Planning-Based Control of Interface Animation

David Kurlander and Daniel T. Ling

Microsoft Research
One Microsoft Way
Redmond, WA 98052
(206) 936-2285
E-Mail: djk@microsoft.com

ABSTRACT

Animations express a sense of process and continuity that is difficult to convey through other techniques. Although interfaces can often benefit from animation, User Interface Management Systems (UIMSs) rarely provide the tools necessary to easily support complex, state-dependent application output, such as animations. Here we describe Player, an interface component that facilitates sequencing these animations. One difficulty of integrating animations into interactive systems is that animation scripts typically only work in very specific contexts. Care must be taken to establish the required context prior to executing an animation. Player employs a precondition and postcondition-based specification language, and automatically computes which animation scripts should be invoked to establish the necessary state. Player's specification language has been designed to make it easy to express the desired behavior of animation controllers. Since planning can be a time-consuming process inappropriate for interactive systems, Player precompiles the plan-based specification into a state machine that executes far more quickly. Serving as an animation controller, Player hides animation script dependencies from the application. Player has been incorporated into the Persona UIMS, and is currently used in the Peedy application.

KEYWORDS: Animation, planning, User Interface Management Systems, UIMS, user interface components, 3D interfaces.

INTRODUCTION

With the advent of inexpensive graphics rendering hardware and faster computers, interface animation will become commonplace. One building block of many such interfaces will be a User Interface Management System (UIMS) component that controls the sequencing of animation actions. We have developed such a component, called Player, that simplifies the task of coordinating these animations. Player supports a convenient plan-based specification of animation actions, and compiles this specification into a representation that can be executed efficiently at run time.

To better motivate the need for this capability in a UIMS, consider the Persona UIMS currently being developed by our group, and a prototype agent-based interface called Peedy, built on Persona, and demonstrated at CHI '94 [1]. Peedy's visual representation is that of a 3D animated parrot. Peedy has a rich repertoire of animated bird-like and human-like behaviors, and responds to spoken natural language requests for musical selections. The Peedy application is shown in a video clip accompanying electronic distributions of this paper.

Figure 1a illustrates the slice of the Persona UIMS's architecture that handles output. The application sends control events to the animation controller. This controller interprets the incoming events according to its current internal state, informs the low level graphics system (called Reactor) what animations to perform, and adjusts its own current internal state accordingly.

For example, consider the path of actions when the user asks Peedy "What do you have by Bonnie Raitt?" This is illustrated in Figure 1b. First the application interprets the

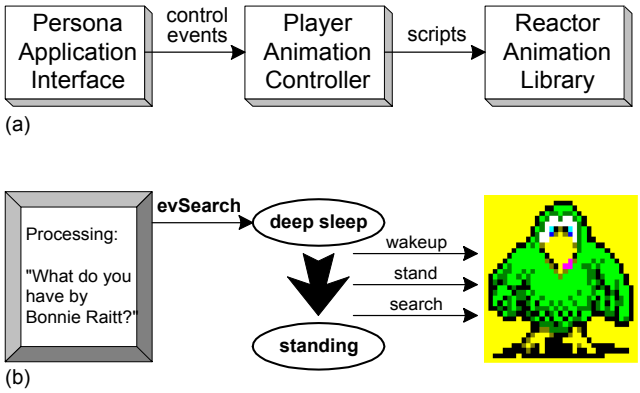

FIGURE 1. Persona and Peedy. (a) architecture of the output component of the Persona UIMS; (b) an example of its use in the Peedy application.

Reprinted from the *Proceedings of CHI '95* (Denver, May 7-11), pp. 472-479.
Technical Report MSR-TR-95-21. January 1995. Microsoft Research, One Microsoft Way, Redmond, WA 98052

message, and sends an **evSearch** event to the animation controller, to have Peedy search for the disc. The animation controller knows that Peedy is in his "deep sleep" state, so it sequentially invokes the wakeup, standup, and search animations. It also changes Peedy's current state (as represented in the animation controller) to standing, so that if another **evSearch** event is received immediately, Peedy will forego the wakeup and standup animations, and immediately perform a search.

One can view the animation controller as a state machine, that interprets input events in the context of its current state, to produce animation actions and enter a new state. Originally we specified the animation controller procedurally as a state machine, but as new events, actions, and states were added, the controller became unwieldy, and very difficult to modify and debug. It became clear that we needed a different manner of specifying the controller's behavior. One of the difficulties of specifying this behavior is that graphical actions make sense only in limited contexts for either semantic reasons (Peedy cannot sleep and search at the same time) or animation considerations (the search script was authored with the expectation that Peedy would be in a standing position).

In traditional UIMSs, the programmer must specify all these transitions, which can be a tedious and error-prone process. The Player component of the Persona UIMS calculates these transitions automatically, freeing the implementer from part of the chore of constructing animated interfaces. To accomplish this, Player relies on *planning*, a technique traditionally used by the AI community to determine the sequence of operators necessary to get from an initial start state to a goal state. In our system, the operators that affect system state are animation scripts, and the programmer declares preconditions and postconditions that explain how each of the scripts depend on and modify state. One of the major problems with planning algorithms is that they are computationally intensive. Animation controllers, however, have to operate in real time. Our solution is to precompile the conveniently specified planning notation into an efficient to execute state machine.

There are several contributions of this work. We describe a UIMS component that isolates animation dependencies from the application, and identify a valuable use of planning technology within this component. To satisfy the real-time constraints of a user interface, we present an algorithm suitable for converting a plan-based specification into a state machine. Unlike traditional AI planning techniques, this algorithm must find transitions to a goal from any possible state, not just a single start state. We describe a technique for doing this efficiently, exploiting coherence in the search space. Our language for specifying animation controllers is described in this paper. It includes provisions for handling goal-oriented behaviors (such as speaking), as well as autonomous actions (such as snoring). Our system employs a novel state hierarchy to simplify the task of specifying preconditions and postconditions. A final contribution of this work is an implementation within the Persona UIMS, and its use within the Peedy application, as a proof of concept of this research.

The next section describes other research that shares some of the same goals or uses related techniques. Following that, we present our language for specifying the behavior of the animation controller. The subsequent section explains Player's planning algorithm. Additional implementation issues are described following this, and then the paper presents our conclusions and possible future directions.

RELATED WORK

One way to discuss UIMSs is in terms of the Seeheim model [14], presented in Figure 2. The user interacts directly with the *presentation* component, which passes interaction events on to the *dialogue control*. The dialogue control component then determines which application services should be requested through the *application interface model*. To produce output, the application interface model can drive the presentation component to display graphics and play sound. Alternatively, it can send events to the dialogue control, which in turn can drive the presentation. Note that the Persona output architecture of Figure 1a can be mapped directly to the Seeheim model of Figure 2, with the animation controller serving as the dialogue control, and Reactor handling presentation services.

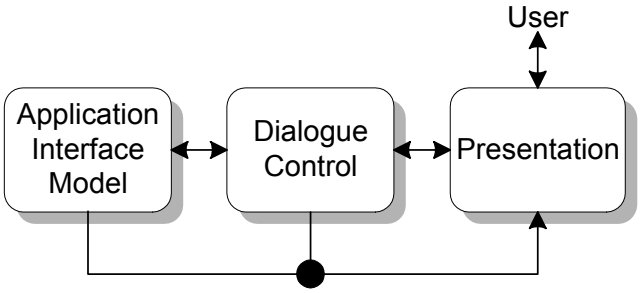

FIGURE 2. The Seeheim UIMS model (adapted from [14]).

UIMSs have employed several different techniques for dialogue control. Green compares the state machine, grammar, and event-based approaches in [9]. However, these techniques have focused on interpreting sequences of input, producing graphics only incidentally in the process. Olsen writes, "The notion of dialog control having an influence over the presentation of application data is signified in the diagram [Figure 2] by the small circle around the path from the application interface to the presentation. This data display facet of the model has not been fully realized in any UIMS" [14]. It is this facet that Player attempts to address.

The UIDE system also relies on preconditions and postconditions for dialogue control [5] [8], using them to determine when to enable application actions, and to support interface transformations. However, UIDE does not use planning to automatically sequence visual presentations. An exception to this is Sukaviriya's Cartoonist system, that extends the

UIDE model to automatically provide context-sensitive help animations [16]. UIDE also uses planning to generate context-sensitive textual help [4]. However, the UIDE work does not use planning to sequence general animation presentations, and does not employ precompilation techniques to maintain real-time constraints.

Many researchers have addressed other aspects of incorporating animation into the interface. For example, PARC's Cognitive Co-processor deals with timing issues [2]. Several interfaces incorporate traditional animation effects [3] [10]. Virtual reality research frequently coordinates simulation and novel interface devices with animation [13].

Researchers in artificial intelligence have studied planning issues extensively, and their work is summarized in several surveys [7] [6]. The planning technique described here is a simple variant of goal regression, and is not intended to contribute to the planning literature. However, we do believe that this paper presents a novel application for planning technology. Our approach differs from traditional planning by precompiling all necessary plans prior to execution. Schoppers also precompiles plans, though his methods differ somewhat from ours and he applies his work to the robotics domain [15].

Others have applied planning to computer animation. For example, Lengyel uses graphics hardware to accelerate path planning, and then animates his results [12]. Koga uses special purpose planning techniques to compute grasping animations [11]. None of this work exploits precompilation, nor is integrated in a UIMS.

THE LANGUAGE

This section describes the language used to express the desired run-time behavior of the animation controller. Presenting this language here serves two purposes. The language helps us communicate the capabilities of the controller. Also the language has been refined over time to make behavior specification easier, so our design choices should be of general interest.

There are five components to the language. Recall that the animation controller accepts high-level animation events and outputs animations scripts. So the language must contain both event and script definitions. The language also contains constructs for defining state variables that represent animation state, autonomous actions called autoscripts, and a state class hierarchy that makes defining preconditions easier. Each of these language constructs will now be described in turn.

State variables

State variables represent those components of the animation configuration that may need to be considered when determining whether a script can be invoked. State variable definitions take on the form:

(state-variable name type initial-value <values>)

All expressions in the language are LISP s-expressions (thus the parentheses), and bracketed values represent optional parameters. The first three arguments indicate the name, type, and initial value of the variable. State variables can be of type boolean, integer, float, or string. The last argument is an optional list of possible values for the variable. This can turn potentially infinitely-valued types, such as strings, into types that can take on a limited set of values (enumeration types). Examples of state-variable definitions are:

(state-variable 'holding-note 'boolean false)

(state-variable 'posture 'string 'stand '(fly stand sit))

The first definition creates a variable called **holding-note**, which is a boolean and has an initial value of false. The second creates a variable called **posture**, which is a string that is initialized to **stand**. It can take on only three values (**fly**, **stand**, and **sit**), and this should be expressed to the system because in some cases the system can reason about the value of the variable by knowing what it is not.

There is a special class of state variable, called a time variable. Time variables are set to the last time one of a group of events was processed.

Autoscripts

Autoscripts make it easy to define autonomous actions, which are actions that occur typically continuously when the animation system is in a particular set of states. Examples of this would be having an animated character snore when it is asleep, or swing its legs when it is bored. Autoscripts are procedures that are executed whenever a state variable takes on a particular value. For example, to have the **snore** procedure called when a variable called **alert** is set to **sleep**, we write the following:

(autoscript 'alert 'sleep '(snore))

The third argument is a list, because we may want to associate multiple autonomous actions with a given state variable value. Note that though we typically bind autoscripts to a single value of a state variable, we could have an autoscript run whenever an arbitrary logical expression of state variables is true, by binding the autoscript to multiple variable values, and evaluating whether the expression is true within the autoscript itself before proceeding with the action.

Event definitions

For every event that might be received by the animation controller, an event definition specifies at a high-level what needs to be accomplished and the desired timing. Event definitions take on the form:

(event name <directives>*)

The term **<directives>*** represents a diverse set of statements that can appear in any number and combination. The **:state** directive tells the controller to perform the sequence of operations necessary to achieve a particular state. The single argument to this directive is a logical expression of state variables, permitting conjunction, disjunction, and negation.

This high-level specification declares the desired results, not how to attain these results. In contrast, the **:op** directive instructs the system to perform the operation specified as its only argument. The animation controller may not be in a state allowing the desired operation to be executed. In this case, the controller will initially perform other operations necessary to attain this state, and then execute the specified operation.

For example, the **evBadSpeech** event is received by Player whenever our animated agent cannot recognize an utterance with sufficient confidence. Its effect is to have Peedy raise his wing to his ear, and say "Huh?" This event definition is as follows:

```
(event 'evBadSpeech :state 'wing-at-ear :op 'huh)
```

When an **evBadSpeech** event comes over the wire, the controller dispatches animations so that the expression **wing-at-ear** (a single state variable) is true. It then makes sure that the preconditions of the **huh** operator are satisfied, and then executes it. Note that **wing-at-ear** could have been defined as a precondition for the **huh** operator, and then the **:state** directive could have been omitted above. However, we chose to specify the behavior this way, because we might want **huh** to be executed in some cases when **wing-at-ear** is false.

By default, the directives are achieved sequentially in time. Above, **wing-at-ear** is made to be true, and immediately afterwards **huh** is executed. The **:label** and **:time** directives allow us to override this behavior, and define more flexible sequencing. The **:label** directive assigns a name to the point in time at which it appears in the directives sequence. The **:time** directive adjusts the current time in one of these sequences. An example follows:

```
(event  'evThanks
        :op 'bow
        :label 'a
        :time '(+ (label a) 3)
        :op 'camgoodbye
        :time '(+ (label a) 5)
        :op 'sit)
```

As defined above, when the animation controller receives an **evThanks** event, Peedy will bow. The label **a** represents the time immediately after the bow due to its position in the sequence. The first **:time** directive adjusts the scheduling clock to 3 seconds after the bow completes, making this the time that the **camgoodbye** operator executes, moving the camera to the "goodbye" position. The second **:time** directive sets the scheduling clock to 5 seconds after the bow, and then Peedy sits. If Peedy must perform an initial sequence of actions to satisfy the sit precondition, these will begin at this time, and the sit operation will occur later. Note that these two timing directives allow operations to be scheduled in parallel or sequentially.

Four additional directives are used, albeit less frequently. The **:if** statement allows a block of other directives to be executed only if a logical expression is true. This allows us, for example, to branch and select very different animation goals based on the current state. Occasionally it is easier to specify a set of actions in terms of a state machine, rather than as a plan. The **:add** and **:sub** directives change the values of state variables, and in conjunction with the **:if** directive, allow small state machines to be incorporated in the controller code. The **:code** directive allows arbitrary C++ code to be embedded in the controller program.

Operator definitions

Scripts are the operators that act on our graphical scene, often changing the scene's state in the process. Operator definitions are of the following form:

```
(op opname    <:script scriptname>
              <:precond precondition>
              <:add postcondition>
              <:sub postcondition>
              <:must-ask boolean>)
```

This creates an operator named **opname** associated with the script called **scriptname**. The operator can only execute when the specified **precondition** is true, and the **postcondition** is typically specified relative to this **precondition** using **:add** or **:sub**. Since operators typically change only a few aspects of the state, relative specification is usually easiest. The **:must-ask** directive defaults to false, indicating that the planner is free to use the operator during the planning process. When **:must-ask** is true, the operator will only be used if explicitly requested in the **:op** directive of an event definition. An example script definition appears below:

```
(op  'search
     :script 'stream
     :precond '((not holding-note) and ...)
     :add 'holding-note)
```

This defines an operator named **search**, associated with a script called **stream**. The precondition is a complex logical expression that the state class hierarchy, described in the next section, helps to simplify. The part shown here says that Peedy cannot be holding a note before executing a search. After executing the search, all of the preconditions will still hold, except **holding-note** will be true.

Though we have so far referred to operators and scripts interchangeably, there are really several different types of operators in Player. Operators can be static scripts, dynamic scripts (procedures that execute scripts), or arbitrary code. In the latter two cases, the **:director** or **:code** directives replace the **:script** directive.

We can also define macro-operators, which are sequences of operators that together modify the system state. As an example, the **hard-wake** macro-operator appears below:

```
(macro-op  'hard-wake
           :precond '(alert.snore and ...)
           :add 'alert.awake
           :seq '(:op snort :op exhale :op focus))
```

The above expression defines a macro-operator that can only be executed when, among other things, the value of

alert is snore. Here, the '.' ("dot") comparator denotes equality. Afterwards, the value of alert will be awake. The effect of invoking this macro-operator is equivalent to executing the snort, exhale, and focus operators in sequence, making Peedy snort, exhale, then focus at the camera in transitioning from a snoring sleep to wakefulness in our application. The **:time** and **:label** directives can also appear in a macro definition to control the relative start times of the operators, however, our system requires that care be taken to avoid scheduling interfering operators concurrently.

State class hierarchy

In the last two examples, the preconditions were too complex to fit on a single line, so parts were omitted. Writing preconditions can be a slow, tedious process, especially in the presence of many interdependent state variables. To simplify the task, we allow programmers to create a state class hierarchy to be used in specifying preconditions. For example the complete precondition for the search operator defined earlier is:

```
((not holding-note) and alert.awake and
    posture.stand and (not wing-to-ear) and
    (not wearing-phones))
```

Since this precondition is shared by five different operators, we defined a state class (called **standing-noteless**) that represents the expression, and is used as the precondition for these operators. This makes the initial specification easier, but also subsequent modification, since changes can be made in a single place.

Class definitions take the following form:

```
(state-class classname states)
```

State class hierarchies support multiple inheritance. Here, **states** is a list of state variable expressions or previously defined state classes. A state-class typically inherits from all of these states, and in the case of conflicts, the latter states take precedence. State hierarchies can be arbitrarily deep. The **stand-noteless** class is not actually defined as the complex expression presented earlier, but as:

```
(state-class stand-noteless
             '(stand-op (not holding-note)))
```

In other words, the **stand-noteless** class inherits from another class called **stand-op**. Figure 3 shows most of the state class hierarchy used in the Peedy system, with extra detail given for the descendents of **ack-op**. It shows the expressions that each class adds to those inherited from its parents.

We have found that the semantics of an application and its animations tend to reveal a natural class hierarchy. For example, for our animated character to respond with an action, he must be awake, and for him to acknowledge the user with an action, he must not have his wing to his ear as if he could not hear, and cannot be wearing headphones. These three requirements comprise the class **ack-op** (for acknowledgment operation), from which most of our operations inherit, at least indirectly.

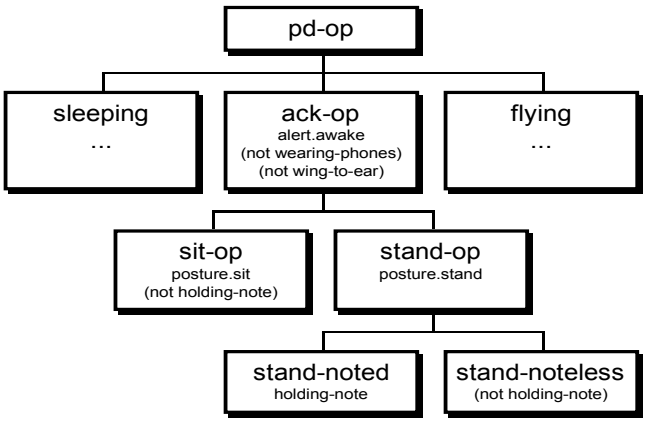

FIGURE 3. The state class hierarchy used in the Peedy system. Class names are in the larger font. Classes inherit state components from their ancestors.

ALGORITHM

Typical planning algorithms take a start state, goal state, and set of operators, compute for a while, then return a sequence of operators that transforms the start into the goal. Since our animated interface must exhibit real-time performance, planning at run-time is not an option. Instead, Player precompiles the plan-based specification into a state machine that has much better performance. This places an unusual requirement on the planning algorithm—it must find paths from any state in which the system might be to every specified goal state.

A naive approach might apply a conventional planner to each of these states and goals independently. Fortunately, there is coherence in the problem space that a simple variation of a traditional planning algorithm allows us to exploit. Our planning algorithm appears in Figure 4, and like other goal regression planners, works by beginning with goals and applying operator inverses until finding the desired start state (or in our case, start states). The algorithm is a breadth-first planner, and is guaranteed to find the shortest sequence of operators that takes any possible start state to a desired goal.

The basic algorithm consists of two procedures: Main and an auxiliary routine called SubPlan. Main begins by iterating over the goals specified in the event definitions (line 2). These include each argument to the **:state** directives, and the preconditions of the operators specified by the **:op** directives. Often the same goal appears in multiple event definitions, so we remove duplicates after putting the goals in a canonical form that makes identifying these duplicates easy.

Next, the algorithm sets up two queues, called ResultQ and WorkingQ. These queues both hold plans, which are pairs whose first element is a state, and whose second element is

```
1  Procedure Main ():
2      for each goal state, G, specified in an event definition do
3          WorkingQ := MakeEmptyQueue()
4          ResultQ := MakeEmptyQueue()
5          Enqueue([G, NULL], ResultQ)
6          Enqueue([G, NULL], WorkingQ)
7          SubPlan(WorkingQ, G, ResultQ)
8          RecordPlan(G, ResultQ)
9      end for
10 end procedure

11 Procedure SubPlan(WorkingQ, SolvedStates, ResultQ):
12     while (NotEmpty(WorkingQ)) do
13         RootPlan := Pop(WorkingQ)
14         RootState := First(RootPlan)
15         RootOps := Second(RootPlan)
16         if (Length(RootOps) >= MAXDEPTH) exit loop
17         for each state transforming operator O do
18             if (not(RootState => not(Postconditions(O))) do
19                 NewState := TransformState(RootState, O)
20                 if (not (NewState => SolvedStates)) do
21                     NewOps := Concatenate(O, RootOps)
22                     Enqueue([NewState, NewOps], ResultQ)
23                     Enqueue([NewState, NewOps], WorkingQ)
24                     SolvedStates := SolvedStates OR NewState
25                 end if
26             end if
27         end for
28     end while
29 end procedure
```

FIGURE 4. Regression-based planner, which finds plans from all possible states, leading to specified goals. The algorithm is explained in the text.

a sequence of operators that transform this state to the current goal. ResultQ will hold all the plans (ordered from shortest to longest) found for a given goal, and WorkingQ holds those plans that perhaps can still be expanded to form other plans. Both of these queues are initialized to contain the empty plan, representing the fact that when the start state is the goal state, no operators need be executed (lines 3-6).

The SubPlan procedure performs regression-based planning, beginning with the empty plan, sequentially adding operators to it in a breadth-first manner. It takes as arguments the two queues and the parameter SolvedStates, an expression representing the disjunction of the start states for which we have already found solutions. WorkingQ represents the current positions in the search. While there are still plans in WorkingQ, we remove the first element, call it RootPlan, call this plan's start state RootState, and its operator sequence RootOps (lines 12-15). It is from these roots that we try to build the next plans. If the length of RootOps is greater than a predefined MAXDEPTH constant, then we truncate the search (line 16). This guarantees that the algorithm will terminate when presented with an infinite search space.

The procedure now considers, in sequence, concatenating each operator onto the front of RootOps (line 17). Actually, it need only consider those operators that can change RootState in some way. If the operator does not change RootState, then we have already found a plan (namely RootPlan), that transforms RootState to the goal state in fewer operations. We need not consider operators with the same preconditions as postconditions. Also, operators whose **:must-ask** flag is true (see the Language section) need not be considered, since they cannot participate in the planning process.

For each of the operators that we need to consider, we ensure that its postconditions might possibly be true in RootState (line 18). If so, we calculate the effect of applying the inverse of the operator to RootState, and call this NewState (line 19). If NewState is not subsumed by SolvedStates, then we need to build a plan for it (line 20). First, we construct the operation sequence that converts NewState to the goal state, by concatenating the new operation onto the front of RootOps to form NewOps (line 21). Next, we enqueue the plan represented by NewState and NewOps onto the ends of ResultQ and WorkingQ (lines 22-23). Finally, we express that we have now found a solution for NewState by setting SolvedStates to be the disjunction of the previous SolvedStates and NewState (line 24).

When SubPlan returns, ResultQ contains plans that take any possible state to the goal state (except those whose operation sequences would be longer than MAXDEPTH). Back in the Main procedure, these plans are recorded for this particular goal (line 7), and then Main continues to solve for other goals.

IMPLEMENTATION

The next step, after the planning algorithm finishes, is to build the actual state machine. Our system generates C++ code for the state machine, which is compiled and linked together with the Reactor animation library and various support routines. The heart of the state machine has already been calculated by the planner. Recall that plans are (state conditional, action sequence) pairs, which the planner computed for every goal state. These plans can readily be converted to if-then-else blocks, which are encapsulated into a procedure for their corresponding goal. These procedures also return a value indicating whether or not the goal state can be achieved. We refer to these procedures as *state-achieving procedures*, since they convert the existing state to a desired state.

Next, the system outputs *operator-execution procedures* for every operator referenced in event definitions. These procedures first call a state-achieving procedure, attempting to establish their precondition. If successful, the operator-execution procedures execute the operator and adjust state variables to reflect the postcondition. When multiple operators share the same precondition, their operator-execution procedures will call the same state-achieving procedures.

Finally, we generate *event procedures* for every event definition. These procedures, called whenever a new event is received from the application interface, invoke state-achieving procedures for each **:state** directive, and operator-execution procedures for each **:op** directive in the event definition. The **:time** directive produces code that manipulates a global variable, used as the start time for operator dispatch. The **:label** directive generates code to store the current value of this variable in an array, alongside other saved time values.

The planner and ancillary code for producing the state machine are implemented in Allegro Common Lisp, and run under Microsoft Windows NT. Our animation controller specification for the Peedy demo contains 7 state variables (including 1 time variable), 5 autoscripts, 43 operators, 9 state classes, and 35 event definitions. The planner had to solve for 13 unique goals, and the plans tended to be short (all were 5 operators or less). The system took about 4 seconds to generate a state machine from this controller specification on a 90 MHz Pentium.

It is important to note that in our Peedy application, not all animation is scheduled via planning. We have found that low-level animation actions, such as flying or blinking, are conveniently implemented as small procedural entities or state machines that are invoked by the higher-level animation planner. These state machines can be activated through autoscripts and the **:director** directive, and they can maintain their own internal state, or reference and modify the animation controller's state variables at run-time. As mentioned earlier, state machines can also be embedded into the animation controller using the event definition's **:if** directive. Our experience suggests that planning-based specification should not entirely replace procedurally based specification. The two techniques can best be used together.

CONCLUSIONS

Animation is of increasing interest to people building user interfaces, and tools must be created to facilitate incorporating animation in the interface. Animation actions or scripts often have strict dependencies on the state of the scene, so one of these tools should be concerned with tracking these dependencies and guaranteeing that they are satisfied. We have built such a tool, called Player, that uses a precondition and postcondition based specification to encode these dependencies, and automatically determine the sequence of animation actions that must be executed when a high-level animation event is received.

Effectively, this raises the level of the protocol used by the application interface to put graphics on the screen. On the input side, traditional UIMSs rely on a dialogue control subsystem to mediate between low-level input events from the presentation component and higher-level application services. However, on the output side, the application interface typically controls the screen state directly through low-level calls to the presentation interface. For complex presentations, such as animations, this low-level output dialogue is inadequate, since it requires that the application concern itself with the detailed requirements of the animation scripts. The animation controller described here raises the level of the output protocol, hiding presentation details from the application, by serving as an output dialogue control component. By acting as a dialogue control for output, Player establishes a link from the dialogue component to the presentation component of the Seeheim model, which though called for by the model, is typically ignored by UIMSs.

Initially we implemented an animation controller specified entirely as a state machine, but as the controller grew, the state machine became difficult to maintain and enhance. The plan-based specification described here has proven easier to work with, in part because the system automatically calculates necessary transitions between animation states. Player has been incorporated within the Persona UIMS, and is employed by the Peedy application. In building Player, we made a number of design decisions to simplify specifying the animation. We have found the need to incorporate autonomous actions, as well as goal-directed behaviors. A state class hierarchy has simplified the process of specifying operator preconditions. In event definitions, we have found it useful to provide directives for both establishing new states, and invoking particular operators. Timing directives provide necessary control over the overlap and spacing of animation actions. It has proven convenient to allow small state machines and procedures to be embedded in the plan-based specification, providing a greater degree of expressivity within the easily-specified plan-based framework.

To provide real-time interaction, particularly as planning time increases, we believe it necessary to precompile the plan-based behavior specification into a state machine. This requires an algorithm that computes plans from all possible start states to all goal states expressed in event definitions. The paper describes such an algorithm, and the process for converting the resulting plans into a state machine. We have been happy with both the ease of specifying animation control, and the performance of this approach, for controllers of the complexity described earlier.

Although we have referred to Player as an animation controller, it truly can be applied to a wide range of presentation media. In the Peedy application, Player controls the sequencing of several media types, including animation, sound, and text. It could also be used to sequence static graphics, and general multimedia systems could benefit from this technology.

FUTURE WORK

This work suggests a number of important topics for subsequent research. Planning algorithms tend not to scale well, and planning research has investigated ways of dealing with this problem. We would like to determine the realistic maximum specification size and state space complexity that can be handled by our approach, and explore acceleration tech-

niques that others have developed for traditional planning that might be also applicable to our plan precompilation method.

We have not adequately dealt with the problem of animation parallelism across events. In the current system, people can specify that two animation actions dispatched by a single event definition should execute in parallel, but they cannot indicate how two parallel event executions should affect each other. For example, if the system receives two events in rapid succession, one that requires Peedy to fly to a given location, and another requiring him to sit, there are several ways to interpret this sequence. The actions can be interpreted sequentially, allowing Peedy to sit only after he reaches his destination. The second action might preempt the first, making Peedy start to fly, and then sit instead. Alternatively the first action might override the second, circumventing the sit operation entirely. The actions could also be interleaved, with the character starting his flight, deciding to sit instead, and then reconsidering and completing his flight. A more complete animation controller would allow these alternatives to be expressed. Currently our application interface never sends conflicting events to Player simultaneously, however this should be allowed.

Several software engineering issues should be considered. We have no simple mechanism for automatically duplicating the name space of event, operator, and state variable definitions, to allow multiple identical agents to be controlled in the same scene. It would be helpful to have graphical tools for debugging and performing regression testing on plan specifications. We would like to build an animation authoring environment in which planning is performed dynamically during the development cycle, compiling plans only after the interface is debugged.

Projects are currently under way to add collision detection, inverse kinematics, and dynamics into our graphical environment. We hope to make our animation controller work seamlessly with these techniques. In addition, we would like to explore more sophisticated planning techniques, including temporal planning.

ACKNOWLEDGMENTS

The Persona and Peedy projects have benefited from the efforts of many group members. Andy Stankosky and David Pugh wrote Reactor. Tim Skelly modeled and animated Peedy. Maarten Van Dantzich wrote our graphical file format translator. David Thiel added sound to Peedy's environment, and Gene Ball dealt with natural language issues and wrote much of Persona's and Peedy's substrates.

REFERENCES

1. Ball, J. E. et al. Reactor: A System for Real-Time, Reactive Animations. In *CHI '94 Conference Companion*. (Boston, MA, April 24-28). ACM, New York. 1994. 39-40.
2. Card, S. K., Robertson, G. G., and Mackinlay, J. D. The Information Visualizer, an Information Workspace. In *CHI '91 Proceedings*. (New Orleans, LA, April 27-May 2). ACM, New York. 1991. 181-188.
3. Chang, B. and Ungar, D. Animation: From Cartoons to the User Interface. In *UIST '93 Proceedings*. (Atlanta, GA, November 3-5). ACM, New York. 1993. 45-55.
4. de Graaff, J. J., Sukaviriya, P., and van der Mast, C. A. P. G. Automatic Generation of Context-Sensitive Textual Help. Tech. Report GIT-GVU-93-11. GVU Center. Georgia Institute of Technology. 1993.
5. Foley, J. et al. A Knowledge-based User Interface Management System. In *CHI '88 Conference Proceedings*. (Washington, DC, May 15-19). ACM, New York. 1988. 67-72.
6. Genesereth, M. R., and Nilsson, N. J. *Logical Foundations of Artificial Intelligence.* Morgan Kaufmann, San Mateo, CA. 1988.
7. Georgeff, M. P. Planning. In *Readings in Planning,* edited by J. Allen et al. Morgan Kaufmann, San Mateo, CA. 1990. 5-25.
8. Gieskens, D. F., and Foley, J. D. Controlling User Interface Objects through Pre- and Postconditions. In *CHI '92 Conference Proceedings.* (Monterey, CA, May 3-7). ACM, New York. 1992. 189-194.
9. Green, M. A Survey of Three Dialogue Models. *ACM Transactions on Graphics 5,* 3. (July 1986). 244-275.
10. Hudson, S. and Stasko, J. T. Animation Support in a User Interface Toolkit: Flexible, Robust, and Reusable Abstractions. In *UIST '93 Proceedings*. (Atlanta, GA, November 3-5). ACM, New York. 1993. 57-67.
11. Koga, Y. et al. Planning Motions with Intentions. In *SIGGRAPH '94 Proceedings*. (Orlando, FL, July 24-29). ACM, New York. 1994. 395-408.
12. Lengyel, J. et al. Real-Time Robot Motion Planning Using Rasterizing Computer Graphics Hardware. In *SIGGRAPH '90 Proceedings*. (Dallas, TX, August 6-10). ACM, New York. 1990. 327-335.
13. Lewis, J. B., Koved, L., and Ling, D. T. Dialogue Structures for Virtual Worlds. In *CHI '91 Proceedings.* (New Orleans, LA, April 27 - May 2). ACM, New York. 1991. 131-136.
14. Olsen, D. R. Jr. *User Interface Management Systems: Models and Algorithms.* Morgan Kaufmann, San Mateo, CA. 1992.
15. Schoppers, M. J. Universal Plans for Reactive Robots in Unpredictable Environments. In *IJCAI '87 Conference Proceedings.* Vol. 2. (Milan, Italy, August 23-28). Morgan Kaufmann. 1039-1046.
16. Sukaviriya, P. and Foley, J. D. Coupling a UI Framework with Automatic Context-Sensitive Animated Help. In *UIST '90 Conference Proceedings.* (Snowbird, UT, October 3-5). ACM, New York. 1990. 152-166.