Reusable Interactions for Animation

Gene S. Lee
Department of Computer Science
University of British Columbia
Vancouver, BC Canada V6T 1Z4
gslee@cs.ubc.ca | gslee@alum.mit.edu

Abstract

This paper identifies *reusable interactions* and presents them as an important construct for the development and reuse of software for computer animation. A reusable interaction manages the flow of information among software components. They are first-class entities that are easy to adapt, to organize hierarchically, and to operate dynamically. In computer animation and other time-dependent systems, communications among components evolves as (simulated) time advances. With reusable interactions, developers carefully identify and control this evolution. A novel approach for computer animation that employs reusable interactions is presented in the form of the RASP toolkit. The toolkit provides tools to manage and to organize hierarchically interactions over time. The hierarchical organization of the tools promote multiple levels of reuse. Each level introduces greater means to coordinate the interactions and to reuse them appropriately.

1 Introduction

Software reuse is a growing concern in the computer graphics community. Many graphical applications share similar approaches to describe and to visualize animated scenes. Reusing existing code and design improves the quality of new applications and reduces the effort it takes to develop them. Similar to the tools of many other disciplines, tools for graphics, such as INVENTOR [23] and GRAPHICS GEMS [6], consist of collections of components that represent basic data structures and routines. Common graphics elements include geometric models, surface shaders, and interactive techniques. Developers select useful components and then create additional code to integrate them. The code establishes relations among the components and permits the components to interact.

[0] *Presented at the 5th International Conference on Software Reuse, June 1998, Victoria, Canada*

Developmental tools for computer graphics address three main research areas: *shape modeling*, *image synthesis*, and *scene animation*. Tools for the first two areas enjoy moderate success. The tools work well because the general process of shape modeling and image synthesis is reasonably well-defined and widely accepted. As advocates of domain modeling have discovered in other disciplines [19, 5], a well-established process decomposes into a set of well-defined components. The components are well-defined for reuse because their relations and interactions with others is understood.

Tools for scene animation, however, enjoy only limited success. Most tools are not well-suited to create and to control the wide variety of relations that can exist among components. Scene animation, unlike shape modeling and image synthesis, is more than just components with fixed interactions. *It is the interaction between components over time*. When and how these interactions form is the essence of computer animation. Components represent the state of an animation while the interactions control the changing relations within the animation. As simulated time advances, many different animation processes can arise which form varying types of interactions among components. Each new interaction forms a new, short-term relation that causes components to send and to receive information. The management of component interactions in relation to time is an extremely important concept for computer animation and other time-varying systems, such as distributed simulation. Components alter state as they interact dynamically.

Currently, there are no tools for computer animation that successfully promote a general approach to controling interactions that animate time-varying properties and to reusing the code that produces the animation. Instead of moderating and reusing interactions, most tools for scene animation address a specific animation process. The tools decompose the process into components with fixed relations. This approach limits the way the tools animate and often permit the tools to animate only a subset of a component's properties. For many of these tools, software reuse is less of a concern than is the development of rapid proto-

types. Most tightly couple a component's properties to the methods that change their values. It is difficult to partition the two and reuse them separately.

It is the aim of this paper to identify *reusable interactions* and to present them as an important construct for the development of software for computer animation. Devised as first-class abstractions, software interactions advance a variety of benefits towards software modeling and reuse. They encourage hierarchical approaches to assess and to adapt software for reuse. A novel approach for computer animation is presented in the form of the RASP toolkit. The toolkit supplies tools to organize and to control interactions hierarchically. Each level of the organization introduces greater means to coordinate the interactions over time and to reuse them appropriately.

This paper is divided into eight sections. Section two defines a software interaction and identifies important variations. It analyzes current techniques for interaction, enumerates their drawbacks for animation, and describes interaction-based programming. Sections three and four discuss the RASP toolkit. They present a short overview of RASP's features and identify those that build interactions hierarchically. Section five identifies three levels of reuse as encouraged by the toolkit's design. Each level of reuse fosters alternative methods to comprehend and to adapt the temporal properties of time-varying interactions. Sections six and seven assess the toolkit's approach towards reuse and animation. They analyze its usefulness and relate it to previous works in graphics and software engineering. Section eight presents the conclusion and describes future work.

2 Definitions

2.1 Software Interaction

A *software interaction* occurs when software components communicate to exchange information. The interaction is *direct* if the components mediate their own communications. They establish when and how a communication occurs. If external sources mediate communications, the interactions is *indirect*. Of the two interaction types, the latter interaction encourages separate conception and representation of relations among components. It is easy to find and update interactions because the code and data to produce interactions is distinct from the code and data of the interacting participants. Once established, an interaction is *instantaneous* if it occurs immediately. However, if it occurs in the future, the interaction is *delayed*. Animation requires delayed interactions because not all interactions in a dynamical system occur immediately after they are created. Most interactions remain inactive until important times or states arise.

Conditioned to employ standard programming techniques, most programmers form instantaneous, direct interactions between communicating components. As noted by Sullivan [24], they create interactions by having components communicate through *implicit* or *explicit* invocations. With explicit invocations, components call each other directly. They maintain references to their peers so they may exchange information. With implicit invocations, components register callback events. They inform their peers to provide them with information when certain conditions or events arise. By themselves, both invocation techniques produce immediate results. Components immediately exchange information or register callbacks after the invocations occur. To modify either invocation technique to produce a delayed interaction, it is necessary to either delay the original invocation from occurring or alter the way the invocations operate. Either way, it is necessary to intermix additional code with the original calls.

Both implicit and explicit invocations hinder reuse because they produce static dependencies. Components store references to peers or to receivers of callback events. Altering these reference during development is manageable, but sometimes troublesome. However, run-time manipulation is almost always arduous. To design components that dynamically refer to and interface with many types of components requires significant effort. In addition, the complexity of components increases as the components interact with greater numbers of peers. This produces larger components that are harder to interpret and, eventually, to reuse.

2.2 Reusable Interaction

A *reusable interaction* is a first-class abstraction that forms a basic, indirect interaction among components. It is an identifiable element that mediates interactions between two or more communicating components. An interaction's type determines how many components it manipulates, and the manner it facilitates communication. Typically small in size, every interaction type produces a specific effect that is easy to comprehend and relatively easy to adapt. Common changes modify interactions to interconnect new components, to reconfigure existing relations, and to monitor new flows of information.

The primary benefit of a reusable interaction is its ability to mediate interactions among multiple kinds of components in multiple types of settings. Neither the components nor the setting affects the interaction's behavior. The interaction is neither a primary component of a system nor a primary client of a component. It is the glue that binds components to communicate. An interaction structures a program as a collection of components interacting under external controls.

A reusable interaction produces the greatest impact when collections of them, forming a *behavior*, are reused as a whole. Reusing an entire set of interactions, or only a subset of them, creates new sets of similarly acting behav-

iors. Rearranging the relative order of interactions, an essential process for animation, also produces new behaviors. Swapping or parallelizing two previously tandem interactions creates subtle variations of original behaviors. Sequence, an uncommon theme in software reuse, is a vital concern in animation. The order that components interact affects the state and outcome of all dynamical systems.

2.3 Time-Varying Interaction

A *time-varying interaction* augments a reusable interaction with temporal attributes. The attributes specify when, how often, and under what conditions the interaction occurs. The attributes work best with an interaction that delays its operation. The interaction establishes a relationship between components for the attributes to control. For animation and other dynamical systems, reuse of temporal attributes is equally important as the reuse of interactions. The temporal attributes of an interaction characterize the nature of a scene as much as its composition of components does. Overlaying interactions with previously defined attributes produces differing scenes with similar temporal characteristics. Likewise, modifying the attributes of existing interactions produces similar scenes with differing temporal characteristics.

2.4 Interaction-Based Programming

Interaction-based programming employs first-class interactions to organize the structure of a program into components and interactions. Components send data to and receive data from the interactions. The interactions maintain references to the components they link. Interaction-based programs are evaluated by a standard programming language compiler and by a run-time evaluator. The compiler ensures that the interaction-based program follows the proper language syntax. The evaluator analyzes and verifies the program's interactions. It examines the interactions to verify that they are formed properly and to understand when and how they operate. The analysis enables the evaluator to execute parallel interactions, to optimize run-time performance, and to flag run-time errors.

Interaction-based programming encourages separate development and representation of *algorithms for computation* and *algorithms for coordination*. Algorithms for computation decide the state of components while algorithms for coordination decide the state of communications between components. The separate representations introduce a form of modularity that does not naturally develop with standard programming practices. As programs grow large and more complex, this separation provides great benefits. It permits rapid identification and manipulation of the attributes that control when and why an interaction occurs, who an interaction manages, and in what manner the interaction operates.

In an interaction-based program, both components and interactions are potentially reusable. Each may be adapted separately to accept new parameters and to work in varying contexts. Completely different approaches may be implemented to reuse each effectively. Compositional techniques towards reuse, such as [3, 11], may be applied to components only, and hierarchical schemes, such as the one described in this paper, may be applied strictly to interactions. A robust scheme to manipulate time-varying interactions is essential for animation and critical for high-level reuse. The scheme affects how interactions form behaviors, accept temporal attributes, and prompt reuse.

3 The RASP Toolkit

3.1 Overview

RASP [15, 16] is an experimental toolkit for computer animation that promotes interaction-based programming. It consists of tools to compose, render, and animate geometric models. The toolkit promotes reuse by supporting reusable, time-varying interactions. It employs hierarchical structures to interconnect components systematically over time. From the organization of the structures, it is easy to determine what components are interacting, and when and how often they interact.

The toolkit endorses a three-phase approach to application development. In the first phase, *model specification*, developers design components for animation. In the second phase, *dynamic specification*, developers adapt and sequence reusable interactions. In the final phase, *temporal specification*, developers assign temporal attributes to the interactions of phase two. Collectively, the three phases form a script that progressively moderates sequences of interactions among many components.

To integrate existing applications, developers reuse and intermix phase elements. The primary reusable elements are the components of phase one, the interactions of phase two, and the temporal script of phase three. Elements from the last two phases form RASP's primitive hierarchy. Four levels deep, this hierarchy identifies interactions and augments them with temporal attributes. Each level orders and controls the elements of the previous level, and promotes an alternative approach to reuse. Higher levels enlist abstract ways to describe and to adapt the collective effects of multiple interactions.

3.2 Primitive Hierarchy

As shown in Figure 1, the primitive hierarchy consists of *events*, *activities*, *timingActs*, and *processions*.[1] Events manage single interactions. They form temporary bindings among communicating components. Activities organize interactions to form behaviors. They delimit an interval of time that controls how often and in what order inter-

[1] The hierarchy also contains additional primitives to manipulate processes. They have been omitted intentionally because of their irrelevance to this paper's subject.

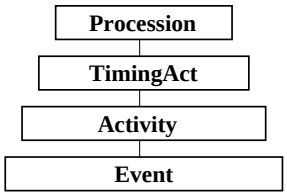

Figure 1: Primitive hierarchy

actions occur. TimingActs and processions collate behaviors to form temporal scripts. The scripts determine how behaviors relate and when behaviors occur.

4 Primitives

4.1 Events

Events are first-class abstractions that establish delayed, indirect interactions. Lowest in the primitive hierarchy, they produce reusable interactions by manipulating the *ports* of interacting components. Ports provide events with bindings to access the structures and functions of components. Ports, too, are first-class abstractions that send and receive information. The events communicate with the ports to produce interactions. Communication between events and ports occurs when the events receive notification from higher level primitives. The higher level primitives, not the events themselves, determine when an interaction occurs.

4.1.1 Programming with Ports

From an object-oriented perspective, components with ports are objects with *objects as member functions*. The member functions, being objects, possess state and accept messages. To communicate with a component, callers interact with the component's ports. Ports breach the barrier between components and their environments. When ports provide information from the component, they are *outports*; when they accept information for the component, they are *inports*. Ports return and accept values by accessing the data structures and functions of their associated components[2]. Like regular functions, they may invoke many methods, perform many computations, or access many data structures before they produce or consume information for the events.

[2] From a strict definition of an object-oriented system, ports violate the rules of encapsulation. They access information directly from the component which stores them as data members. However, ports are special objects that are elemental units of components. They, similar to standard member functions, help components encapsulate information and behavior.

Method 1 - Events Only

```
Event evt1( A→outPos(), C→inVal1() );
Event evt2( B→outPos(), C→inVal2() );
Event evt3( C→outSum(), D→inPos() );
```

Method 2 - With Virtual Ports

```
VPort *sum = *A→outPos() + *B→outPos();
Event evt( sum, D→inPosition() );
```

Figure 2: Sets D's position to the sum of A's and B's position

```
void Event::update()
{
  verify that ports are compatible
  if (outPort→getTypeId() == inPort→getTypeId())
    verify that inPort is ready to accept data
    if (inPort→getState() == TRUE)
      send data from inport to output
      inPort→doHandler( outPort→getValue() )
}
```

Figure 3: Update routine for sample event

To parameterize events, developers obtain ports from components using normal object-oriented method invocations. They invoke methods that appear by name to produce results or to return values, but in reality return references to ports. This scheme permits developers to program interactions as though they were applying normal object-oriented techniques. They need not learn a new programming paradigm or radically change their mindset. They need only realize that their invocations produce references, not immediate results or actions. For example, the code in method one of Table 2 parameterizes three events. Each event produces an interaction between an outport and an inport. When activated, the events communicate with the ports to move data from one port to the next. Collectively, the three events indicate that D's position is equivalent to the sum of A's and B's position. The component C accepts values from A and B to compute their sum. The events control the interactions which developers describe by invoking methods that return references to ports.

4.1.2 Event Types

An event's type determines what type of arguments it accepts and what type of interaction it produces. In other words, it determines how events manipulate the ports of its interacting components. The code in Figure 3 defines the "update" routine for a sample event that simply passes information from an outport to an inport, both declared pre-

Kind	Type
Simple	Event; StateEvent; CallEvent
Time	TimeEvent; StepEvent; DurationEvent; TimeStepEvent;
Logical	BoolEvent; CondEvent; BiCondEvent;
Group	ToggleEvent; ChainEvent; SwitchEvent;

Table 1: Kinds of event types

viously in the event's constructor (not shown in figure). At run-time, the event passes information between two ports after it verifies that the two communicating ports are compatible and that the inport is ready to receive data.

Table 1 identifies four kinds of events currently supplied by RASP. Simple events apply basic techniques to manage inports and outports. They either monitor them, pass data between them, or instruct them to invoke commands. Time events disseminate temporal information. They provide inports with timing values, such as current time, expected duration, and future time step. The states of components advance as they receive timing values from their inports. Logical events relate ports to logical flags and conditions. The state of the logical conditions determines how higher level primitives interpret the events. Group events contain other events. They selectively toggle, chain, or order events to bind them closely.

4.1.3 Port Types

Outports further divide into two outgoing port types, *general* and *virtual*. General ports associate with components. They represent first-class member functions. Virtual ports create associations, similar to one-way constraints, between other outports. They express symbolically complex relationships among components. They form mathematical expressions with outports as variables. The expressions provide events with formulas to derive complex interactions. The formulas evaluate when the virtual ports receive requests from events to provide values. The initial specification identifies a delayed interaction. All events, as described in section 4.1.2, freely accept virtual ports as though they were general ports. The virtual ports simply employ alternative means to provide out-going values. With virtual ports, the previous example of section 4.1.1 appears as method two of Table 2. The event `evt` employs the virtual port `sum` to compute the sum of A's and B's positions.

4.2 Activities

Activities manage collections of events. They sequence and partition events into groups to form *behaviors*. Groups determine when and how often events produce interactions.

```
      Activity* chase( Object *A, Object *B )
      {
                  produces sequences of values
(1)   Interpolator *ip = new Interpolator();
              events to parameterize interpolator
(2)   DurationEvent *dEvt = new DurationEvent( ip→inDurationVal() );
(3)   Event *bEvt = new Event( A→outPosition(), ip→inBeginVal() );
(4)   Event *fEvt = new Event( B→outPosition(), ip→inFinishVal() );
              events to compute new position
(5)   TimeEvent *tEvt = new TimeEvent( ip→inCalcValue() );
(6)   Event *eEvt = new Event( ip→outCurVal(), A→inPosition() );
                  add events to activity
(7)   Activity *act = new Activity;
(8)   act→addInitEvent( dEvt, bEvt );
(9)   act→addActEvent( fEvt, tEvt, eEvt );
(10)  return act;
      }
```

Figure 4: Chasing activity

Interactions may occur throughout an activity's lifetime or only when the activity begins or ends. Higher level primitives determine when activities start and stop. Activities continually induce events to produce interactions until they receive notice to terminate. Most often, termination occurs when specific run-time conditions arise or an activity's lifetime expires.

Simultaneously active activities produce concurrent behaviors. Interactions occur in parallel as simulation time progresses. Concurrency, an essential element of animation, is difficult to specify with conventional programming practices. One must explicitly use co-routines or intertwine operations with semaphores to adequately ensure concurrent execution. As proponents of aspect-oriented programming can attest [13], both approaches produce code segments that are unwieldy to manage and hard to reuse. Developers address issues of concurrency and design simultaneously. It is difficult to rapidly identify, extract, and reuse only design operations. Fortunately, RASP's primitive hierarchy isolates developers from this difficulty. Developers focus their efforts on constructing activities with events. The activities and higher level primitives manage concurrency.

The code in Figure 4 defines the activity `chase`. Consisting of one interpolator and five events, `chase` moves object `A` to follow object `B`. The interpolator produces a sequence of intermediate values that the activity uses to control `A`'s position. The five events produce five interactions that pass data and temporal values to and from the objects and the interpolator. The first three events (lines 2-4) parameterize the interpolator with a starting value, an ending value, and a temporal duration. The two values, each set to the positions of one object, notifies the interpolator what sequence of values to produce. The last two (lines 5-6) obtain an intermediate value from the interpolator to calculate

α *Relation* β	*Inactive* β	*Active* β
Stops	N/A	stops
Starts	starts	restarts
ReStarts	N/A	restarts
InStarts	starts	N/A
Delimits	starts & stops	starts & stops
ReDelimits	N/A	starts & stops
InDelimits	starts & stops	N/A
Meets	starts	restarts
ReMeets	N/A	restarts
InMeets	starts	N/A

Table 2: The effects of temporal relations on active and inactive timingActs. TimingActs are active if they are actively producing interactions when a relation comes into effect.

A's time-varying position. The activity groups two of the five events as *initial* events (line 8), and the remainder, as *acting* events (line 9). Initial events trigger once while acting events trigger continually. The acting event `fEvt` updates continuously the final value of the interpolator with the position of B. As B moves, the interactions between the objects and the interpolator produces a new sequence of intermediate values that accurately moves A after B.

4.3 TimingActs & Processions

TimingActs assign temporal attributes to activities. They associate activities with absolute timing values or relative temporal relations. Timing values delimit explicit intervals of time. They specify the times that activities begin and end. Temporal relations establish time-based relationships between activities. They relate the lifetimes of activities to other activities or important states. The relations, shown in Table 2, initiate and terminate activities simultaneously, create tandem sequences, and order activities conditionally. The effects on activities vary according to state values and run-time conditions. For example, `InDelimits` only delimits the lifetimes of inactive activities. Those that are active when the timingAct acts remain untouched.

Collections of timingActs form temporal scripts. Managed by processions, the temporal scripts sequence timingActs. *They describe when and why collections of interactions occur*. It is the responsibility of the activities of the timingActs to determine "who" interacts and "what" the interactions produce. Since scripts organize only timingActs, timingActs and activities may be formed independently. Reassigning new activities to existing timingActs produces a new animation with similar temporal characteristics. The outcome of the interactions may differ, but the relative order of the interactions is always the same.

Figure 5 shows the temporal script that forms the time line (see section 5.3) of Figure 9. The script assigns explicit

```
(1)  B→setRel( MEETS, C );
(2)  C→setRel( MEETS, F );
(3)  C→setRel( DELIMITS, E );
(4)  C→setTrueRel( STARTS, D );
(5)  F→setRel( STOPS, D );
     add timingActs to procession
(6)  Procession *proc = new Procession();
(7)  proc→addTimingAct( A, 1, 10 );
(8)  proc→addTimingAct( B, 30, 40 );
```

Figure 5: Temporal script of Figure 9

timing values to timingActs A and B (lines 7-8) and temporal relations to C, D, E, and F (lines 1-5). The relations indicate that C occurs after B, F occurs after C, and C and E occur simultaneously. The fourth relation indicates that C starts D if a specific condition within C occurs. The last relation indicates that F stops D when F begins.

5 Levels of Reuse

Each level of the primitive hierarchy promotes a different approach to software reuse. To access, to understand, and to adapt code for reuse, developers apply either *fine-grain*, *medium-grain*, or *coarse-grain* reuse. Fine-grain reuse pertains to events, while medium-grain and coarse-grain reuse pertain to activities and temporal scripts. Each level of reuse supports a different abstraction level to manipulate time-varying interactions.

5.1 Fine-Grain Reuse

When developers reuse individual events, they exercise fine-grain reuse. They choose, understand, and adapt quickly the event that most closely performs a desired interaction. Developers access and understand events by their kind (see section 4.1.2) and type. Together, the two identifiers outline the purpose and describe the general parameters for every event. Developers adapt events in one of two ways. They either use the events as is or they modify the events to produce a new interaction. The former approach modifies an event's arguments while the latter approach modifies an event's code segment.

Reuse is uncomplicated because events are small and specialized. Modifications to events range from changing an event's parameter list to altering an event's means of passing information. In general terms, event modifications alter "who" interacts or "how" interactions occur. For example, the code in Figure 6 adapts the sample event of Figure 3 to forward the greater value from two outports to an inport. The augmented event accepts an extra outport and performs an additional logical operation.

5.2 Medium-Grain Reuse

Developers exercise medium-grain reuse when they reuse activities. They choose, understand, and adapt activities to form similar, yet slightly differing behaviors. To

```
void Event::update()
{
        verify that all ports are compatible
  if (outPort→getTypeId() == inPort→getTypeId() &&
    (outPort2→getTypeId() == inPort→getTypeId())
        verify that inPort is ready to accept data
    if (inPort→getState() == TRUE)
          compare values from both outputs
      if (outPort→getValue()> outPort2→getValue())
        inPort→doHandler( outPort→getValue() )
      else
        inPort→doHandler( outPort2→getValue() )
}
```

Figure 6: Update routine for sample event

Descriptor

- ObjectA chases ObjectB

Component List (*ports*)

- ObjectA (in Point3, out Point3)
- ObjectB (out Point3)
- Interpolator (in double, out Point3)

Interaction Table

1. **DurationEvent**: Interpolator receives durational value.
2. **Event**: Interpolator receives Point3 (position) from ObjectA.
3. **Event**: Interpolator receives Point3 (position) from ObjectB.
4. **TimeEvent**: Interpolator receives timing value.
5. **Event**: Interpolator computes a Point3 (new position) for ObjectA.

Figure 7: Behavioral description

select appropriate activities, developers peruse *behavioral descriptions*. Behavioral descriptions outline the goals of activities by enumerating main components and general interactions. For each component, the descriptions express its purpose and identify its ports. For each interaction, the description identify its event type and its participants.

The behavioral description of activity `chase` is shown in Figure 7. It consists of three parts: a *descriptor*, a *component list*, and an *interaction table*. The descriptor identifies the goal of the activity: `objectA` chases `objectB`. The component list identifies components and ports. The activity consists of two objects and one interpolator. Port information identifies what type of data the interactions of the activity manipulate. Often, this information is more useful than is the identification of components. The activity produces interactions among ports, not components. So, any component that provides the same type of port works equally well. The interaction table highlights important interactions among components. It indicates that interpolator interacts with `objectB` after it receives positional information and temporal data.

Once relevant activities have been selected, developers assess the activities by their events. They mentally sequence through the events to observe how data flows from component to component. The resultant dataflow determines how developers partition activities into reusable parts and extract critical sequences of events. For example, the diagram in Figure 8 illustrates the partitioning and reuse of two activities to form a third. The third combines elements from the first two with additional events to form a new behavior. The resultant activity customizes the events with new parameters, and reorders them, if necessary, to form the proper interactions.

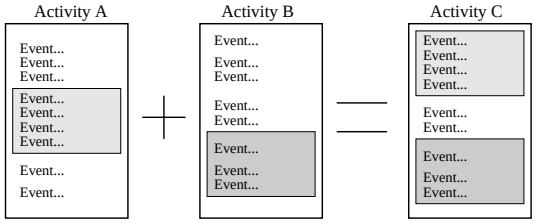

Figure 8: Medium-Grain Reuse

5.3 Coarse-Grain Reuse

Coarse-level reuse occurs when developers reuse the timingActs of temporal scripts. They find and adapt the script(s) that closely match a desired temporal sequence. Reuse of sequence, not interactions or behavior, is the goal of coarse-level reuse. To access appropriate scripts, developers scan *temporal descriptions*. Similar to behavioral descriptions, yet differing in content, temporal descriptions describe visually the relative order of timingActs over time. The visual representation depicts a time line with timingActs dispersed temporally. The time line identifies relationships between timingActs with conditions and relations, not with explicit values of time.

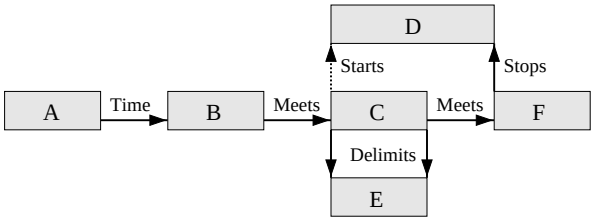

Figure 9: Timeline: (1) Timing values separate A and B; (2) F follows C which follows B; (3) C delimits E; (4) F stops D which C conditionally starts.

A sample visualization, shown in Figure 9, identifies a timeline with six generic timingActs and six relations. The

placement of timingActs indicates sequence while the labels on arrows identify relations. The stippled arrow identifies a conditional relationship between C and D. C starts D if and only if special states arise. The generic labels on the timingActs indicate that the timeline is not dependent upon specific interactions or behaviors. Any timingAct substitutes freely.

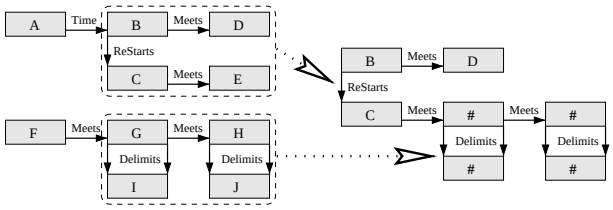

Figure 10: Coarse-Grain Reuse

Once a script is chosen, developers partition scripts sequentially and extract the useful sequences. Afterwards, they either parameterize the sequences with new behaviors, or modify the sequences with the original activities of each timingAct intact. The former action produces animations that are temporally equivalent, while the latter produces animations that are behaviorally equivalent. For example, the illustration in Figure 10 blends elements from two scripts to form a third. The resultant script employs both the script and timingActs of the first, but only the script of the second.

6 Discussion

6.1 Application Development

Numerous animation sequences have been successfully built with the RASP toolkit. As can been seen by browsing the Rasp webpage[3] or reviewing the illustrative examples of [16], the interaction-based approach applies well to popular animation techniques, such as key-framing, behavioral modeling, and physically-based motion. Interactions coordinate communications among the objects in a scene and the algorithmic components that modify them. Introducing variations to an animation often requires only changing the interactions, not the components. A wide variety of effects occur when interactions filter, reorder, or alter data as it flows between components.

From the author's experience, initial reuse with RASP is medium-grain. Activities and their behaviors provide the most benefits. Animators (developers) typically think of behaviors before they think of individual interactions or scripts. The behaviors control the most important elements of an animation - they determine how states change. After appropriate behaviors have been selected, the next phase of reuse is either fine-grain or coarse-grain. Fine-grain reuse occurs first if the behaviors necessitate amendments. Otherwise, the animator may choose initially to refine the scripts before she updates the individual interactions.

6.2 Drawbacks

Developing reusable applications with RASP entails three drawbacks. First, RASP requires developers to familiarize themselves with interaction-based programming. Interactions, not functions calls, establish the structure of communications within an application. The order and the attributes of interactions determine when and how often communications occur. Second, RASP requires developers to build *time-invariant components* with ports. Time-invariant components always receive information from their environments to change state. Components that automatically change state over time are difficult to manage. The tools of the primitive hierarchy are not designed to interface with components that maintain their own notion of time. Third, RASP requires developers to augment their code with two descriptions, behavioral and temporal. The current version of RASP provides few guidelines to create these descriptions. No process exists to determine if a description adequately presents information that is proper and useful.

7 Related Work

7.1 Software Engineering

Much work in software reusability attempts to identify, assess, and construct reusable components, not reusable interactions. Popular research topics involving components include complexity measurement [18], compositional development [3], and domain engineering [19, 5]. Integrating components with interactions shares many commonalities with entity-relationship (ER) data modeling and software architecture design. Both methods distinguish between elements that compute and elements that interconnect.

ER data modeling methods, such as those proposed by Chen [4], Rumbaugh [21], Sullivan [24], and Helm et al [10], employ "relations" to identify interactions between independently computing components and processes. They establish behavioral relationships that are easy to assess and to use without modifying components. Although similar to the interactions of this paper, relations of ER methods are typically more complex and not as adaptable. To express complex relationships, such as triggers, the primitives intermix control statements with logical expressions. The logical expressions produce changes to variables that are difficult to observe and to coordinate globally; thus, they are not well suited for animation. Except for Helm et al's work, the ER methods do not readily produce generic relationships between components. Relations are often bound tightly to a specific component. This makes them harder to

[3] `www.cs.ubc.ca/spider/gslee/Rasp/rasp.html`

adapt and difficult to reuse. Helm et al's work emphasizes genericity of relationships, but does not provide a medium to implement them. Their work is intended for specification, not implementation.

Software architecture specification methods, such as DARWIN [17] and UNICON [22], employ architectural connections to connect components and mediate interactions. The connections - first-class, high-level abstractions - regulate data flow, data access, and resource allocation within a program. The specification methods employ port-like structures and advanced compilers to ensure the connections are generic. To create animations with these methods requires great work. The methods neither control interactions dynamically nor organize connections hierarchically. There exist few structures to sequence interactions and to reuse collections of interactions efficiently. The goal of these specification methods is to integrate components, not to govern them over time.

7.2 Computer Graphics

Most tools for computer graphics employ standard programming techniques, such as explicit invocation and data abstraction, to aid in application development. They consist of components and functions that developers organize and invoke. For modeling and rendering, these tools have proven useful and have widespread use. With popular component-based tools, such as INVENTOR [23] and QUICKDRAW-3D [1], developers freely exchange and rehash existing applications. However, for animation, these tools have provided limited success. The most common component for animation stores collections of "time,value" pairs, commonly referred to as *keyframes*. Although useful, this approach, as employed by TWIXT [8] and SWAMP [2], constrains developers to specify animation at one, very low-level of abstraction.

Most tools for animation, such as those proffered by GROOP [14] and ASAS [20], consist of components that encapsulate geometry, time-varying behaviors, and temporal progression techniques. The tools manage the interconnections among the components implicitly, or they require developers to manage the connections explicitly. Either way, the tools emphasize the reuse of computation, not interaction. To reuse existing behaviors, developers either modify components and preserve interactions, or modify interactions and preserve components. The former approach encourages the reuse of algorithms - a difficult process that is not fully understood. The latter approach requires developers to overlay components with structures for communication. Without proper tools that manage these communications, the results are difficult to manage and hard to re-employ.

Few tools in computer graphics organize interactions. Of the few that do, such as Conman [9], Condor [12], and Bramble [7], none are designed for animation. They interconnect components to compose formulas and algorithms. Few primitives exist to control the interactions dynamically or to manage the interactions hierarchically. The interaction are neither first-class nor separate from the components they interconnect; thus, they are difficult to identify and to extract for reuse.

8 Conclusion

Computer animation is the interaction of components over time. The components establish the state of an animation while the interactions change the state of the components. Thus, the interactions that occur in animation are equally important as the components of the animation. For applications that animate, they establish structure and identify relations. Interactions which are first-class and transferable are reusable. Reusable interactions mediate communications among multiple kinds of components in multiple types of settings. They are easy to organize and to control hierarchically. They operate instantaneously or delay until needed. Interactions accepting temporal attributes are time-varying. They operate as simulation time progresses. Time-varying, delayed interactions interact well with operations that sequence communications. Sequence, an uncommon theme in software engineering, but vital to animation, orders the operation of interactions. It determines which interactions occur in parallel and which occur in succession.

RASP, an experimental toolkit for computer animation, employs reusable interactions to animate the states of software components. A primitive hierarchy, composed of events, activities, timingActs, and processions, organizes the interactions and controls their execution. Events form interactions which activities order. Activities form behaviors which timingActs and processions manage. Altogether, the primitives form scripts that produce dynamic interactions. As time flows, interactions occur. The hierarchy promotes three approaches towards software reuse. Developers apply either fine-grain, medium-grain, or coarse-grain reuse to assess and to adapt existing scripts. They choose primitives by examining a parameter's parameters, type and accompanying description. The description identifies the order, intent, and timing properties of the interactions within the script.

8.1 Future Directions

More work is required to enhance RASP's usefulness. Better guidelines must exist to inform developers how to create descriptions for behaviors and temporal scripts. Techniques to retrieve and to assess a description generally work better when the information conforms to an established pattern or standard. Preliminary results indicate that a semi-automated process may help create and compare descriptions for developers. This would increase productivity

and maintain minimal standards.

Future versions of RASP will include higher-level events and constraints. Higher-level events, such as pipes, queues, and remote-procedure calls, help developers create a wider variety of applications. RASP's current set of events provides no support to develop distributed animations or resource-based simulations. Constraints are necessary to update automatically relations among components. Virtual ports establish one-way relations that update only on command. An automated process with additional features, such as bi-directional relations or a cyclic solver, would reduce the number of events developers must specify to create complex interactions.

Acknowledgments

The author would like to thank Gail Murphy, for her insightful comments and her inspiration that motived the development of this paper, and Gia Lee, for her helpful comments on grammar and style.

References

[1] APPLE COMPUTER INC. *3D Graphics Programming With QuickDraw 3D*. Addison-Wesley, Reading, MA, 1995.

[2] BAKER, M. P. Streams-Based Animation in an Object-Oriented Graphics Environment. Tech. Rep. UIUCDCS-R-90-1633, University of Illinois at Urbana-Champaign, October 1990.

[3] BATORY, D., AND O'MALLEY, S. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Transactions on Software Engineering and Methodology 1*, 4 (Oct 1992), 355–398.

[4] CHEN, P. The Entity-Relationship Model - Toward a Unified View of Data. *ACM Transaction of Database Systems 1*, 1 (1976), 9–36.

[5] DECIMA, A., WERNER, C., AND CERQUEIRA, A. The Design of Object-Oriented Software with Domain Architecture Reuse. In *3rd International Conference on Software Reuse* (Rio De Janeiro, Brazil, Nov 1994), IEEE, pp. 178–187.

[6] GLASSNER, A., Ed. *Graphics Gems*. The Graphics Gems Series. Academic Press, Inc., San Diego, CA, 1990.

[7] GLEICHER, M. *A Differential Approach to Graphical Interaction*. PhD thesis, Carnegie-Mellon University, 1994. Tech Report CMU-CS-94-217.

[8] GOMEZ, J. E. Twixt: A 3D Animation System. In *Eurographics '84* (North-Holland, 1984), Eurographics Association, Elsevier Science Publishers B. V., pp. 121–133.

[9] HAEBERLI, P. ConMan: A Visual Programming Language for Interactive Graphics. In *Computer Graphics* (1988), ACM SIGGRAPH, pp. 103–111.

[10] HELM, R., HOLLAND, I., AND GANGOPADHYAY, D. Contracts: Specifying Behavioral Composition in Object-Oriented Systems. In *Proceedings of OOPSLA/ECOOP '90* (1990), ACM, pp. 169–180.

[11] KAISER, G. E., AND GARLAN, D. Composing Software Systems from Reusable Building Block. Tech. rep., Carnegie-Mellon University, July 1986.

[12] KASS, M. CONDOR: Constraint-Based Dataflow. In *Computer Graphics* (Chicago, IL, July 1992), ACM SIGGRAPH, pp. 321–330.

[13] KICZALES, G., LAMPING, J., MENDHEKAR, A., MAEDA, C., LOPES, C., LOINGTIER, J.-M., AND IRWIN, J. Aspect-Oriented Programming. Tech. Rep. SPL97-008, Xerox Parc, Palo Alto, CA, February 1997.

[14] KOVED, L., AND WOOTEN, W. L. GROOP: An Object-Oriented Toolkit for Animated 3D Graphics. In *OOPSLA '93* (1993), ACM, pp. 309–325.

[15] LEE, G. S. An Object-Oriented Graphics Kernel. Tech. Rep. TR-97-06, Univ. of British Columbia, Jan 1997.

[16] LEE, G. S., FORSEY, D., AND PAI, D. RASP - A Visual Simulation Platform. *Submitted for publication to IEEE Transactions on Visualization and Computer Graphics* (1997).

[17] MAGEE, J., DULAY, N., AND KRAMER, J. Structuring parallel and distributed programs. *Software Engineering Journal 8*, 2 (March 1993), 73–82.

[18] MCCABE, T. J. A complexity measure. In *Proceedings of the 2nd International Conference on Software Engineering* (Oct. 1976), IEEE Computer Society Press, p. 407.

[19] NEIGHBORS, J. M. The Draco Approach to Constructing Software from Reusable Components. *IEEE Trans. Software Engineering* (Sept 1984), 564–574.

[20] REYNOLDS, C. Computer Animation with Scripts and Actors. In *Computer Graphics* (July 1982), ACM SIGGRAPH, pp. 289–296.

[21] RUMBAUGH, J. Relations as Semantic Constructs in an Object-Oriented Language. In *OOPSLA'87 Conference Proceedings* (1987), ACM, pp. 466–481.

[22] SHAW, M., DELINE, R., AND ZELESNIK, G. Abstractions and Implementations for Architectural Connections. In *Third International Conference on Configurable Distributed Systems* (May 1996).

[23] STRAUSS, P. S., AND CAREY, R. An Object-Oriented 3D Graphics Toolkit. In *Computer Graphics* (Chicago, IL, July 1992), ACM SIGGRAPH, pp. 341–349.

[24] SULLIVAN, K. J. *Mediators: Easing the Design and Evolution of Integrated Systems*. PhD thesis, Univ. of Washington, 1994. Tech Report 94-08-01.