Temporal Scripts for Objects[1]

L. Dami
E. Fiume
O. Nierstrasz
D. Tsichritzis

Abstract

Computer animation, computer simulation, computer music and other areas often need to deal with concurrent activities with specific temporal characteristics. This paper proposes a scripting facility to help program such applications. This facility provides support for specifying long-term behaviour of objects in an object-oriented environment. Temporal scripts can be instantiated and combined using a set of temporal operators, saying for example that two activities begin at the same time, or that one has to follow the other. Through a flexible sampling policy based on a notion of virtual time, temporal specifications can be executed at various temporal resolutions, and therefore can be reused in different contexts.

1 Introduction

Programming languages and systems offer a wide variety of mechanisms to deal with concurrency. A number of techniques are available to solve common concurrent programming problems, like *mutual exclusion* or *process synchronization* [Andr83]. Typical examples of these problems, appearing in database applications (e.g. transactions), communication protocols, device drivers or operating systems can be programmed using semaphores, monitors, or message-passing primitives. Most of these examples do not need any explicit notion of time: the order in which events may occur is important, but not the actual time spent between the events. This allows most concurrent models to work with processes whose relative execution speeds are left indeterminate, so that concurrent programs can run without modification in various environments. Such programs will yield the same results on any machine, as long as the elementary temporal constraints are preserved.

Several applications areas, however, need a notion of time for other purposes than simple ordering of events. For example in discrete-event simulation the events are usually associated with *time stamps* in virtual time. Time stamps are used for ordering purposes, but also constitute relevant information for the application: various results may be derived from the event times or from the intervals between events. Virtual time in such models has nothing to do with real-time: the real-time interval between two events depends on the number of operations to perform and on the speed of the machine, not on the virtual-time interval between them. Even the virtual ordering of events need not necessarily be preserved at execution; indeed, several authors have proposed schemes

[1]Submitted for publication.

to take advantage of parallel computation by allowing local clocks to be temporarily desynchronized[Jeff85,Misra86,Bez87]. Yet such systems provide a coherent temporal model for application designers.

This paper describes a similar approach for addressing another problem encountered in temporal specifications: how to model activities that *span* over time. In numerous areas, like computer animation, computer music, simulation, process control, or robotics, programmers have to specify long-term, concurrent activities. Often these activities must execute at a similar speed. For example a robot controller must move several articulations at the same time, or a sound synthesizer must change several parameters simultaneously, like pitch and amplitude. Such synchronous activities are not easily programmed if the environment only provides processes with indeterminate execution speeds. Programmers have to juggle with semaphores, monitors or message-passing in order to control global synchrony. Generally this is done by decomposing the activities into a set of discrete steps that are triggered either by a global ticking procedure or by a collection of local clocks with some synchronization mechanism.

Decomposing a global activity into a discrete set of temporal steps can be a non-trivial problem. It is especially difficult to ensure that the program remain well-structured and that its components be reusable for different applications. Let us consider, for example, the problem of an animator who wants an animated character to walk during 10 seconds. Knowing that a film typically needs 24 images per second, he can decompose the global motion into 240 discrete moves, using current animation techniques. Unfortunately, if a similar motion is to be used later in the film for a different duration, or if the scene is to be recorded for video at a different number of frames per second, he probably will not be able to directly reuse directly the specification. The problem arises because a motion which is conceptually continuous is only known to the computer in terms of a discrete series of instructions.

Our approach is to provide a *continuous model* for defining temporal activities that supports specification, instantiation and synchronization of *temporal scripts*. Conceptually, we have a world of continuously evolving objects, which is being watched by some "observers" (such as a virtual camera for computer animation). Scripts describe the objects' evolution, and observers take discrete sequences of snapshots of the object world, in which they might look at the state of the objects in order to produce images, commands to a sound synthesizer, simulation statistics, etc. When a new state of the object world is required, scripts are given a certain amount of virtual time to "consume", so that they can update themselves. Observers are free to choose any step in virtual time between two successive snapshots; the sampling policies typically depend on the application domain. Therefore, the specification of sets of conceptual continuous activities in time is completely separated from the way they will be executed in a discrete sequence of steps. Similarly to what happens in the concurrent discrete-event simulation systems described above, the temporal concepts manipulated by the programmer may not directly reflect the sequence of events occuring at execution; yet they provide a coherent model which is closer to the needs of the application. We hope to demonstrate that temporal scripts are useful tools for building temporal applications in such a way that their components can be reusable in different contexts.

Since our main concerns, namely those of modularity and reusability of code, are also those of object-oriented programming, it was natural to build our facility in an object-oriented environment. The term "object-oriented" is becoming very fashionable, but its interpretation varies [Nier86,Weg87]; we shall consider here only a limited number of basic features of objects, that can be found in almost every system that claims to be object-oriented:

- Objects have *instance variables* which hold private information; the types of the variables are identical for all objects of the same class, but the values may vary. The *state* of an object is the set of values of its instance variables at a given time. The values persist during an instance's life, i.e. they do not change between successive accesses to the object.
- Objects are accessible through a set of *methods*. Methods hide the information within the object and are the only way to access the instance variables.

Several other important aspects of object-oriented systems, like simple or multiple inheritance, message-passing, or dynamic binding, are consistent with but not essential to our scripting model. That is, temporal scripts can be used in an environment where a mechanism like inheritance is missing; nevertheless, inheritance adds more power to the model, as some examples will demonstrate.

The next section presents an overview of PRESTO, a PRogramming Environment for Synchronous Temporal Objects. The main concepts of scripts, temporal steps and temporal operators are introduced. We then give the formal semantics of the temporal operators. Finally we show some examples from our current implementation, and conclude on possible future directions of research.

2 An overview of PRESTO

In structured programming languages, different levels of complexity can be separated by organizing a program as a hierarchy of subprograms. Similarly, we specify a temporal application as a hierarchy of *scripts*. A temporal script is a set of temporal actions that are grouped under a single name, either to hide some elementary details that need not be known at a higher level of specification, or later to allow these actions to be bound to different objects through the mechanism of parameter passing. When a script is activated, it is bound to one or several objects. The objects contain private information which defines their state. The script is a description of a particular trajectory through the state space.

The difference between temporal scripts and procedures is that scripts are driven by a notion of *virtual time*. This means that a script does not, like a procedure, execute immediately, continuing until it reaches a return instruction; once a script is activated, it can only proceed when it receives some virtual time to "consume". When it receives some amount of time, the script can progress along its local time line; we call *temporal steps* these successive advances in virtual time. Therefore, although the script conceptually

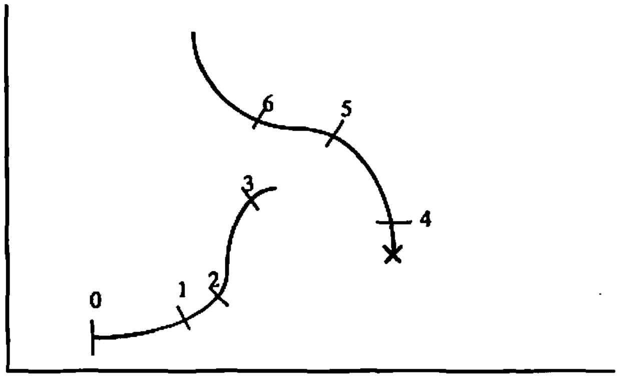

Fig 1: A script is a trajectory in the state space

Numbers are temporal units. The trajectory need not be continuous, but a state is defined for every real *t*.

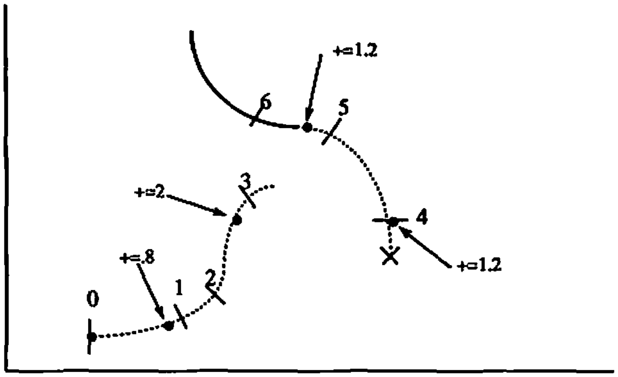

Fig 2: The script has consumed the temporal steps <0.8, 2, 1.2, 1.2>

Although the trajectory conceptually defines a state for every real *t*, the script only went through discrete jumps.The amount of time that is consumed at each jump is decided not by the script but by an external sampler. *From its current point, the script can go to any suceeding point on the time line.*

defines a state for every time *t*, where *t* is any positive real number, it will only go through a discrete sequence of points on the trajectory when executed. That sequence is not chosen by the script itself: instead of specifying within the script a sequence of steps in time, we specify *how the script would react to any temporal step*. This means that temporal steps are not merely tick signals from a clock, but have an associated duration. In other words, scripts receive sequences of messages of the form : "Age yourself by *t* units of time". So each script possesses an *aging function* that allows it to go from the current point to any next point on the state trajectory.

A PRESTO program is a set of *objects* performing temporal scripts. Temporal scripts involving only one object class can be defined within its interface, and use the private instance variables. So the set of methods for that class will consist of one or several temporal scripts, and possibly some other, non-temporal methods, like for instance a **print** method. Usually in object-oriented systems the methods define the atomicity of access to an object: an object is either idle or performing a method; while performing a method, it can be either actually computing or waiting for a return from another object. Thus, an object performs only one method at a time, and the sequence of method invocations fully determines the trace of its states. In PRESTO we allow an object to perform several temporal scripts concurrently, and even to perform non-temporal methods while executing a script, provided that the following constraints are respected:

- A particular instance variable can be used only by one script at a time.
- While an instance variable is being used by a script, no other method of the object is allowed to modify the value of the variable. Other methods can read the value, however.

The idea is that an object can be partitioned on its instance variables. For example, an animation character may be performing one script that moves its legs, another script that moves its head, and still allow other methods to set the position of its arms.

PRESTO temporal scripts are intended to be defined within an existing object-oriented environment. Host language instructions can be used to specify scripts, if they follow a particular protocol. Once a set of basic scripts has been designed, new scripts can be specified as combinations of other scripts, connected by a set of *temporal operators*. The operators introduce temporal relationships between scripts, saying for example that two activities have to begin at the same time, or that one should start 7.5 units of time after the other. So in the following example, taken from an animation script:

```
Bird bird1, bird2, bird3;
Man man1;

ScriptVar scene =
    ((bird1.flap(17) & bird2.flap(13)) >> 5*bird3.flap(10.5))
    &
    (3.5 >> man1.step(10));
```

four activations of scripts are created: three instances of the script **flap**, with different parameters, and one instance of the script **step**. They are bound together by some temporal operators, and the configuration is given the name **scene**. All activations of scripts are created at the same virtual time, but the execution of some will be delayed according to the semantics of the operators. Informally, it can be described as follows:

- bird1 and bird2 will together begin to perform one flap, but for different durations
- when *both* are finished (that is, at virtual time 17), bird3 will start flapping five times.
- beginning 3.5 temporal units after bird1 and bird2, man1 will make one step, for a duration of 10 units.

3 Scripts and temporal operators

We mentioned earlier that at any point during its state trajectory, a script must possess an *aging function* through which it can proceed to successive points on the trajectory. This function will be written '+='. So if S_t is an activation of the script S at time t, and $state(S_t)$ is the state at that time of the objects controlled by the script, then $state(S_t \mathrel{+}= x)$ is the state of these objects after the activation has been aged by the value x.

Continuity property: since we want a continuous model in which a state is defined for every time $t \in \mathcal{R}^+$, we impose the following constraint on all aging functions:

$$\forall x \in \mathcal{R}^+, state(S_t \mathrel{+}= x) = state(S_{(t+x)})$$

which has the following consequence:

$$\forall x_1, x_2 \in \mathcal{R}^+, state((S_t \mathrel{+}= x_1) \mathrel{+}= x_2) = state(S_t \mathrel{+}= (x_1 + x_2))$$

In other words, the state of the objects at any particular time does not depend on the sequence of temporal steps through which the script reached that time.

When a script decides to terminate (if ever), it may still possess some time that has not been "consumed". This amount of time has to be reported, since it can be used by another script. We shall write $timeleft(S_t)$ to designate the amount of time unused by script S at time t. So

- if $timeleft(S_t) = 0$, then S has not yet terminated at time t
- if $timeleft(S_t) = u$, where $u > 0$, then script S has terminated at time $(t - u)$.

After a script has terminated, the state of its objects does not change. So if S_t is a terminated script, $state(S_t \mathrel{+}= x) = state(S_t)$ for every positive real x.

As mentioned above, some scripts can be specified in the host object-oriented language, directly implementing the aging function '+=' and the *timeleft* function. Examples of such basic scripts will be given later, but since they are rather low-level issues, we prefer to turn now to the semantics of the temporal operators. Their semantics is expressed in terms of the '+=' and *timeleft* functions. We will consider the following operators:

1. sequential execution: $S_1 \gg S_2$
2. delay: <positive real>$\gg S$
3. parallel execution: $S_1 \& S_2$
4. repeated execution: <positive integer> $*S$
5. speed modulation: $S[f]$

3.1 Sequential execution

Sequential composition of two scripts S_1 and S_2 is written $S_1 \gg S_2$. The compound script $(S_1 \gg S_2)$ will first redirect all received temporal steps to the sub-script S_1. When S_1 terminates, its un-used virtual time is transmitted to S_2. Subsequent temporal steps received by $(S_1 \gg S_2)$ are forwarded to S_2.

This can be stated formally:

1. **aging function:** $((S_1 \gg S_2) \mathrel{+}= t) = \begin{cases} ((S_1 \mathrel{+}= t) \gg S_2) & \text{if } \mathit{timeleft}(S_1 \mathrel{+}= t) = 0 \\ (S_2 \mathrel{+}= \mathit{timeleft}(S_1 \mathrel{+}= t)) & \text{if } \mathit{timeleft}(S_1 \mathrel{+}= t) > 0 \end{cases}$
2. **timeleft function:** $\mathit{timeleft}(S_1 \gg S_2) = \mathit{timeleft}(S_2)$

3.2 Delayed execution

The expression $(x \gg S)$ states that script S has to wait x units of time before starting. Its obvious semantics can be written as follows:

1. **aging function:** $((x \gg S) \mathrel{+}= t) = \begin{cases} ((x - t) \gg S) & \text{if } x \geq t \\ (S \mathrel{+}= (t - x)) & \text{if } x < t \end{cases}$
2. **timeleft function:** $\mathit{timeleft}(x \gg S) = \mathit{timeleft}(S)$

3.3 Parallel execution

When two scripts are linked in an expression of the form $(S_1 \& S_2)$, they define a new compound script through which all temporal steps will be duplicated and sent in parallel to S_1 and S_2; $(S_1 \& S_2)$ terminates when *both* S_1 and S_2 have terminated:

1. **aging function:** $((S_1 \& S_2) \mathrel{+}= t) = ((S_1 \mathrel{+}= t) \& (S_2 \mathrel{+}= t))$.
2. **timeleft function:** $\mathit{timeleft}(S_1 \& S_2) = \mathit{min}(\mathit{timeleft}(S_1), \mathit{timeleft}(S_2))$.

3.4 Repeated execution

A script can be repeated in expressions of the form $(n * S)$, where n is a positive integer value. The definition can be given inductively:

$$n * S = \begin{cases} S & \text{if } n = 1 \\ S \gg ((n-1) * S) & \text{otherwise} \end{cases}$$

3.5 Speed modulation

Speed modulation allows external control over the apparent speed of an activity, while the activity internally always behaves at constant speed. This is a very powerful operator for a modular approach towards specification of temporal behaviour. For example, if we have an animation script in which a bird is flapping wings, and we want the bird to accelerate, we need not write a new *flapping* script: it suffices to use an *accelerate* speed modulator with the existing script. A speed modulator acts as a filter on the temporal steps. It receives temporal steps from the environment, modifies them and forwards them to the modulated script. The filter must have an inverse function to translate virtual time unused by the modulated script back into the environment's notion of time.

We write $S[f]$ for "script S is modulated by filter f". It has the following semantics:

1. **aging function:** $(S[f] \mathrel{+}= t) = (S \mathrel{+}= f(t))[f]$
2. **timeleft function:** $\mathit{timeleft}(S[f]) = f^{-1}(\mathit{timeleft}(S))$.

The filter f should be chosen so that it preserves what we called the "continuity property". If we want that

$$((S[f] \mathrel{+}= t_1) \mathrel{+}= t_2) = (S[f] \mathrel{+}= (t_1 + t_2))$$

then f should be *distributive for addition*. This can be shown easily: if $f(t_1) + f(t_2) = f(t_1 + t_2)$, then

$$\begin{aligned} ((S[f] \mathrel{+}= t_1) \mathrel{+}= t_2) &= ((S \mathrel{+}= f(t_1))[f] \mathrel{+}= t_2) \\ &= ((S \mathrel{+}= f(t_1)) \mathrel{+}= f(t_2))[f] \\ &= (S \mathrel{+}= f(t_1 + t_2))[f] \\ &= (S[f] \mathrel{+}= (t_1 + t_2)) \end{aligned}$$

3.6 Other operators

We have limited ourselves till now to the operators that seemed most generally useful. Nothing, however, should prevent a user from defining new operators to suit particular needs. The only requirement is to provide definitions of the '+=' and *timeleft* functions that respect the continuity constraint discussed above. We could imagine, for example, an **alternate** operator that would alternatively send temporal steps to several sub-scripts, or a **rounding** operator that would force temporal steps to be rounded to particular integral values.

If we possess more information about the scripts, more operators can be defined. In particular, if the total duration of scripts is always known, we can use operators like $S_1 \wr S_2$ (simultaneous termination), or $S_1[t_1] \asymp S_2[t_2]$ (general synchronization of S_1's local time T_1 with S_2's local time t_2. These operators are described in detail in [Fium87].

4 Basic scripts

Using the temporal operators, complex hierarchies of scripts can be specified. At some level, however, these hierarchies have to rely on a set of so-called "basic scripts" that directly implement the aging behaviour. The way these scripts are written is highly dependent on the host language chosen for the implementation. We currently use the C++ programming language [Strous86], mainly because temporal expressions can be easily integrated into the language by means of its *operator overloading* feature. Parallelism is implemented by a coroutine extension. Basic scripts are coroutines in which temporal behaviours are specified as sequences of "time-consuming loops". These loops are places where the programmer describes how to consume the temporal steps received by the script.

Whenever a basic script receives some time to consume, it enters its first loop. That loop has to modify the state of the script's objects according to the duration of the step. If the loop decides that the amount of time was not enough, then the script waits for more time; otherwise control is transferred to the next loop. Between successive loops, the programmer is free to insert any block of C++ instructions. These instructions have a virtual duration of zero for the temporal system; but they will be executed at the appropriate virtual time.

The simplest loop is of the form:

```
CONSUME(var)
<C++ instructions>
TIMELEFT(value)
```

When entering the loop, the variable is set to the last temporal step received by the script. The instructions are expected to modify the state of the objects according to that value. Then, if the amount of time given to the TIMELEFT macro is non-negative, control goes to the next loop.

There exist a higher-level construct built using that mechanism to facilitate the specification of temporal activities with a fixed duration. Its syntax is:

```
DURING(<arithmetic expression>, <C++ instruction>);
```

and it is implemented as a macro:

```
#define DURING(duration, C_INSTRUCTION)\
  {                                                    \
   double Duration = (duration);                       \
   double Current = 0.;                                \
   double LastTick = 0.;                               \
   double TimeLeft = -1.;                              \
   CONSUME(LastTick);                                  \
   if ((Current += LastTick) >= Duration) {            \
       LastTick -= (TimeLeft= (Current - Duration));\
       Current = Duration;                             \
   };                                                  \
   {C_INSTRUCTION}                                     \
   TIMELEFT(TimeLeft);                                 \
  }
```

Within the DURING statement, the programmer can use the variables **Current**,**LastTick** and **Duration** to specify how this script should react when receiving temporal steps. We used this construct to specify most of the basic motions in our animation library, as in the following example:

```
SCRIPT RotateX(GraphObjNode& g, double dur, double angle)
{
    DURING(dur, {
        double percent = LastTick / Duration;
        g.TM *= MatrRotX(angle*percent);
            /* multiply the object's transformation matrix
               by a rotation matrix */
        }
    )
}
```

The purpose of this script is to let the graphical object g perform a rotation of angle on its local X axis, for a total duration **dur**. In order to do this, it enters a DURING loop, so that each time a temporal step is received by the script, a partial rotation is performed, the angle of which is computed from a ratio between the last temporal step and the total duration of the motion. Eventually, when enough time has been received, the sum of the partial rotation angles will be equal to the total rotation angle, and the script will terminate. The important idea is that the final result will always be the same, regardless of the number of steps in which the rotation was performed.

Of course the code within the loop has to fulfill certain constraints if we want to avoid unpredictable results. In particular, it has to respect our continuity requirement, namely the fact that two executions of the loop with steps t_1 and t_2 are identical to a single execution of the loop with step $(t_1 + t_2)$. Scripts using temporal operators are guaranteed to be correct with respect to the continuity property, because the semantics of the operators has been chosen accordingly. There is nothing, however, that prevents a programmer from writing a basic script that does not respect this property. Unfortunately no automatic checking can be performed, because the compiler or interpreter does not have enough information about the semantics of the instructions in the loop. Hence, basic scripts have to be written with care, and are not likely to be used heavily by users with limited experience. We expect those users to work mostly with temporal operators and libraries of pre-packaged scripts.

5 Some Examples

In this section we shall present some examples of scripts from the user point of view. We hope to demonstrate with them that our approach leads to concise specifications that clearly exhibit the *global behaviour* of a system, in contrast to systems with asynchronous communication events, where the overall result of a set of local interactions might be difficult to understand from reading the specification.

5.1 Example: a simple animation script

```
#include <man.h>                        /* use the class "man" in library" */

main()
{
    Man firstMan, secondMan;            /* create two "men" in the scene */

    firstMan.rotateZ(180);     /* initial positioning */
    firstMan.translate(Vect3d(5, 4));
    secondMan.translate(Vect3d(-10));

    Camera mycamera;                    /* create a camera object */
    Vect3d v;
    cout << "initial eye position:";
    cin >> v;                           /* get eye position from user */
    Translate(mycamera.lookFrom, v);

    ScriptVar scene =                 /* specification of animation */
        firstMan.walk(20)
        &
        (3 >>secondMan.walk(14.6)[Accelerate(2)])
        &
        RotateZ(mycamera.lookFrom, 50, PI*2);
        ;


    double t;                           /* ask the user for the interval */
    cerr << "delta t?";                 /* between two frames */
    cin >> t;

    while (scene.hasValue()) {       /* while scene not terminated */
        scene += t;                  /* send temporal step to the scene */
        mycamera.shootImage();       /* take a picture */
    }
}
```

The first lines in this program indicate which libraries to use. Then two objects of class **Man** are created, and their initial position is set using basic graphical commands. The scene is defined as a script in which the first figures walks 10 units in its forward direction, for a duration of 20; 3 units of time after the beginning of **firstMan**, **secondMan** also starts walking, for a duration of 14.6, but that motion will exhibit a constant acceleration. While the two men are walking, and also after they have finished, the camera will be rotating around the scene.

At execution, the user will be prompted for the initial position of the camera and for the interval between frames. The user can try several positions, generating only a limited number of frames, until he or she finds one which is satisfactory; then that position can be kept to compute a more extensive set of frames for this animation. None of these operations will require any modification to the program.

This example demonstrates several interesting features:

- *object orientation:* the objects in this scene are abstractions that encapsulate a considerable amount of information (logical structure, graphical appearance). However, the information is *hidden*, because it is not relevant at this level of specification. Manipulation of objects is done through *methods* (e.g.: Man.walk()) that help to ensure proper use of the data within the objects. Objects can be *instantiated* (**firstMan** and **secondMan** are two instances of the class **Man**), and an *inheritance* mechanism allows one to create new classes of objects out of previous ones: the class **Man** is a subclass of a general class **GraphObj**, from which it inherits general graphical methods (**rotate**, **translate**); but the method **walk** is defined only for the class **Man**.
- *modularity:* not only the structure and the graphical definition of objects, but also the temporal activities they may perform can be defined independently and later composed.
- *flexibility:* since the scene is abstractly defined, frames can be computed at any resolution of virtual time. The same specification can thenerefore be used for both cheap prototyping and high-quality rendering.

5.2 Example: basic scripts for computer animation

The previous example only showed how to combine predefined scripts into a new one. In the next example, we specify two "basic" scripts, i.e. scripts that define changes of state without making use of sub-scripts.

The first one performs a cyclic change of the tension within a β-Spline. β-Splines are curves defined by a set of control points and two parameters β_1 and β_2 in the range [0..1]. The example performs a cyclic modification on one of these parameters, which is usually called the *tension* parameter; as a result, the shape of the curve will oscillate between a smooth appearance and a more angular one.

```
class BetaSpline : Spline{
    double bias;
    double tension;                /* range [0..1] */
public:
    SCRIPT CycleTension(double);
};


SCRIPT BetaSpline::CycleTension(double period)
{
    DURING(ETERNITY,
        {tension = (sin((Current*period) / (2*PI)) + 1) / 2;}
    );
}
```

We found this example useful to animate motions like respiration or the beating of a heart, where a shape is constantly expanding and retracting under certain mechanical constraints.

The next script performs two scaling operations on a graphical transformation matrix in a homogeneous coordinate system [Rog76]. The size of the object associated with the matrix can be changed by multiplying the diagonal of the matrix. We define a script that makes the object grow rapidly and shrink more slowly.

```
SCRIPT GrowAndShrink(GraphObjNode& g, double d, double factor)
{
    DURING(d/3, {
        double f = pow(factor, LastTick/Duration);
        for (int i = 0; i<4; i++)
            g.TM[i][i] *= f;
        }
    );
    if (debug) cerr << "now start shrinking\n";
    DURING(d*2/3, {
        double f = pow(factor, LastTick/Duration);
        for (int i = 0; i<4; i++)
            g.TM[i][i] /= f;
        }
    );
}
```

The script takes three parameters: a reference to the object that will be animated, the duration of the motion, and a factor controlling the change of size (a factor of 2 means to double the size). The first DURING statement will increase the size of the object during one third of the total duration *d*. The second DURING statement brings the object back to its initial state, but more slowly (during two thirds of *d*). Both operations multiply or divide the diagonal of the matrix by $factor^t$, where $t = (LastTick/Duration)$ is in the range [0..1] and represents a ratio between the current discrete jump in virtual time and the total virtual duration of the DURING statement. Between two DURING statements, the programmer can include any sequence of C++ instructions, like a debugging instruction in our example. The virtual duration of these instructions is zero; they will be executed at the appropriate moment in virtual time: in our example, the debugging instruction will be executed at time $d/3$.

5.3 Example: a script to control a sound synthesizer

Musical composition and synthesis computer applications are good examples of parallel applications with a global notion of time. They have to simultaneously control a number of parameters, like pitch, amplitude, or spectrum.

The FORMES object-oriented system developed at IRCAM[2] [Coin87] provides an explicit notion of time, with mechanisms for building hierarchical temporal processes . Processes are linked through *monitors*, whose function is somewhat similar to our temporal operators (but they have nothing to do with the monitors presented in [Hoare74]). Local clocks associated with processes simulate time flow in the system.

We give a short example to show how PRESTO scripts could provide functionalities similar to the FORMES system, such as generating sound envelopes, for example. Only the amplitude parameter of sounds is considered in the example:

```
class Sound{
    double amplitude;
    ...                   /* other sound parameters (pitch, etc.) */
public:
    ...
}
```

First a script to perform crescendo and decrescendo can be defined by parametric interpolation:

```
SCRIPT Sound::changeAmpl(double duration, double newAmpl)
{
    DURING(duration, {
        double t = LastTick / (Duration - Current - LastTick);
        amplitude = (1 - t)*amplitude + t*newAmpl;
    });
```

[2]Institut de Recherche et Coordination Acoustique/Musique, Paris

```
}
```

Using the sequential and delay operators, we can now define a script to generate envelopes for sounds. An envelope is a sequence of three stages: an *attack*, during which the sound augments up to a given amplitude, a *sustain* where the amplitude is stationary, and a *decay* to let the sound disappear:

```
SCRIPT Sound::envelope(double attack, double sustain,
                       double decay, double ampl)
{
   return
      (changeAmpl(attack, ampl) >> sustain >> changeAmpl(decay, 0));
}
```

The first three parameters are the duration for the three stages, and the last parameter is the amplitude at which the sound will be sustained. The operator '$\gg$' indicates both sequential execution and delay; thus, the first instance of the changeAmpl script is followed by a delay (sustain), followed again by a second instance of changeAmpl that goes through a decrescendo to perform the decay.

Similarly, higher-level musical scripts can be defined by specifying sequential or parallel execution of several envelopes. Musical parameters like *legato* or *staccato* can be introduced by choosing appropriate delays between the notes, or by slightly overlapping successive envelopes. Scripts controlling other parameters (pitch, timbre) can be executed concurrently.

6 Conclusion

We have developed a system to provide programmers with tools to manipulate temporal scripts. Our temporal environment has been fully implemented first in the C language and then in C++ on a network of Sun 3 workstations running the Unix[3] operating system. The *parameterization* of scripts, and their combinations using a set of *operators* gives a high degree of flexibility for modular building of temporal applications. The independence between specification in terms of continuous time, and execution in a discrete number of steps enhances reusability of temporal behaviours in different contexts. Various sampling policies can be designed to best suit the application's needs; here are some suggestions:

- For computer animation, the goal is typically to generate a sequence of frames that are evenly spaced in time (typically 24 images per second). So, assuming that we want virtual-time units to be directly mapped to real-time seconds, the sampler should merely issue a sequence of temporal steps of 1/24. To see the same scene at double speed, the temporal steps could be 1/12.

[3]UNIX is a trademark of ATT Bell Laboratories

- If some periods of a temporal simulation have to be analyzed in finer detail than others, we could imagine an *interactive* sampler that allows the user to choose the next temporal step after having examined the current state of the system. Such a sampler provides an appropriate solution for debugging temporal applications.
- A sampler could be *adaptive*, trying to map its notion of virtual time with the real time on its host machine. Since the two notions of time are independent, the real time needed to age the hierarchy of scripts by a virtual step t does not depend on t: it only depends on a ratio between the complexity of the scene being sampled and the speed of the host machine. If these two factors do not vary too frequently, which seems a reasonable assumption, then the real time needed to consume any step in virtual time will be relatively constant. Hence, the number of virtual units to consume in the next step can be dynamically chosen by the sampler, so as to fit the estimated real time needed by the system to perform the update.

This last proposition is particularly interesting. It guarantees an execution of the scripts that is kept close to real time, whatever the complexity of the main script may be, and whatever computing power is available. The only thing that changes is the granularity of the updates: in other words, the system will always do the same thing, but if it has more time it will do it *better*, by displaying a smoother motion in a computer animation, or by giving more detailed results for a computer monitoring system. So scripts can be used to design systems capable of graceful degradation.

Future research on this area can take several directions:

- Add operators to define only *partial order* on activities, so that activity expressions only define *temporal constraints* on the ordering of temporal events. Considerable work has been done in the area of temporal logic [Allen83]; we hope to be able to use these results, not only for temporal reasoning, but also for the specification of temporal applications.
- Design an interactive graphical editor for temporal expressions. Most of the temporal concepts, like duration, parallelism, simultaneity, could be represented and manipulated graphically, in an easier way than writing down temporal expressions.
- Work on the difficult problem of object interaction. Our current system is still inadequate to model worlds where parallel agents constantly exchange information. A number of applications, however, need that kind of system, for instance, programs for simulating physical particles, or dynamic animations as they are pictured in [Rey87].
- Implement multimedia objects that share a common virtual time.
- Investigate the real-time applications in which our idea of an adaptive sampler would prove particularly useful.

Acknowledgements
We would like to thank Xavier Pintado and Gerti Kappel for their useful comments on an earlier draft of this paper.

References

[Allen83] James F. Allen, Maintaining Knowledge about Temporal Intervals, CACM, Vol. 26 No 11, p. 832 ss, 1983.

[Andr83] Gregory R. Andrews and Fred B. Schneider, Concepts and Notations for Concurrent Programming, ACM Computing Surveys, Vol. 15 No 1, 1983.

[Bez87] Jean Bézivin, Some Experiments in Object-Oriented Simulation, OOPSLA'87 Conference Proceedings, Special Issue of SIGPLAN Notices, Vol 22 No 12, December 1987.

[Coin87] Pierre Cointe, Jean-Pierre Briot, Bernard Serpette, "The Formes System: a Musical Application of Object-Oriented Concurrent Programming", in *Object-Oriented Concurrent Programming*, ed. by A. Yonezawa and M. Tokoro, MIT Press, 1987.

[Fium87] Eugene Fiume, Dennis Tsichritzis and Laurent Dami, "A Temporal Scripting Language for Object-Oriented Animation", in *Proceedings of Eurographics 1987*, Elsevier Science Publishers (North-Holland), Amsterdam.

[Hoare74] C.A.R. Hoare, "Monitors: an Operating System Structuring Concept", CACM, Vol. 17, No 10, pp 549-557, Oct 1974.

[Jeff85] David R. Jefferson, "Virtual Time", in *ACM TOPLAS*, Vol 7. No 3, July 1985.

[Mag85] Nadia Magnenat-Thalmann and Daniel Thalmann, "Subactor Data Types as Hierarchical Procedural Models for Computer Animation", in *Proceedings of Eurographics 1985*, Elsevier Science Publishers (North-Holland), Amsterdam.

[Misra86] Jayadev Misra, "Distributed Discrete-Event Simulation", in *ACM Computing Surveys*, Vol 18 No 1, March 1986.

[Nier86] Oscar M. Nierstrasz, "What is the "Object" in Object-oriented Programming?", in *Proceedings of the CERN School of Computing,*, Renesse, The Netherlands, 1986.

[Rey82] Craig W. Reynolds, "Computer Animation with scripts and actors", in *ACM SIGGRAPH 1982 Conference Proceedings* (July 1982)

[Rey87] Craig W. Reynolds, "Flocks, Herds, and Schools: A Distributed Behavioral Model", in *ACM Computer Graphics*, Vol. 21, No 4, July 1987.

[Rog76] David F. Rogers and J. Alan Adams, Mathematical elements for Computer Graphics, McGraw-Hill, 1976.

[Shi87] Etsuya Shibayama and Akinori Yonezawa, "Distributed Computing in ABCL/1", in *Object-Oriented Concurrent Programming*, ed. by A. Yonezawa and M. Tokoro, MIT Press, 1987.

[Strous86] Bjarne Stroustrup, "The C++ Programming Language", Addison-Wesley, Reading, Massachusetts, 1986.

[Weg87] Peter Wegner, "Dimensions of Object-Based Language Design", OOPSLA'87 Conference Proceedings, Special Issue of SIGPLAN Notices, Vol 22 No 12, December 1987.