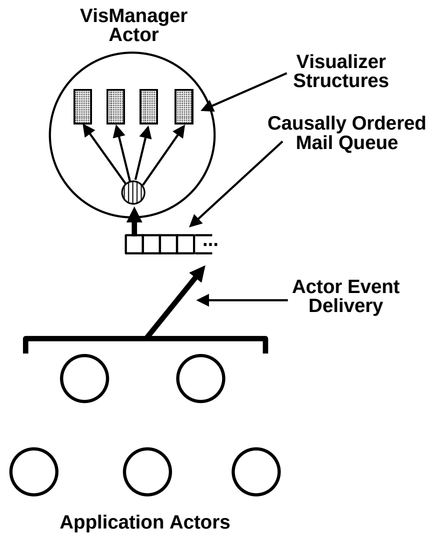

Figure 4.1: **Visualization in STAGEHAND.** Application actors detect actor events and report them to the VisManager. The VisManager distributes actor events to the appropriate visualizers.

AClass methods are implemented by accessing methods either in one of the immediate superclasses or in Actor. Thus, the standard actor functionality may be augmented by inserting the appropriate class between AClass and Actor and modifying the appropriate methods. Moreover, interactions may be explicitly directed at superstructure classes without the knowledge of the application actor. This extendibility provides a form of static reflection: messages may be sent to superstructure classes in order to customize the behavior of an actor; however, classes in the superstructure are fixed at compile time.

We take advantage of BROADWAY's static reflective support to link application actors to the VisManager. In particular, we introduce a special event detection class into the AClass superstructure for the purpose of event detection and delivery. This class, together with the VisManager, combine to implement a causal delivery protocol which ensures that the VisManager only receives events in causal order. Within the VisManager, each visualizer is implemented as an object which maintains a list of events and evaluation mechanisms for determining when an event is satisfied. When the VisManager receives an actor event, it distributes the event to the appropriate visualizer object and checks all objects to determine if any visualization events

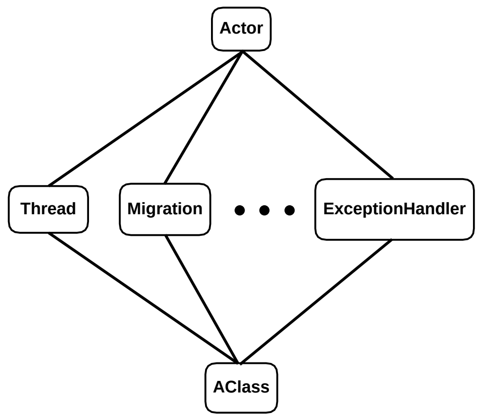

Figure 4.2: **Actor Superstructure.** Application actors inherit from AClass. Classes may be inserted between Actor and AClass and accessed from AClass methods to customize specific actor features.

have been matched. A matched visualization event triggers the appropriate visualization action and the used actor events are discarded. We discuss the implementation of event detection, delivery, and rule evaluation in some detail in the remaining sections.

4.3 Event Detection and Delivery

We implement local actor event detection by adding the VisEvent class to the AClass superstructure. The appropriate methods in AClass are then modified to allow VisEvent access to all interactions which constitute an actor event. The VisEvent instantiation at each actor tracks the set of actor events which should be detected locally, called a *local detect list*, and updates causal information in response to interactions with other actors. This causal information is used to implement causal delivery at the VisManager. In particular, the VisEvent class provides the following functionality:

Visualizer Membership. The VisEvent class responds to requests from the VisManager to add or remove the local actor from a visualizer. In the case of joining a visualizer, events are added to the local detect list. In the case of leaving a visualizer, events are removed from the local detect list.

Event Detection. The VisEvent class determines when the local actor has participated in an interaction which matches an event in the local detect list, and sends information about the event to the VisManager.

Causal Information. The VisEvent class updates local causal information in response to interactions with other actors and events caused by the local actor.

Figure 4.3 illustrates the functionality of the VisEvent class.

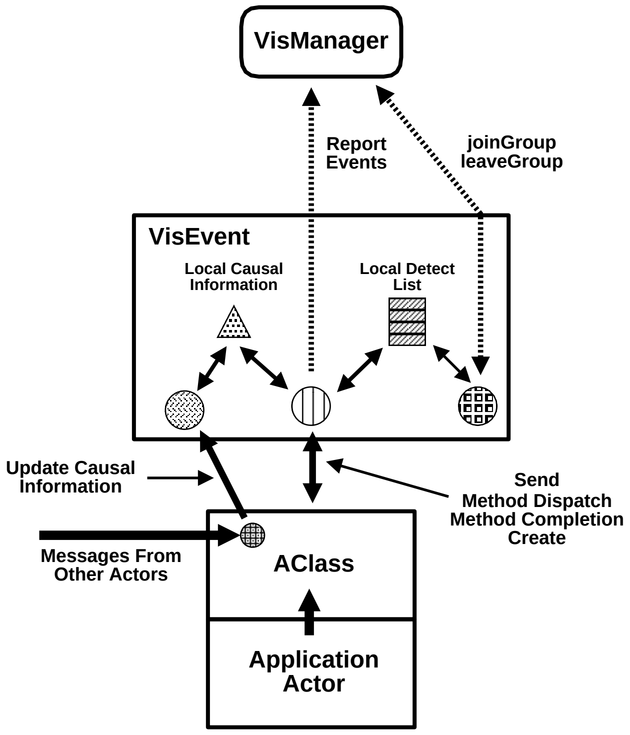

Figure 4.3: VisEvent **Functionality.** The VisEvent class manages the visualizer membership of a local actor, detects and reports local events, and maintains local causal information.

The local detect list maintains the following information for each event to be detected:

1. The type of event: send, method dispatch, create, or method completion.

2. A unique ID for the event.

3. The address of the actor to report the event to (*i.e.* the VisManager).

4. Event specific information:

   - Send, method dispatch, and method completion events require the name of the method to be detected.
   - Create events require the type of the actor created.

Each visualizer maintains a list of actor events, called a *member detect list*, which specifies the exact set of events which its members should detect locally. Actors are added to a visualizer by sending the member detect list to the VisEvent component of the actor which, in turn, adds the list of events to its local detect list. Actors are removed from a visualizer by simply reversing the process.

Requests to add or remove actors are processed by the VisManager. Actors are added to visualizers in one of three ways: first, an actor may add itself to a visualizer; second, an actor may add another actor to a visualizer if it knows its address[2](*i.e.* an *acquaintance*); and third, an actor may be implicitly added to a visualizer as a result of a **create** visualization event. In order to maintain consistency, it is necessary to synchronize the installation or removal of an actor from a visualizer. This is done via remote procedure calls (RPC) among the involved actors. Specifically, when an actor wishes to change its own membership it initiates an RPC with the VisManager which adds the actor to the appropriate visualizer and returns the *member detect list* to the calling actor. The caller adds the list to its own *local detect list* before exiting the RPC. If an actor wishes to modify the membership of an acquaintance, it again initiates an RPC with the VisManager. The VisManager, in turn, initiates an RPC with the actor whose visualizer membership will be modified and passes the appropriate *member detect list*. The modified actor performs the appropriate modifications to its *local detect list* and the RPCs unwind to the original calling actor. In the case of an implicit add due to a **create**, the same scheme is used where the creator calls the VisManager which in turn calls the created actor.

[2]Technically, this feature is not supported by the language constructs defined in Chapter 3. However, to help bootstrap visualization we have added this feature to the implementation of STAGEHAND.

As discussed above, local events are detected by modifying the send, method dispatch, method completion and create methods in AClass. Specifically, when an actor invokes one of these methods, control is eventually passed to the local VisEvent instantiation. VisEvent checks to see if the interaction should be reported and returns immediately if no action is required. If the local event should be reported then local causal information is updated, and appropriate information about the event is packaged together with causal information and sent to the VisManager. The local detect list is stored as a hash table so that it can quickly be determined whether or not an event should be reported. In the case of a create event, some overhead may result due to the necessity of adding the created actor to a visualizer. This is unfortunate but necessary in order to preserve the causal relationship between the create event and any subsequent interactions involving the created actor.

A rather straightforward vector clock protocol, described in Appendix A, is used to store local causal information and implement causal delivery at the VisManager. Specifically, the VisEvent portion of each visualized actor maintains a local vector clock which is updated in response to local events and interactions with other actors. The VisEvent portion of non-visualized actors also maintains a vector clock but only updates the clock in response to information obtained from messages received from other actors. This preserves causal relationships between visualized actors who interact through a non-visualized actor.

4.4 Rule Matching and Execution

In STAGEHAND, visualizers are managed by the VisManager using an interface for relaying events received from individual actors. Each visualizer specification is compiled into a class with *event handler* methods corresponding to each of the visualization events defined for the visualizer. In addition, a visualizer class instance maintains structures which track a list of *candidate* actor events which may be used to satisfy some visualization event defined by the visualizer. Figure 4.4 presents C++ pseudo-code illustrating the basic structure of a visualizer class.

```
class compiled_visualizer : public visualizer {
public:
  ActorEventList Candidates;

   Variables from var block copied here

  /* Constructor calls init code block */
  compiled_visualizer();

  /* These methods invoked by VisManager
   * when a new actor event is received.
   */
  HandleSend();
  HandleDispatch();
  HandleCompletion();
  HandleCreate();

  /* Visualization action methods for each visualization event.
   * Code block for each action is invoked by these methods.
   */
  EventEnter_1(); ... EventEnter_n();
  EventExit_1(); ... EventExit_n();
  EventAction_1(); ... EventAction_n();
  EventCreate_1(); ... EventCreate_n();
  EventBecome_1(); ... EventBecome_n();
}
```

Figure 4.4: **Compiled Visualizer Class.** Each visualizer specification is compiled into a class with methods for handling each of the various actor events. A list of *candidate* actor events which may be used to satisfy a visualization event is maintained by the class.

When the VisManager is created (at system startup), an instance of each visualizer specific class is created and initialized by calling the initialization block defined in the corresponding visualizer specification. Each event handler is responsible for determining if an incoming actor event may satisfy a particular visualization event. When a visualization event is matched, appropriate bindings are generated according to the specification of the related visualizer rule, and the corresponding visualization action is activated. In STAGEHAND, visualization actions are specified as C++ code blocks. Visual representations are created and manipulated using a simple graphics library described in Appendix B. Note that the event-based model defined in

Chapter 2 requires that visualization actions be atomic. In STAGEHAND, this is accomplished by serializing the execution of visualization actions. That is, only one visualization action may execute at a time. We mandate (but do not enforce) a programming discipline in which visualization action blocks terminate in a "timely" fashion. Since a visualization action may activate animation primitives, the VisManager periodically handles rendering updates allowing the initiating visualization action to terminate.

The evaluation of *enter*, *exit*, *create*, and *become* rules is straightforward: it can be immediately determined whether or not the actor event may be used to satisfy a visualization event. For these rules, if an event is satisfied then the corresponding visualization action is immediately invoked. Determining whether or not an action rule may fire is more involved. Action rules are compiled into a list of basic message specifications, all of which must be satisfied before the event may be satisfied. If a guard is present in the action rule, then the guard expression is attached to this list. When a send or method dispatch actor event is received, the following steps are taken for each action rule:

1. Test if it may be possible for the new message to satisfy a basic message specification. If this is not possible (for example, there may be a local constraint on the basic message specification which the message does not satisfy), then discard the event.

2. Add the message to the list of candidates.

3. While there is a permutation of causally ordered candidates which has not been tried, do:

    (a) If the current permutation satisfies all the basic message specifications and the optional rule guard is satisfied, then

    - Fire the associated visualization action.
    - Remove the used messages from any candidate list in which they are stored.

Note that the algorithm above adheres to the use earliest and use once policies as required by the model.

4.5 Discussion

The static reflective capabilities supported by BROADWAY provide a clean mechanism for adding functionality to actors. In particular, note that the additions required by STAGEHAND are completely transparent to application code. All the required changes are implemented in the AClass superstructure. Existing BROADWAY applications may be visualized simply by recompiling them with the new AClass.

In terms of overhead, the event detection mechanisms require space and time overhead which is proportional to the number of actor events which match a particular interaction. Only a small constant amount of overhead is imposed on actors which are not being visualized. In general, it is expected that each actor will have a relatively small local detect list, usually on the order of five events or less. As a result, event detection will not impose an unreasonable amount of overhead on visualized actors.

The vector clock protocol which implements causal delivery requires space and time overhead which is proportional to the number of vector clock entries. We implement a vector clock protocol, described in more detail in Appendix A, in which the number of vector clock entries in a single actor will never be more than the number of visualized actors which are causally related. That is, the size of the vector clock will only increase when an actor learns of other visualized actors by receiving messages. In particular, isolated actors will always have a vector clock with only a single entry.

Several design tradeoffs are possible in the implementation of the rule matching and visualization action execution mechanism. In particular, we could have chosen to implement visualizers as separate actors rather than encapsulating them within a single VisManager actor. The central approach has the advantage of simplifying causal delivery but results in a rather bulky actor and serializes the execution of visualization actions. The distributed approach neatly partitions visualizer code but increases overhead due to the need to synchronize the delivery of causally related events to separate visualizers. Thus, we have utilized the central approach in order to provide a straightforward implementation. Future work should consider the benefits of a distributed implementation such as concurrently executing visualization actions when possible, and so on.

Chapter 5

Conclusion

5.1 Summary

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, this thesis advances the concept of developing program visualizations of concurrent algorithm execution which can be used to reason about causal behavior and coordination. We have developed a model which emphasizes distributed detection of visualization events and captures coordination activity with minimal overhead. The event-based 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.

Given that we wish to visualize a distributed computation based on local events, we have developed linguistic support for encapsulating visualization paradigms for groups of actors. In particular, a *visualizer* is a language construct, much in the spirit of abstract data types, which defines visual abstractions and rules which modify these abstractions in response to interaction patterns. Visualizers are *specification transparent* in that they need only refer to the interfaces of member actors and hence may be specified separately from algorithm code. Rules may be specified within visualizers which respond to patterns of actor events. Moreover, visualizers maintain state so that temporal patterns of interaction may be used to satisfy rules.

In order to demonstrate the transparent realization of the event-based model and our language constructs, we have implemented STAGEHAND, a prototype environment for visualizing distributed computations expressed in the BROADWAY actor-based environment. As per the event-based model, actor events are detected locally and reported in causal order for the purpose of visualization. STAGEHAND provides for *execution transparency* by way of the static reflective capabilities of BROADWAY and the low-overhead filtering of interface invocations by the base actor. We utilize the same mechanism to guarantee the causal connection restriction when triggering visualization actions.

5.2 Future Work

5.2.1 Linguistic Support for Event Patterns

We provided only rudimentary linguistic support for expressing event patterns in Chapter 3. Our goal has been to express unambiguous patterns from which the participants and interactions involved are readily discernible. However, a more comprehensive environment should include more powerful event specification mechanisms. In particular, it should be possible to be able to express general patterns over all actor events. Moreover, we might wish to allow more flexible constraint mechanisms for specifying basic interactions. We continue to research more flexible mechanisms for specifying interaction patterns among groups of actors. In addition to the visualization context, we are also considering interaction patterns as applied to synchronization and real-time constraints, fault-tolerance requirements, and load balancing and migration.

5.2.2 Internal Transition Events

The definition of actor events we have provided is particularly powerful in that it captures coordination related behavior with breaking object encapsulation. However, as visualization becomes more integrated with the debugging process, events which are based on internal actor transitions and state may become desirable. Allowing such an event may not require a complete violation of object integrity, however. In particular, if the underlying actor system supports reflective descriptions of actor state manipulation then it would be possible to customize reflectively for the purpose of visualization. Although we have demonstrated that internal transitions bias views of coordination behavior, we are considering adding reflective descriptions of state for the purpose of supporting these internal events. Such events may prove particularly useful in a distributed debugging context.

5.2.3 Comprehensive Modeling and Playback

We have concentrated on specifying visualization event specification and detection mechanisms rather than specifying explicit graphics modeling support. However, a comprehensive modeling and rendering environment is critical for allowing users to create the most appropriate visual abstractions. In particular, features such as a visual control panel for monitoring the visualization mechanism as well as more powerful rendering and modeling constructs are required. Moreover, it should be possible to record visualization for later replay and analysis. A complete environment should include multiple displays showing both a literal view of the involved components and their interactions as well as a view showing the visualization generated.

We are currently investigating an extension of these ideas in which visualization would be coupled with a distributed debugging environment. In such an environment, event detection would be less passive and would allow execution to be retraced in order to discover features such as race conditions, synchronization errors, and so on.

Appendix A

A Simple Vector Clock Protocol

In order to guarantee that visualization preserves the causal order of the underlying execution, we deliver actor events to the VisManager in causal order. This requires a mechanism for tagging messages with causal information so that the mail queue can be ordered appropriately. A straightforward mechanism for implementing causal delivery is to use vector clocks [23]. Vector clocks are a well known mechanism for implementing causal delivery and several implementation techniques have been suggested in the literature [28, 33, 7, 24]. The algorithm we describe here is similar to that described in [23].

In a vector clock protocol, each participant maintains a local Lamport clock [17] as well as a Lamport clock for each participant it has received a message from. One participant is designated the *observer* and will use this *vector* of clock information from other participants to causally receive messages. Initially, every participant (including the observer) has their vector clock initialized with all zeros. When a message is sent to another participant, the message is tagged with the vector clock being maintained locally. When a message is received from a participant, the vector tag is inspected and all local clocks except for the recipient's clock are updated with the latest values.

When an interesting event occurs at a participant, the participant's local clock is incremented and an event message is sent to the observer tagged with the local vector clock. Using the observer's own vector clock together with the vector clock tag of incoming messages, the following delivery rule is used:

Delivery Rule: Let V_{obs} be the observer's vector clock. Let $V_{m,i}$ be the vector clock tag of message m sent from participant i. Then, message m should be delivered as soon as both of the following conditions are satisfied:

$$V_{obs}(i) = V_{m,i}(i) - 1$$

$$V_{obs}(j) \geq V_{m,i}(j)\ \forall j \neq i$$

The first condition guarantees that all causally preceding messages from the sender have already been delivered. Similarly, the second condition guarantees that all messages which causally precede the message m have already been delivered.

The protocol given above is fairly straightforward to implement. However, two issues must be resolved: first, how should participants be added to the protocol; and second, how should vector clock information be stored. The first issue is fairly important when considering event-based visualization since it is likely that participants will added to the protocol quite often. The second issue is important when considering space overhead. In particular, recall that we desire visualization mechanisms which do not require oppressive amounts of overhead. The remainder of this appendix discusses how vector clocks are implemented in STAGEHAND.

Considering the first issue, observe that the knowledge that a new participant has joined the protocol is only required by participants which interact with the new member. However, this information will be naturally transmitted the first time the new member sends a message. In particular, the new member will attach a vector clock tag which will always include its local clock. Thus, new members will be discovered when other participants receive messages with vector clock entries that are not present in the locally maintained list. The observer will learn of new members in a similar fashion. In short, no extra overhead is required to add a participant to the protocol, the new participant is simply told to begin maintaining a vector clock.

In light of the first issue, it seems logical to only require a participant to store clock information concerning participants which it has interacted with (*i.e.* received messages from).

Thus, in STAGEHAND, a vector clock consists of a list of address-clock pairs where each entry corresponds to vector clock information obtained from a received message, plus an extra entry to maintain the participant's own local clock. All other participants will have an implicit clock value of zero which need not be stored in this list.

The implementation described above may be formalized as follows. Each participant tags all outgoing messages with the local vector clock. When a participant i with vector clock V_i receives a message m with vector clock tag V_m, the following algorithm is performed:

1. For each address-clock pair $(a, c) \in V_m$ do the following:

   (a) If $a \neq i$ and a is not an address in the local vector clock, then add (a, c) to the local vector clock.

   (b) If $a \neq i$ and $c > V_i(a)$, then set $V_i(a) = c$.

Similarly, upon receiving a message, the observer performs the following algorithm:

1. For each address-clock pair $(a, c) \in V_m$ do the following:

   (a) If a is not an address in the local vector clock, then add $(a, 0)$ to the local vector clock.

2. Add message m to the local mail queue.

3. Deliver all messages in the local mail queue which satisfy the vector clock delivery rules. When message m from participant i is delivered:

   (a) Increment the value for i in the local vector clock.

Note that in the case of the observer, when a new participant is discovered, the local clock is initialized to zero. This guarantees that the observer will not miss any events.

Appendix B

Modeling and Rendering Support

B.1 Overview

Visualization actions are invoked when a rule in a visualizer is satisfied by some interaction. The purpose of visualization actions is to update the graphical display based on the algorithm event detected. Components and their interactions are represented by graphical abstractions which may be manipulated by multiple visualizers. Thus, linguistic constructs are necessary which support both a wide range of modeling attributes as well as mechanisms for arbitrating shared manipulation of models among visualizers. In STAGEHAND, visualization actions manipulate objects in a hierarchical modeling environment [12]. Models are rendered in 3D with lighting and animation which may be specified within visualization actions.

Visualization actions are specified as blocks of C++ code which are parameterized by identifiers in the triggering event. Objects in the modeling environment are represented as instances of special modeling classes which support object-specific attributes. User-defined attributes may be assigned to modeling objects for customization. Models are organized into *scene hierarchies* composed of objects from the modeling hierarchy. Scene hierarchies specify a *view* which defines camera position and one or more lights or surfaces representing the scene. Figure B.1 illustrates the modeling hierarchy.

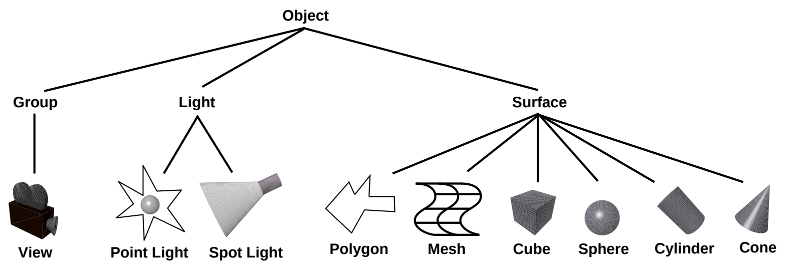

Figure B.1: **Model Hierarchy.** Hierarchy of objects for creating visualization.

Each modeling object maintains a list of attributes which determines its appearance. Some objects, such as groups and views, are not displayed and maintain parameters which determine general features of the scene. We briefly describe each of the modeling objects below:

Group. A group object maintains a list of children objects and is used to build more complex, hierarchical scenes.

View. A view object defines a camera in a scene and maintains parameters which determine the position of the viewer. A view is a subclass of group and inherits the ability to manage children. Only the children of a view are visible through its camera.

Point Light, Spot Light. These objects define light sources in a scene and maintain related parameters such as position, color, brightness, and so on.

Polygon, Mesh, Cube, Sphere, Cone, Cylinder. These objects define actual visible objects and maintain parameters such as color, position, radius, shininess, surface normals, and so on.

Each object requires specific attributes in order to define its position, appearance, and structure. View and group objects are used to build and view scenes. Thus, view and group objects define default settings for all properties which are inherited by the objects they organize. In particular, the objects organized under a group or view may be manipulated as a single unit by modifying group or view attributes. The following attributes are supported by STAGEHAND modeling objects:

Transform. This attribute defines the set of transformations which should be applied to the object. Objects may be scaled, rotated or translated. All objects use the transform attribute.

Camera. This attribute defines camera position, window size, view up and aspect. This attribute is only used by view objects.

AmbientLight. This attribute defines the ambient contribution to the shading of all objects in the current view. This attribute is only used by view objects.

SurfColor, SurfSpecColor, SurfShininess, SurfEmissiveColor, SurfAmbientColor. These attributes define surface shading properties for any viewable object. These attributes are only used by the polygon, mesh, cube, sphere, cone, and cylinder objects.

MeshData. This attribute defines surface properties for each facet of a mesh. This attribute is only used by mesh objects.

CubeVert1, CubeVert2. These attributes define the opposite corners of a cube object. This attribute is only used by cube objects.

LightStatus, LightColor, LightPosition, LightExp, LightAngle, LightDir. These attributes define properties of a light source. These attributes are only used by point and spot light objects.

SphereRadius. This attribute is used to set the radius of a sphere object. This attribute is only used by sphere objects.

CylRadius, CylHeight. These attributes define the radius and height of a cylinder object. These attributes are only used by cylinder objects.

ConeRadius, ConeHeight. These attributes define the radius and height of a cone object. These attributes are only used by cone objects.

Children. This attribute specifies a list of children objects. This attribute is only used by view and group objects.

PolyVerts. This attribute specifies the vertices of a planar polygon. This attribute is only used by polygon objects.

In addition to the attributes above, each object supports the methods setPropVal and getPropVal which allow arbitrary data to be associated with an object and referenced by a string name. This mechanism allows users to customize properties of each object.

The organization of objects into scene hierarchies allows for a natural approach to scene modeling where objects are manipulated by local modeling transformations. In addition, scene hierarchies are easy to transform into displays. In particular, a scene hierarchy is rendered by way of a *traversal* [14]. A scene traversal is accomplished by the following pseudo-code function which is called with the top object of the scene hierarchy and the empty list as initial arguments:

```
function SceneTraversal( Object top, TraversalStateList state) {
  TraversalStateList old;
  old := state;
  Update state from the attributes of top;
  if ((top.type = view) or (top.type = group)) {
    for each i in top.children
      SceneTraversal(i, state);
  } else
      Render object using properties from state;
  state := old;
}
```

Essentially, a stack of attributes is maintained using a TraversalStateList and passed recursively down the scene hierarchy. In this fashion, attributes are inherited down the scene hierarchy from parent objects.

B.2 Using Modeling Objects

Modeling objects are created by instantiating the appropriate modeling hierarchy classes. Objects are maintained as part of the state of one or more visualizers. In particular, declarations of modeling objects may be specified as part of the variable and initialization sections of a visualizer. At a minimum, at least one *view* object must always be specified in order to view modeling objects. Specific examples of the declaration and manipulation of modeling objects are provided in Appendix C.

Because visualizers do not share state, but may share members, it may be the case that a single visual representation is used to represent an actor which is a member of several visualizers. Thus, we require a mechanism for obtaining references to modeling objects maintained by other visualizers. The setPropVal and getPropVal support routines described above may be used to support this functionality. In particular, the global support function FindObjectWithProp takes as argument a list of property names and returns the list of objects in all scene hierarchies which define the given properties. Similarly, the function FindObjectWithVal takes as argument a list of property name/value pairs and returns the list of objects in all scene hierarchies in which each given property in each object has the specified value.

Primitive animation support is provided for transforming modeling objects. The applyTransform function applies a given transformation to a modeling object a specified number of times, rendering the scene after each application. Only one object at a time may be animated using applyTransform. To allow multiple object transformations, the addAnimRequest function may be used to specify several modeling objects and corresponding transformations. Animation requests added in this fashion are processed concurrently by calling the function processAnimRequests, which regenerates the scene after each transformation. The VisManager actor periodically calls processAnimRequests to guarantee that any requests added within a visualization action are eventually processed.

B.3 Implementation

The modeling support described above is not tied to any specific computer graphics library. In particular, the scene traversal algorithm is non-library specific. Only the actual rendering routines must be customized for different computer graphics libraries. In light of this observation, we have implemented modeling support using a graphics library independent front-end which may be linked to an appropriate graphics library specific back-end for handling display generation.

We have implemented each modeling primitive as a C++ class which inherits from the PropertyManager class. The PropertyManager class manages instances of the properties described above. Each primitive class defines a general set of modeling and rendering support routines customized for the primitive. Actual rendering code, however, is specified as a stub which is linked to an appropriate graphics library specific class method. Models are created and rendered using the library independent interface where control is passed to the library specific routines when rendering is required.

Using this approach, a general purpose front-end, and back-ends for the Tcl/Tk [22] and GL graphics libraries have been implemented.

Appendix C

Examples

In this chapter, we provide two fully coded examples of the use of STAGEHAND for specifying visualization. For each example, we briefly describe what is being visualized, outline the corresponding actor code, provide the full visualizer specification for visualizing the execution, and provide screenshots from the actual visualization. In Section C.1, we visualize a simple distributed implementation of the Fibonacci function. In Section C.2, we visualize the two-phase commit and primary backup protocols first introduced in Chapter 2.

C.1 A Distributed Fibonacci Application

The Fibonacci function represents a classic example of a recursive control structure. Actor-based systems yield a natural distributed implementation of such structures. Figure C.1 illustrates a pseudo-code implementation of the Fibonacci function. A single Fibonacci actor (not shown), called the server, is created to service all requests to compute the Fibonacci function. Clients request computation by sending the method FibCall to the server. In order to service the request, the server creates a FibWorker actor and passes the value requested along with the address of the requesting customer. A FibWorker will immediatedly return a result if the value requested is less than two. Otherwise, two additional FibWorkers are created to handle the recursive cases. Note that because each Fibonacci request is distributed, multiple requests may be handled concurrently. Furthermore, recursive sub-requests are handled concurrently.

In order to demonstrate how Fibonacci requests are satisfied, we will create a simple visualization of the creation of FibWorkers and the delegation of tasks via message passing. In particular, we will represent a Fibonacci request as a tree of FibWorkers and use animation to show messages passed among the tree's components. We will use color changes to indicate when FibWorkers have completed various stages of their computation.

Figures C.2, C.4, and C.5 illustrate the complete code required to visualize the Fibonacci function. Figure C.2 outlines the Fibonacci visualizer. We require global variables to manage the view and the address of the first FibWorker created to satisfy a request. In the init section, we initialize the position of the view. Figure C.4 gives the action rule block for the Fibonacci visualizer. The rules defined in this block animate the exchange of information via message passing between FibWorkers. In particular, the **send** portion of an exchange causes a visual representation of the message to be moved from the sender halfway to the receiver. The corresponding **receive** portion of an exchange causes the visual representation to be moved to the receiver. To illustrate how state is affected by information exchange, the visual representation of a FibWorker is colored red when it is created, yellow when it has received one reply from a recursive sub-request, and green when it has received both replies and has sent its result to its client. Finally, Figure C.5 gives the create rule block for the Fibonacci visualizer. The create rule simply creates the visual representation for new FibWorkers. We assign properties to the visual representation in order to maintain information about the corresponding FibWorker. In particular, we store the value of the request that this FibWorker is processing. New visual representations are positioned relative to the creating FibWorker so that a tree-like structure of visual representations is maintained.

Figure C.3 shows several frames from the resulting visualization.

C.2 Two-Phase Commit with Primary Backup

In Chapter 3 we introduced an example of a visualization of a primary backup protocol overlapping with a two-phase commit protocol. In this section we give the complete code for generating this visualization. Figure C.6 gives a pseudo-code specification of the relevant actors. In reality, each actor would have several other application specific methods. However, we only illustrate those methods pertinent to the visualization. Moreover, we assume that the coordinator periodically sends state updates to its backup.

The Backup actor contains the single method update, which is called by the Coordinator in order to pass state updates. The Participant actor defines the methods vote_request, commit, and abort. The vote_request method is called by the Coordinator at the start of a two-phase commit. The commit and abort methods are used by the Coordinator to notify the Participant as to what action should be taken. Finally, the Coordinator actor defines the methods vote_commit and vote_abort which are used by the Participants to register votes. As in Chapter 3 we assume one Coordinator and five Participants.

We have described the specification of the PrimaryBackup and TwoPhaseCommit visualizers in some detail in Chapter 3 thus we will not reiterate that discussion here. Figures C.8 and C.9 give the complete code for both visualizers. Figure C.7 shows several frames from the resulting visualization.

```
Actor FibWorker {
  Int return_val;
  Bool finished;
  Actor client;

  Method FibCall(Int request, Actor customer) {
    if ( request ≤ 1) then
      send FibReply(1) to customer;
    else {
      left = create FibWorker;
      right = create FibWorker;
      finished = false;
      client = customer;
      send FibCall(request - 1, self) to left;
      send FibCall(request - 2, self) to right;
    }
  }

  Method FibReply(Int result) {
    if finished then
      send FibReply(result + return_val) to client;
    else {
      finished = true;
      return_val = result;
    }
  }
}
```

Figure C.1: **A Distributed Implemention of Fibonacci.** The Fibonacci actor receives requests to compute the Fibonacci function. FibWorker actors are created dynamically to service the request and return the result to the client.

```
// Visualizer for the fibonacci example
visualizer fib_vis {
  begin var
    // Global Variables
    viewVO vRoot;
    Address root;
    int start=false;
  end var

  begin init
    // Set camera position
    vRoot.Camera_setVRP(0,0,0);
  end init

  Action rules: Figure C.4

  Create rules: Figure C.5
}
```

Figure C.2: **Outline of Fibonacci Visualizer.** The structure of the Fibonacci visualizer.

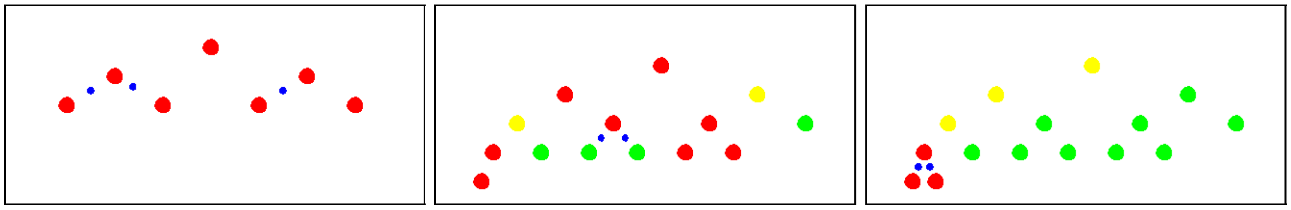

Figure C.3: **Fibonacci Screenshots.** Blue circles indicate messages being passed between FibWorkers. The first frame shows the status of a request shortly after it has begun: three messages are in transit and no processing has been completed. The second frame shows partial completion of the request: four FibWorkers have completed processing. The last frame indicates that all but the far left FibWorkers have completed processing.

```
begin action
  // Send fib_call
  local a -> b : fib_call do
    // Find objects rep. sender and receiver
    send=FindObjectWithVal("address", a);
    rcv=FindObjectWithVal("address", b);
    pos_s=send->getPropVal("pos");
    pos_r=rcv->getPropVal("pos");

    // Animate message send
    mObj=new sphereVO;
    vRoot.addChild(mObj);
    mObj->SurfColor_color(blue);
    mObj->Transform_translate(pos_s);
    rcv->setPropVal("call_msg", mObj);
    vRoot.animTranslate(mObj, pos_s,
      0.5*pos_r);
  end

  // Receive fib_call
  local b <- a : fib_call do
    // Find objects rep. sender and receiver
    send=FindObjectWithVal("address", a);
    rcv=FindObjectWithVal("address", b);
    pos_s=send->getPropVal("pos");
    pos_r=rcv->getPropVal("pos");

    // Finish animation of message delivery
    mObj=rcv->getPropVal("call_msg");
    vRoot.animTranslate(mObj, 0.5*pos_r,
      pos_r);
    vRoot.removeChild(mObj);
    vRoot.renderObject();
  end

  s <- d : fib_call(Integer req) where !start do
    // Event will be triggered on the initial request
    root=s; start=true;
    new_sphere=new sphereVO;
    new_sphere->SurfColor_color(red);
    new_sphere->SphereRadius_setValue(10.0);

    // Set sphere properties
    new_sphere->setPropVal("request", req,
      "address", s, "child", 0,
      "pos", (0,0,0));

    vRoot.addChild(new_sphere);
    vRoot.renderObject();
  end

  // Send fib_reply
  local a -> b : fib_reply do
    // Find objects rep. sender and receiver
    send=FindObjectWithVal("address", a);
    rcv=FindObjectWithVal("address", b);
    pos_s=send->getPropVal("pos");
    pos_r=rcv->getPropVal("pos");

    // Change color of sender
    send->SurfColor_color(green);

    // Animate message send
    mObj=new sphereVO;
    vRoot.addChild(mObj);
    mObj->SurfColor_color(blue);
    mObj->Transform_translate(pos_s);
    send->setPropVal("reply_msg", mObj);
    vRoot.animTranslate(mObj, pos_s,
      0.5*pos_r);
  end

  // Receive fib_reply
  local b <- a : fib_reply do
    // Find objects rep. sender and receiver
    send=FindObjectWithVal("address", a);
    rcv=FindObjectWithVal("address", b);
    pos_s=send->getPropVal("pos");
    pos_r=rcv->getPropVal("pos");

    // Finish animation of message delivery
    mObj=send->getPropVal("reply_msg");
    vRoot.animTranslate(mObj, 0.5*pos_r,
      pos_r);
    rcv->SurfColor_color(yellow);
    vRoot.removeChild(mObj);
    vRoot.renderObject();
  end

  s -> d : fib_reply where s == root do
    // Result sent to client
    rootObj=FindObjectWithVal("address", s);
    rootObj->SurfColor_color(green);
    vRoot.renderObject();
  end
end action
```

Figure C.4: Action Block for Fibonacci Visualizer. Action rules for the Fibonacci visualizer.

```
begin create
  // Create rep. for new actor
  on FibWorker from creator join fib_vis do
    // Find the object representing our parent
    parent=FindObjectWithVal("address", creator);

    // Set up a new sphere for this actor
    request=parent->getPropVal("request") - 1;
    sub_space=powf(2, request - 1) * hor_spacing;
    pos=parent->getPropVal("pos");
    child=parent->getPropVal("child");

    us=new sphereVO;
    us->SurfColor_color(red);
    us->SphereRadius_setValue(10.0);
    us->setPropVal("request", request,
      "address", self, "child", 0);

    pos.y -= vert_spacing;
    if (child == 0)
      pos.x -= 0.5 * sub_space;
    else
      pos.x += 0.5 * sub_space;
    us->setPropVal("pos", pos);
    parent->setPropVal("child", child + 1);

    // Transform, add ourselves, and view
    us->Transform_translate(pos);
    vRoot.addChild(us);
    vRoot.renderObject();
  end
end create
```

Figure C.5: **Create Block for Fibonacci Visualizer.** A create rule for the Fibonacci visualizer.

```
Actor Backup {
  Method update(State coord) {
    Record state of coordinator
  }
}

Actor Coordinator {
  Method vote_commit(Address part) {
    Record vote from participant
  }

  Method vote_abort(Address part) {
    Record vote from participant
  }
}

Actor Participant {
  Method vote_request(Address coord) {
    Send our vote to the coordinator
  }

  Method commit {
    Commit the transaction
  }

  Method abort {
    Abort the transaction
  }
}
```

Figure C.6: Primary Backup and Two-Phase Commit Actors. The Backup actor periodically receives state updates from the Coordinator through the update method. The Coordinator sends vote requests to participants and receives replies through the vote_commit and vote_abort methods. Similarly, Participants return their vote when they receive a vote_request and receive the Coordinator's decision through the commit and abort methods.

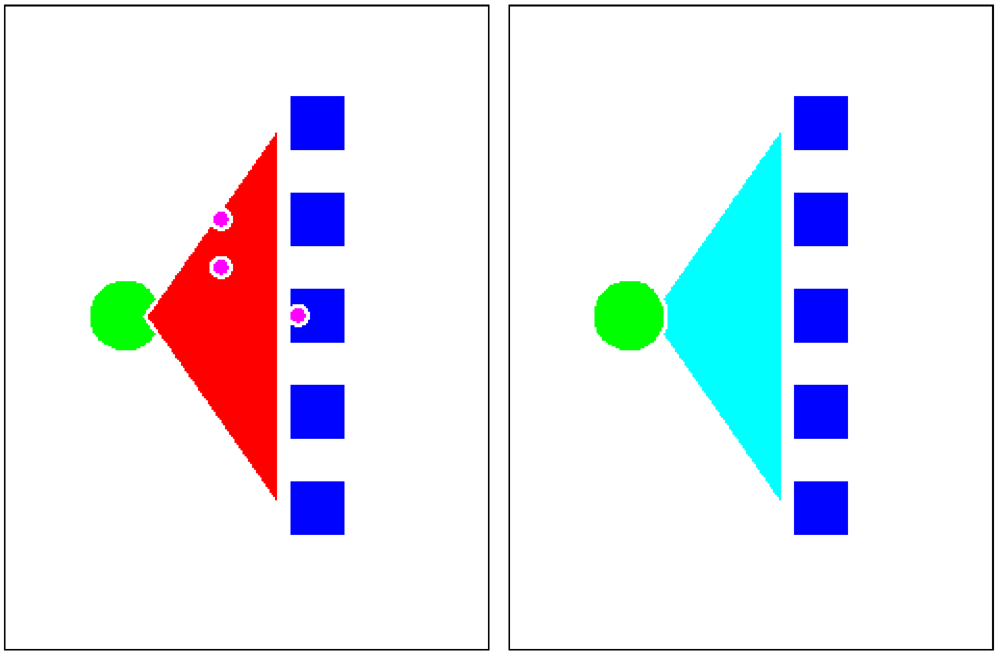

Figure C.7: Two-Phase Commit with Primary Backup Screenshots. On the left, the vote_request message has been broadcast and three participants are in the process of replying. The coordinator and backup are consistent. On the right, all votes have been received and the coordinator has broadcast the abort message.

```
visualizer PrimaryBackup {
  begin var
    // Global variables
    viewVO vRoot;
    VO cur_rep;
    sphereVO coord1;
    cubeVO coord2;

    // Consistent flag
    int consistent;
  end var

  begin init
    // Set camera position
    vRoot.Camera_setVRP(0,0,0);
    vRoot.setPropVal("theView", 0);
  end init

  begin enter
    // Create coordinator representation
    on coordinator do
      coord1.SurfColor_color(green);
      coord2.SurfColor_color(green);
      coord1.SphereRadius_setValue(20.0);
      coord2.CubeVert1_position(-15.0, -15.0, 0.0);
      coord2.CubeVert2_position(15.0, 15.0, 0.0);
      cur_rep = coord1;

      vRoot.addChild(cur_rep);
      vRoot.renderObject();
    end

    // Assume consistency initially
    on backup do
      consistent = true;
    end
  end enter

  begin action
    // Coord sends update
    Coord -> Back : update do
      consistent = true;
    end

    // Coord receives message
    Coord <- P : vote_commit do
      consistent = false;
      if (cur_rep != coord2) {
        vRoot.removeChild(coord1);
        cur_rep = coord2;
        vRoot.addChild(cur_rep);
        vRoot.renderObject();
      }
    end

    Coord <- P : vote_abort do
      consistent = false;
      if (cur_rep != coord2) {
        vRoot.removeChild(coord1);
        cur_rep = coord2;
        vRoot.addChild(cur_rep);
        vRoot.renderObject();
      }
    end

    // Backup is consistent
    Back <- Coord : update where consistent do
      if (cur_rep != coord1) {
        vRoot.removeChild(coord2);
        cur_rep = coord1;
        vRoot.addChild(cur_rep);
        vRoot.renderObject();
      }
    end

    // Backup is not consistent
    Back <- Coord : update where !consistent do
      // Do nothing, discards event
    end
  end action
}
```

Figure C.8: Visualizer for PrimaryBackup. Visualizer for primary backup interactions.

```
visualizer TwoPhaseCommit {
  begin var
    VO *view;
    cubeVO parts[5];
    coneVO funnel;
    int cur_part;
  end var
  begin init
    // Find the view, init primitives
    view = FindObjectWithProp("theView");
    for (i=0; i¡5; i++) {
      parts[i].SurfColor_color(blue);
      parts[i].Transform_translate(100, (2-i)*50, 0);
    }
    funnel.ConeHeight_setValue(70.0);
    funnel.ConeRadius_setValue(100.0);
  end init
  begin enter
    // Create rep. for participant
    on participant do
      view->addChild(parts[cur_part]);
      parts[cur_part].setPropVal("address", self);
      parts[cur_part].setPropVal("y",
        (2-cur_part++)*50);
      view->renderObject();
    end
  end enter
  begin action
    // Coord sends vote
    coord -> p1 : vote_request and
    coord -> p2 : vote_request and
    coord -> p3 : vote_request and
    coord -> p4 : vote_request and
    coord -> p5 : vote_request do
      funnel.SurfColor_color(red);
      view->addChild(funnel);
      view->renderObject();
    end
    // Coord sends commit decision
    coord -> p1 : commit and
    coord -> p2 : commit and
    coord -> p3 : commit and
    coord -> p4 : commit and
    coord -> p5 : commit do
      funnel.SurfColor_color(yellow);
      view->renderObject();
    end
    // Coord sends abort decision
    coord -> p1 : abort and
    coord -> p2 : abort and
    coord -> p3 : abort and
    coord -> p4 : abort and
    coord -> p5 : abort do
      funnel.SurfColor_color(cyan);
      view->renderObject();
    end
    // Participant sends response
    part -> coord : vote_commit do
      P = FindObjectWithVal("address", part);
      y = P->getPropVal("y");
      reply = new sphereVO;
      reply->SurfColor_color(magenta);
      P->setPropVal("msg", reply);
      view->addChild(reply);
      view->animTranslate(reply,100,y,0,50,y/2,0);
    end
    part -> coord : vote_abort do
      P = FindObjectWithVal("address", part);
      y = P->getPropVal("y");
      reply = new sphereVO;
      reply->SurfColor_color(magenta);
      P->setPropVal("msg", reply);
      view->addChild(reply);
      view->animTranslate(reply,100,y,0,50,y/2,0);
    end
    // Coord receives a reply
    coord <- part : vote_commit do
      P = FindObjectWithVal("address", part);
      y = P->getPropVal("y");
      reply = P->getPropVal("msg");
      view->animTranslate(reply,50,y/2,0,0,0,0);
      view->removeChild(reply);
      view->renderObject();
    end
    coord <- part : vote_abort do
      P = FindObjectWithVal("address", part);
      y = P->getPropVal("y");
      reply = P->getPropVal("msg");
      view->animTranslate(reply,50,y/2,0,0,0,0);
      view->removeChild(reply);
      view->renderObject();
    end
  end action
}
```

Figure C.9: **Visualizer for** TwoPhaseCommit**.** Visualizer for two-phase commit interactions.

Bibliography

[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] G. Agha and D. C. Sturman. A methodology for adapting to patterns of faults. In G. M. Koob and C. G. Lau, editors, *Foundations of Dependable Computing: Models and Frameworks for Dependable Systems*, chapter 1.2. Kluwer Academic Publishers, 1994.

[5] G. A. Agha, I. Mason, S. Smith, and C. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 1996. (to be published).

[6] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[7] K. P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):37–53, December 1993.

[8] D. Bowman, A. Ferrari, M. Kelley, B. Schmidt, B. Topol, and V. Sunderam. The conch network concurrent programming system. Technical report, Emory University, Atlanta, GA, January 1994.

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

[10] 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.

[11] N. Carriero and D. Gelernter. How to write parallel programs: A guide to the perplexed. *ACM Computing Surveys*, 21(3):323–357, September 1989.

[12] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice.* Addison-Wesley, 1990.

[13] S. Frølund. Inheritance of synchronization constraints in concurrent object-oriented programming languages. In *Proceedings of ECOOP 1992.* Springer Verlag, 1992. LNCS 615.

[14] D. Hearn and M. P. Baker. *Computer Graphics.* Prentice Hall, Englewood Cliffs, NJ, second edition, 1994.

[15] M. T. Heath and J. A. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, September 1991.

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

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

[18] P. Maes. Computational reflection. Technical Report 87-2, Artificial Intelligence Laboratory, Vrije University, 1987.

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

[20] 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.

[21] 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.

[22] J. K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley Publishing Company, Inc., 1994. ISBN 0–201–63337–X.

[23] Özalp 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.

[24] M. Raynal, A. Schiper, and S. Toueg. The causal ordering abstraction and a simple way to implement it. *Information Processing Letters*, 39(6):343–350, 1991.

[25] D. A. Reed, R. A. Aydt, T. M. Madhyastha, R. J. Noe, and K. A. Shields. An overview of the pablo performance analysis environment. Technical report, University of Illinois, Urbana, IL, 1992.

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

[27] G.-C. Roman, K. C. Cox, C. D. Wilcox, and J. Y. Plun. Pavane: A system for declarative visualization of concurrent computations. *Journal of Visual Languages and Computing*, 3(2):161–193, June 1992.

[28] R. Schwarz and F. Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed Computing*, 7:149–174, 1994.

[29] B. C. Smith. Reflection and semantics in a procedural language. Technical Report 272, Massachusetts Institute of Technology. Laboratory for Computer Science, 1982.

[30] 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.

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

[32] 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.

[33] P. Stephenson. Fast ordered multicasts. Technical Report TR 91-1194, Cornell University, Ithaca, NY, February 1991. Ph.D. Thesis.

[34] D. C. Sturman. *Modular Specification of Interaction Policies in Distributed Computing.* PhD thesis, University of Illinois at Urbana-Champaign, May 1996.

[35] V. S. Sunderam. Pvm: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

[36] B. Topol, J. T. Stasko, and V. Sunderam. The dual timestamping methodology for visualizing distributed applications. Technical Report GIT-CC-95-21, College of Computing, Georgia Institute of Technology, Atlanta, GA, May 1995.

[37] B. Topol, J. T. Stasko, and V. Sunderam. Integrating visualization support into distributed computing systems. In *15th International Conference on Distributed Computing Systems*, pages 19–26, Vancouver, B.C., May 1995.

[38] S. K. Turner and W. Cai. The 'logical clocks' approach to the visualization of parallel programs. In G. Kotsis and G. Haring, editors, *Performance Measurement and Visualization of Parallel Programs*, pages 45–66. Elsevier, 1993.