A Functional Approach to Real-Time Control*

Roger B. Dannenberg and
Paul McAvinney
Computer Science Department, Carnegie-Mellon University
Pittsburgh, PA 15213

Abstract

Traditional procedural programming languages make real-time programming difficult because the programmer must prevent the procedural nature of his programs from interfering with the real-time response that he wants to implement. To solve this problem, we propose a functional programming language that allows the programmer to "stand outside" the time domain, thus avoiding confusion between the sequential nature of program execution and the time-varying nature of program output. An abstract model for real-time control is presented which is based on *prototypes* and *instances*. A *prototype* is a higher-order function that maps a starting time and duration scale factor into a function of time, called an *instance*. Operations are provided to manipulate and combine prototypes to describe complex responses to system inputs. Using the model as a basis, a language for real-time control, named Arctic, has been designed. The objective of Arctic is to lower the conceptual barrier between the desired system response and the representation of that response.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-81-K-1539.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

1. Introduction

In the past, real-time control has been achieved more through ad-hoc techniques than through a formal theory. Languages for real-time control have emphasized concurrency, access to hardware I/O devices, interrupts, and mechanisms for scheduling tasks rather than taking a high-level problem-oriented approach in which implementation details are hidden. In this paper, we present an alternative approach to real-time control that enables the programmer to express the real-time response of a system in a *declarative* fashion rather than an *imperative* or *procedural* one.

Examples of traditional, sequential languages for real-time control include Modula [13, 15, 14], Ada [6, 3], OWL [7], and OCCAM [10]. These languages all provide support for concurrency through multiple sequential threads of control. The programmer must work hard to make sure his processes execute the right instructions at the right times, and real-time control is regarded as the most difficult form of programming [8]. In contrast, our approach is based on a non-sequential model in which behavior in the time domain is specified explicitly. This model describes possible system responses to real-time conditions and provides a means for manipulating and composing responses. We have defined and implemented a programming language [5] based on the model in order to describe and explore the model through examples. It should be emphasized that our efforts have concentrated on the development of a notation for specifying desired real-time behavior. Any implementation will only approximate the desired behavior, just as computer implementations can only approximate arithmetic on real numbers. We have not addressed the problem of specifying or meeting maximum latency requirements or minimum frequency

*Published as: Roger B. Dannenberg and Paul McAvinney, "A Functional Approach to Real-Time Control," in *Proceedings of the International Computer Music Conference,* Paris, France, October 19-23, 1984. San Francisco, CA: Computer Music Association, June 1985. pp. 5-16.

response; however, our current work is focussed on reimplementing our language to achieve real-time performance capabilities for music applications.

2. The Model

Our model is based on the idea that real-time systems can be described in terms of responses to events, and that the appropriate response may involve a complex behavior that is extended over some interval of time. The response may even be affected by events that occur as the response is in progress. We use higher-order functions[1] called *prototypes* to represent a set of appropriate responses to a type of event. A prototype takes an argument called the *starting time*, which is the real time of the event, and usually determines when the response should begin. The result of applying a prototype to a starting time is a function of time, called an *instance*, representing the response to the event.

Prototypes have two other arguments, called the *duration factor* and *terminate*, on which instances may also depend. The duration factor usually affects the overall duration of the response, and terminate is a time at which a response should be discontinued due to the occurence of an asynchronous event. In some cases, it is convenient to violate these suggested interpretations of a prototype's arguments; therefore, prototypes are not required to obey these conventions.

At this point, the reader may wonder why we have included higher-order functions in our model, when simple functions of time are perfectly good models for envelopes, audio signals, and control inputs. And furthermore, one can perform simple transformations on control functions to shift them in time or to stretch them. The reason for higher-order functions is that they give us the ability to model responses at higher levels of abstraction than the level of audio signals or even control functions. Consider this example: If one were to ask a performer to make a note longer, it is likely that he would increase the duration of the note, but leave the pitch unaltered. One can model this note concept with a prototype such that increasing the duration factor results in a longer instance, but not a lower frequency. On the other hand, if we were to simply stretch a function of time, the resulting function would exhibit lower pitch along with its increased duration.

Figure 2-1 illustrates this concept. If we "stretch" the note by increasing the duration factor argument of the note prototype, then the instantiation will have the desired properties. On the other hand, if we instantiate the note prototype immediately to obtain a function of time, then stretching the function will not produce the desired result, as illustrated at the bottom of the figure. The essential ingredient of the model is its ability to model abstract notions, and to allow the manipulation of these abstractions. Abstractions can then be "instantiated" to produce the control functions or audio signals that realize or implement the desired abstraction.

Let us consider another example: suppose we would like to describe a set of amplitude envelopes with starting times determined by one parameter, and decay times determined by another. The attack time, however, should always be 0.01 seconds. This set of amplitude envelopes could be modeled by a prototype, where the starting time and duration factor of the prototype establish the starting time and decay time of each envelope in the set. Thus,

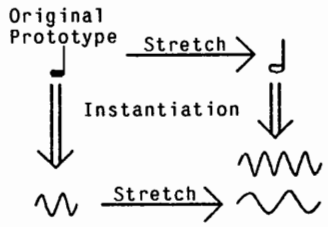

Figure 2-1: Illustration that the stretching of a prototype is not necessarily equivalent to the stretching of an instance.

the prototype represents or can be used to generate an infinite set of envelopes, and each envelope in the set is a particular *instance* of the prototype. Figure 2-2 illustrates several instances of this envelope prototype.

In general, the instances generated by a prototype can be arbitrary, although it is convenient for the parameters to correspond in a reasonable way to the concepts of "when" and "how long". As a third example, a prototype can represent a musical phrase: suppose we want to model a sustained tone preceded by a grace note of constant duration. Notice that if we simply took a representation of two notes and scaled time uniformly,

[1] A higher-order function is a function whose value is itself a function.

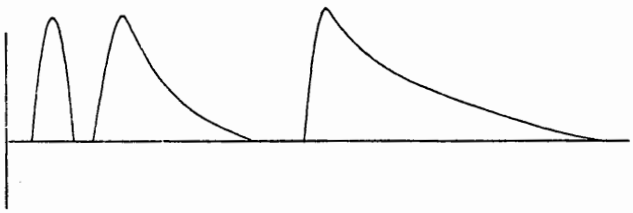

Figure 2-2: A family of envelope contours generated by a prototype.

then the grace note would lengthen along with the other note. (The pitch might drop as well!) In contrast, prototypes allow us to describe the desired response precisely, without necessarily writing a separate pair of notes for each combination of starting time and duration factor. Thus, we can express a multitude of individual responses using a single general description. This example illustrates again the importance of being able to manipulate response descriptions at the appropriate level(s) of abstraction.

In the language Arctic, the starting time and duration factor will be implicit parameters, but for now, we will notate an instance of prototype P by $P(s, d)$, where s is the starting time, and d is the duration factor. For the mathematically inclined, an instance is a function from an interval of time to the reals:

$$f: \mathbb{R} \rightarrow \mathbb{R},$$

and a prototype is a higher order function from starting time (a real) and duration factor (a real) to an instance:

$$P: \mathbb{R} \times \mathbb{R} \rightarrow (f: \mathbb{R} \rightarrow \mathbb{R}).$$

Corresponding to every prototype P is a function

$$P_{stop}: \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$$

representing the time at which $P(s, d)$ "finishes". By convention, instances of prototypes are defined on the interval $(s, P_{stop}(s, d)]$. P_{stop} makes it possible to write expressions representing sequential responses, as we shall see in Section 2.3.

2.1. Shift and Scale

Standard transformations allow prototypes to be transformed by shifting the time argument, or by scaling the duration factor, resulting in a new prototype.

A prototype can be shifted in time using the *shift* (@) notation. $P @ x$ is a prototype defined by:

$$(P @ x)(s, d) = P(s + xd, d)$$

Informally, the expression $P@x$ means "apply P at x". Notice that the delay x is scaled by the implicit duration factor. This scaling may seem a little confusing, but it often simplifies nested expressions, as we shall see.

The duration factor can be scaled using the *scale* (~) notation. $P \sim x$ is a prototype defined by:

$$(P \sim x)(s, d) = P(s, xd)$$

Thus, $P \sim x$ means "an instance of P with duration scaled by x". Notice that $P @ 0 = P \sim 1 = P$.

Shift and scale expressions can be nested; let us look at a few examples. The first example illustrates a nested scale expression:

$$((P \sim x) \sim y)(s, 1) =$$
$$(P \sim x)(s, y) =$$
$$P(s, xy).$$

The effect is to multiply the nested duration factors. The next example shows a shift expression within a scale expression:

$$((P @ 3) \sim y)(s, d) =$$
$$(P @ 3)(s, yd) =$$
$$P(s + 3yd, yd).$$

Notice how the duration factor yd scales the shift amount from 3 to $3yd$.

2.2. Other Operations and Primitives

In addition to the scale and delay operations, conventional operations including addition and multiplication of instances are defined. For example, $3P$ is an instance of prototype P multiplied by 3, and $P+Q$ is an instance of P added to an instance of Q. We note that the ability to operate on instances is an important part of the model, but we will not dwell on the definitions of operations in this paper.

Where do prototypes come from? We take prototypes to be primitives in our model, but we would like to suggest that prototypes can take several forms. First, a prototype can represent a transformation of a function. The function can be described or computed by a programmer, or it can be a recording of a function in the real world, such as a loudness contour of a musical instrument, or the shape of a city skyline.

Further parameterization is a means of achieving

families of prototypes. For example, consider the set of prototypes whose instances are of the form $f(t) = sin(w(t-s))$ over the range $(s, s+d]$. We can imagine a prototype called *sin* that takes a parameter (w) in addition to the usual starting time and duration.[2] . A number of useful parameterized prototypes can be imagined, indicating that time and duration alone are not generally sufficient as a parameter space for prototypes. However, this does not necessarily indicate a problem in the model. Making special cases out of time and duration allows us to adopt a convenient notation in which these parameters are often implicit. This makes sense because they are almost universal among interesting prototypes. Other parameters, like frequency, color, and texture are less universal and are more conveniently handled as explicit parameters.

2.3. Collections and Sequences

Up to this point, we have only considered prototypes that yield instances that are real-valued functions. For several reasons, we must extend the concept of prototype. First, a response to an event may consist of a number of parallel activities; we need some representation for concurrency in our model. We will fill this need using the concept of *collections* described below. Secondly, we would like to be able to express a complex response to an event in terms of simpler responses. This leads us to consider hierarchical descriptions in which a complex prototype is defined in terms of other prototypes. In Section 3, we will describe the programming language Arctic, which is based on the model and allows hierarchical descriptions of prototypes.

Collections are an extension of the concepts of prototypes and instances. A *collection prototype* is a higher-order function with the same domain as a prototype (time and duration factor), but whose range is a set of instances, called a *collection instance.* A collection prototype (hereafter referred to simply as a collection) can be thought of as a set of prototypes. An instance of the collection is equivalent to the set of instances of the component prototypes, all with the same arguments. We indicate collections by square brackets enclosing a list of prototype expressions. For example, the following collection C consists of three prototypes:

$$C = [P \sim x; P @ 5; Q].$$

An instance $C(s, d)$ is the set containing $P(s, dx)$, $P(s + 5d, d)$, and $Q(s, d)$.

Just as every simple prototype P has a corresponding stop time P_{stop}, a collection C may indicate a stop time C_{stop}. The duration is indicated by including "**end @** e" in the list of expressions, where e is a constant, and C_{stop} is defined to be:

$$C_{stop}(s, d) = s + de,$$

that is, the sum of the start time s and the product of the duration factor d and the indicated constant e. For example, if $C = [Q; P @ 2;$ **end** $@ 5]$, then $C_{stop}(0, 2) = 10$. Think of C as starting at time 0 and ending at time 5, relative to the implicit starting time (0), and multiplied by the duration factor (2).

A sequence is a special form of collection that models a sequence of events. We will denote a sequence by using "|" as a separator in place of the semicolon used for collections. For example:

$$[P \sim x \mid P @ 5 \mid Q]$$

is a sequence. A sequence differs from a collection only in the way instantiation is defined. In a collection, the components are all applied to the same starting time, but in a sequence, the components are applied to the stop time of the previous component. A sequence can always be rewritten as a collection; for example:

$$[P \mid Q \mid R \mid \ldots Y \mid Z] =$$
$$[P; Q @ P_{stop}; R @ Q_{stop}; \ldots ; Z @ Y_{stop}].$$

The stop time of a sequence is that of the last element of the sequence. In the previous example, the stop time of the sequence is Z_{stop}.

Together with shift and scale, collections have some nice algebraic properties, some of which are shown below:

Distributive Laws:

$$[P ; Q] @ t = [P @ t ; Q @ t]$$
$$[P ; Q] \sim d = [P \sim d ; Q \sim d]$$
$$[P \mid Q] \sim d = [P \sim d \mid Q \sim d]$$

Associative Laws:

$$[[P ; Q] ; R] = [P ; [Q ; R]]$$
$$[[P \mid Q] \mid R] = [P \mid [Q \mid R]]$$

[2] Alternatively, we could introduce higher ordered functions (meta-prototypes?) whose ranges are prototypes. In this example, *sin* would be a meta-prototype, $sin(w)$ would be a prototype, $sin(w)(s, d)$ an instance, and $sin(w)(s, d)(t)$ the value of the instance at time t.

Commutative Laws:

$$[P\,;Q] = [Q\,;P]$$
$$(P \mathbin{@} x) \mathbin{@} y = (P \mathbin{@} y) \mathbin{@} x$$
$$(P \sim x) \sim y = (P \sim y) \sim x$$

Identity Laws:

$$P \mathbin{@} 0 = P$$
$$P \sim 1 = P$$

2.4. Discussion of the Model

The model can be used to specify the responses of a real-time system to events which may be either synchronous or asynchronous, and either continuous or discrete. We consider a real-time system to have two forms of input or output: continuous and discrete. A continuous input or output is a time-varying value, which corresponds to an instance in the model. For example, if the instance *X* represents a continuous input, then we can write the expression $(aX + b)$ which depends on input *X*.

A discrete input or output is an event that carries no information except for its name and its time of occurrence. Events are modeled by the instantiation of a prototype, where the time of the instance is the physical time of the corresponding event, and the implicit duration factor is (arbitrarily) taken to be equal to 1.

The total input to a system can be written as a collection. For example, suppose we have a system that is controlled by a pointing device and two buttons labeled *A* and *B*. The pointing device generates two functions, *X* and *Y*, which correspond to the position of the pointer at any point in time, and the buttons are associated with prototypes named *A* and *B*. Furthermore, suppose that button *A* is pressed at times 2 and 5 and that button *B* is pressed at time 6. Then a collection describing this input is $[X;\ Y;\ A \mathbin{@} 2;\ A \mathbin{@} 5;\ B \mathbin{@} 6]$. In Section 3, we will describe a language that allows the specification of the system response to such an input.

The model borrows from the GROOVE system [9], in that behavior is represented as the combination of functions of time, and new behaviors can be created by combining and modifying functions. The model is also indebted to the 4CED program [1], in which actions can be triggered by events, and in which timing is notated explicitly. In the model, however, there is no notion of sequential execution or state. Instead, the time at which a function takes on a given value, or the time at which some function starts is specified through expressions that give complete timing information; thus, there is no need for imperative-style commands that must be executed in sequence. Assignment in the traditional imperative language sense is not defined because assignment implies the existence of variables that can take on different values at different points in the execution of a program. In contrast, the model uses functions to denote time-varying responses, and our resulting language will be of the applicative, or functional style [2] as opposed to the imperative style. The language Formes [4, 12] is intended to solve the same problems of real-time control that we address. In Formes, time-varying values are computed by rules that are associated with objects. These rules are invoked by a "monitor" program that walks a "calculation tree" consisting of the currently active objects. The data-structure is walked at intervals corresponding to the time resolution of the system. Formes capitalizes on the abstraction facilities of its underlying object-oriented programming language to hide many of the details of the computation of functions. Interestingly, this allows Formes programs to take on a declarative style.

While it may seem strange to abandon the more sequential programming style for real-time control, there are some good reasons for doing so. In a language in which statements are executed in sequence, we are forced to use the time domain to describe program behavior. The programmer must take great pains to prevent sequential execution from interfering with the real-time behavior of his programs. The sequential nature of procedural program execution forces the programmer to pay attention to many implementation details that are irrelevant to his overall task. On the other hand, our model allows us to describe real-time behavior using a language whose meaning can be understood outside of the time domain. By "stepping-out" of the time domain, behavior can be described in a more problem-oriented fashion, but we must give up the notions of sequence, assignment, and variables: these only make sense within the time domain.

The model gives us powerful concepts of prototypes, instances, and collections, but the model does not deal with the problems of naming objects and building hierarchical descriptions (programs). Using the model as our basis, we now proceed to construct a language for

describing real-time systems. The language allows the definition of prototypes and collections in a structured, hierarchical fashion. The language is called Arctic. It is "poles" apart from previous process-oriented languages.

3. Arctic: An Applicative Language for Real-Time Control

Arctic is a language for describing systems whose inputs and outputs may be time-dependent and asynchronous. The model presented above forms the basis for Arctic, which includes facilities for combining and naming objects in the model. Arctic is an *untyped* language like Lisp or APL, but its syntax has been influenced strongly by Pascal. This is strictly a matter of taste, and a Lisp-like or even a music-oriented syntax can be imagined.

The current version of Arctic is designed as an aid to evaluating the model and the concept of applicative languages for real-time control. Consequently, the language specification omits a number of "features" that might be desirable in a practical implementation, since "features" often distract one's attention from the central issues. The design of Arctic is far from frozen! In order to reduce the length of this presentation, we will concentrate on the interesting aspects of Arctic and not attempt to document a complete definition.

3.1. Preliminaries

An Arctic program specifies a system's response to a set of real-time inputs. This is accomplished by defining prototypes and collections, some of which are instantiated by external, real-time events.

An Arctic program is literally a string of tokens: spaces and new lines serve as separators, but have no other significance. As in other programming languages, but unlike our notation for the model, a token can be composed of multiple characters, and multiplication is indicated explicitly by the symbol "*". For example, *XY* is an identifier, and *X* * *Y* represents the product of two values.

In this paper, we will use *italics* for Arctic identifiers and **boldface** for language keywords.

3.2. Prototypes

A prototype is indicated by its corresponding identifier or by a prototype expression. The following are examples of prototypes:

```
Contour
Contour @ Delay
Contour ~ 2 @ Delay
```

Recall that an instance is an application of a prototype to a particular time and duration factor. In Arctic, this application can only occur as the direct or indirect result of an external stimulus. The direct case occurs when a programmer defines a prototype that corresponds to an external event. The prototype is instantiated when the event occurs. The indirect case is the result of a prototype that is instantiated by another. For example, suppose there is an external event called *Key*, and the programmer writes the following definition:

```
Key causes [A | B].
```

The event *Key* results in the direct instantiation of the prototype *Key*, which in turn instantiates prototypes *A* and *B*. The particular names associated with various physical events are system-dependent, but every system includes the event *Go*, which occurs when the real-time system is started.

3.3. Variables

It is often convenient to refer to the same instance at several points in a program. A *variable* is a reference to a particular instance. Variables are defined using *assignment* (indicated by ":="), and a variable may only be assigned once. Thus, variables only vary in the sense that they refer to time-varying functions; once defined, their meaning is constant because no reassignment can occur. (Perhaps *definition* is a more appropriate term than assignment in the light of this single assignment rule. We chose to use the term "assignment" for historical reasons [11].) In addition to the use of variables to refer to internally generated instances, variables are also used to refer to external, continuous inputs and outputs. As a simple example, an Arctic program to implement a linear amplifier with gain control is:

```
Go causes [output := input * gain];
```

The variables *input* and *gain* are system inputs, and the variable *output* is the system output.

3.3.1. Sums and Products

A variable may also be defined as a sum or product of expressions. The following assignments:

```
a += b; a += c;
x *= y; x *= z;
```

are equivalent to:

```
a := b + c;
x := y * z;
```

However, the first form is useful when the number of addends or factors depends upon the number of instances at a given time. (We will illustrate this with examples below.) Notice that $a += b$ is not equivalent to $a := a + b$, which would violate our single assignment rule, and would also present us with a recursive definition of *a*.

3.3.2. Declarations

Variables must be declared as ordinary (**value**), sums (**sum**), or products (**prod**). In addition, the keywords **input** and **output** may precede one of the three variable keywords to indicate that the variable corresponds to a system input or output. Here are some example variable declarations:

```
input value amplitude, frequency;
output sum SignalOut;
prod Gain;
```

3.4. Prototype Declaration

The language provides a certain number of built-in, or predefined prototypes which can be used within any program. In addition to these predefined prototypes, new prototypes can be created by combining other prototypes in a prototype declaration. We will describe prototype declaration by example.

```
(1) P(in X; out Y) causes
(2)     [value Z;
(3)      Z := X * 2;
(4)      Q(Z) @ 3;
(5)      Y := Z + (sin(Z) ~ 15);
(6)      end @ 15 ];
```

In line 1, the heading names the prototype (*P*) and specifies a few parameters. The forms of parameter specifications and their meanings are listed below (the symbol χ stands for a parameter identifier):

in χ — The effect is to declare a local variable χ and assign to it the value of the actual parameter. Thus, an **in** parameter is one that is passed *to* an instantiation.

out χ — The effect is to declare a local variable χ. The value of χ is assigned to the actual parameter; hence, there must be an assignment to χ within the body of the prototype. Thus, an **out** parameter is one that is returned *from* an instantiation.

sum χ — The actual parameter must be a **sum** variable. The identifier χ becomes a local name for the actual parameter.

prod χ — Similar to **sum** χ, for product (**prod**) variables.

On line 2 begins the body of *P*, which is a collection. The local variable *Z* is declared. Line 3 defines *Z* with an assignment (:=). Line 4 is a prototype specification which says apply *Q* (which we assume is defined elsewhere) with parameter *Z* after a delay of 3. In line 5, an assignment is made to *Y* of an instance that is the sum of *Z* and *sin*(*Z*). This line illustrates a predefined, parameterized prototype *sin*. This prototype yields a sine function with unit amplitude at the specified frequency, *Z*. The sine function starts at the implicit starting time, and its duration (but not frequency) is scaled by the duration factor, 15. Finally, in line 6, the reference duration of *P* is defined to be *15*.

3.5. Parameter Passing

The semantics of parameter passing deserves some further explanation. In particular, we wish to illustrate what happens when we pass an instance as a parameter. We will also illustrate the use of variables as references to instances.

The general rule for parameter passing is: *Parameters to prototypes are always instances or scalars, and are not themselves affected by any shifting or scaling that may be applied to the prototype.* Of course, instances are often the result of applications of prototypes, which *are* subject to both shifting and scaling. To illustrate this rule, let *Ramp* be the prototype illustrated at the top of Figure 3-1, and described by:

$$r(t) = (t - s)/d, \text{ where}$$
$$r = Ramp(s, d).$$

And let *Unit* be a prototype illustrated at the bottom of 3-1, and described by:

$$u(t) = 1 \text{ for } s < t < s+d, \text{ and}$$
$$u(t) = 0 \text{ otherwise, where}$$
$$u = Unit(s, d).$$

Notice that *Unit* can be used to "gate" an interval of a

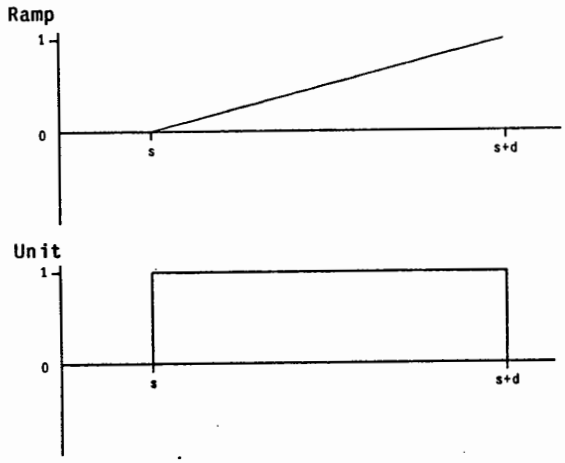

Figure 3-1: The *Ramp* and *Unit* prototypes.

function; for example, Figure 3-2 illustrates the two instances:

*(Ramp ~ 3) * (Unit @ 0)*, and
*(Ramp ~ 3) * (Unit @ 1)*.

In the figure, *t* is the time of instantiation, *3* is the scale factor applied to *Ramp*, and *0* and *1* are shifts applied to *Unit* (relative to *t*).

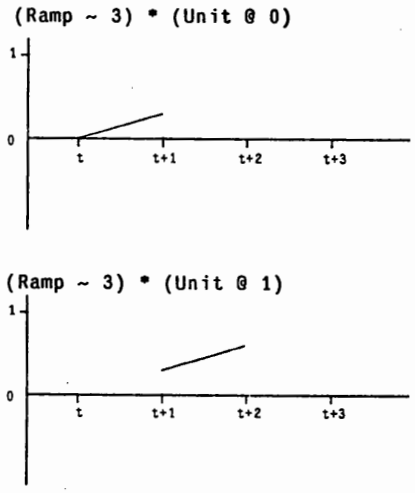

Figure 3-2:
*(Ramp ~ 3) * (Unit @ 0)*,
and *(Ramp ~ 3) * (Unit @ 1)*.

Now, let *Gate* be defined as follows:

```
Gate(in f) causes
    [output += f * Unit;
     end @ 1];
```

This prototype causes an interval of the parameter *f* to be added to *output*. The interval is determined by the implicit starting time and duration of the *Unit* prototype, and these are inherited from the instantiation of Gate. Consider the following program:

```
[value r;
 r := Ramp ~ 3;
 Gate(r) @ 0]
```

This is equivalent to:

```
output += (Ramp ~ 3) * (Unit @ 0);
```

Similarly, the following two programs are equivalent:

```
[value r;
 r = Ramp ~ 3;
 Gate(r) @ 1]
```

```
output += (Ramp ~ 3) * (Unit @ 1);
```

Notice that the value of *r* is bound to *f* (the formal parameter of *Gate*), and this value is independent of the starting time of *Gate*. Finally, we present one more variation, illustrating the importance of our use of the variable *r* in the previous examples:

```
Gate(Ramp ~ 3) @ 1
```

In this example, the instantiation of *Ramp* is within the scope of the shift operation, so the instance of *Ramp* passed to *Gate* is defined on the interval *(t + 1, t + 4)*. The output is illustrated in Figure 3-3.

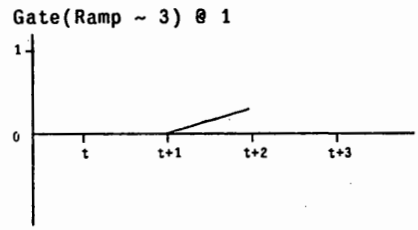

Figure 3-3: Output produced by: *Gate(Ramp ~ 3) @ 1*

3.6. A Simple Application

We are now ready to write a simple real-time system. To illustrate the power of Arctic, we will define a computer-music instrument all the way down to the signal-processing level. (In a more likely scenario, an Arctic program would control special-purpose hardware for music synthesis.) Let us first define a prototype *FMI* that implements a simple FM instrument:

```
FMI(in freq) is
    sin(freq + sin(freq*2) * MI) * AmpEnv;
```

The prototypes *MI* and *AmpEnv* are prototypes for the modulation index and amplitude envelopes, and are not defined here. The prototype *sin(f)* generates a sine function with frequency *f* starting at the implicit starting

time and with the implicit duration. Using *FMI*, we can define a routine that plays a note:

play(**in** *pitch*) **causes**
  [*output* += *FMI*(*PitchToFreq*(*pitch*)) ~ *2*];

When *play* is instantiated, it generates a note and adds it to *output*. The function *PitchToFreq* is used to translate a pitch number to frequency and is not defined in this example. The duration of the note is (arbitrarily) scaled to 2 seconds. If we connect *output* appropriately to a loudspeaker, and instantiate *play(20)*, we should hear a tone with pitch corresponding to pitch number 20. Let us extend our program so that we can perform from a keyboard. We assume that we have a keyboard interfaced to our Arctic system and that pressing a key instantiates the prototype *KeyDown*(*i*), where *i* is the number of the key. The prototype *KeyDown* is defined as follows:

KeyDown(**in** *i*) **causes** [*play*(*i* + *offset*)];

The operation of the program is as follows: whenever a key is pressed, an instance of *KeyDown* is created, which in turn instantiates play. The value *offset* is added to *i* to effect a transposition. Play then creates an instance of *FMI* and adds the resulting signal to *output*, which causes an audio output. If two keys are pressed at once, we get two instances of each of our prototypes, and two independent and concurrent signals are added to the output. The reader may have noticed that there is no way in this simple example to stop a note once it has begun. The problem is that information can only be passed to an instance in one of two ways: by passing parameters at the point of instantiation, or by assignments to global variables that are used by the prototype. There is no convenient way to say "hold this note until the key is released". In the next section, we present new language mechanisms that allow us to express this sort of response to asynchronous events.

4. Asynchronous Events

It must be possible to define prototypes whose responses are altered by discrete events that occur after instantiation. For this purpose, we add a new language construct:

P **until** *C* **then** *Q*

where *P* is a prototype, *C* is a *condition*, and *Q* is another prototype. The effect of this prototype is as follows. First, *P* is instantiated at the implicit starting time. If *C* becomes true before the duration of this instance of *P* has elapsed, then the instance of *P* and all instances created directly or indirectly by *P* are terminated. At this time, *Q* is instantiated. We will now define what we mean by a *condition* and *termination*.

4.1. Conditions

A condition is a time-varying boolean expression. For example, if *X*, *Y*, *Selector*, and *R* are functions of time (they may be instances or variables), then the following are conditions:

(*X* > *Y*) **and** (*Y* >= *0*)
(*Selector* = *STOP*) **or** (*R* > *100*)

A condition may also include terms that are syntactically similar to a prototype heading; for example:

event *Panic*()
event *KeyUp*(**in** *i*) **and** (*i* = *23*)

These terms are true at the instant of time when the prototype is instantiated; if the prototype has **in** parameters, then they are passed to the condition expression as if the expression were the definition of the prototype. The keyword **event** indicates that we are waiting for an event to cause an instance of the named prototype. (If **event** is omitted, the expression would denote the value of a new instance of the prototype as usual.)

4.2. Termination

To terminate an instance at time *t*, we want not only to ignore the value of the instance after time *t*, but also to disable its effect on **sum** and **prod** variables either directly or indirectly. To describe this formally, we add a third implicit parameter, called *terminate*, to each prototype, and define all Arctic primitives (the building blocks of Arctic programs), such as *unit* and *ramp*, to terminate their response at the time indicated by *terminate* whenever it is less than the normal stop time. The terminate parameter is inherited just like the starting time and duration factor, and it can be modified only by **until** expressions. Otherwise, the termination time is inherited by all subexpressions of an expression, giving termination the desired properties.

4.3. Example

Let us modify our previous example to turn off notes quickly when a key is released. First, *FMI* is modified to take an amplitude contour as a parameter:

FMI(**in** *freq*, *env*) **is**
  sin(*freq* + *sin*(*freq***2*) * *MI*) * *env*;

To accomplish our goal, we will generate an amplitude contour that has an initial part while the key is down, and

a final part that commences when the key is released. The overall contour must be continuous at the transition from the initial to the final parts. We assume a suitable final envelope prototype called *AmpFinal*, and all instances of *AmpFinal* have an initial value of 1.

The following program plays a note until the release of the key indicated by *KeyNumber*:

play(**in** *pitch*, *KeyNumber*) **causes**
  [**value** *E*;
  E := *AmpEnv* **until**
    (**event** *KeyUp*(**in** *i*) **and** *i* = *KeyNumber*)
      then *AmpFinal* * *E*(**start**);
  output += *FMI*(*PitchToFreq*(*pitch*), *E*) ~ 2

The variable *E* gets the amplitude envelope from *AmpEnv* until the event *KeyUp*(*KeyNumber*). At that time, *E* is terminated and *AmpFinal* * *E*(**start**) is instantiated. In this prototype expression, *E* is a function of time (an Arctic variable), so *E*(**start**) is the value of *E* at the time **start**; **start** is a keyword which always denotes the implicit starting time of the current prototype. In this case, **start** is the point in time at which *AmpEnv* is terminated and at which *AmpFinal* is instantiated. This insures that *E* is a continuous function. (Recall that the initial value of an instance of *AmpFinal* is 1.) The envelope *E* is passed to *FMI* and the resulting signal is added to *output*. The instance of *FMI* is scaled somewhat arbitrarily to 20. The precise response of FMI and the duration of the note will depend upon the prototypes *AmpEnv*, *MI*, and *AmpFinal*, as well as the timing of events *KeyDown* and *KeyUp*.

To complete our program, we must redefine *KeyDown* to pass the key number to *play*:

KeyDown(**in** *i*) **causes** [*play*(*i* + *offset*, *i*)];

We have now written a program that implements a polyphonic keyboard instrument. Pressing a key will initiate a note with an amplitude contour governed by the prototype *AmpEnv*. Releasing a key will terminate the *AmpEnv* contour, and initiate an *AmpFinal* instance to complete the amplitude contour.

5. Implementation of Arctic

We have been careful to avoid implementation considerations in the design of Arctic; our goal has been to stress elegance and clarity without concern for practical matters. We expect the implementation of Arctic to be both challenging and illuminating. Ultimately, we expect our implementation research to lead us to the design of special-purpose architectures for highly parallel real-time control.

Implementation of a prototype Arctic system was completed in September 1983. This system did not run in real-time and used hand-compiled Arctic programs linked with a function-manipulation library to provide graphic and audio output. We have recently completed an interpreter for a large subset of Arctic. Again, the system does not run in real-time. Instead, it reads an Arctic program and a description of system inputs. The system outputs are then computed and written to a file, after which they may be displayed or used to control real-time or software synthesis. We are presently designing a real-time version of Arctic for the M68000 processor.

6. Summary

We have outlined a new approach to the problem of real-time control. Our approach is based on a powerful model in which the primitive elements are functions of time, and a simple notation is provided for manipulating and combining time-varying functions. Parts of the model correspond directly to continuous and discrete real-time inputs and outputs.

The model provides the semantic foundations for the programming language Arctic, which extends the model with declaration and naming facilities that allow the construction of modular programs. Arctic is an applicative language, which implies no sequential execution, no side effects, and inherent parallelism. We know of no other applicative language that can declaratively specify real-time system response or deal with asynchronous events.

The applicative nature of Arctic is crucial, however. In contrast to sequential programming languages, the *evaluation* of Arctic programs is not time-dependent. This makes it much simpler to manipulate values which *are* time-dependent.

7. Conclusions

Several aspects of Arctic require more thought and experiment. The use of starting times and durations is just a first approach to the more general problem of *temporal alignment*, and we would like to improve this aspect of the model. Second, we would also like to see a more elegant way to handle asynchronous inputs. At present, asynchronous inputs can terminate an instance

and start another, but it is awkward to pass information to the new one. Third, we have not dealt with arbitrary transformations in the time domain. This would provide, for example, a mechanism for implementing tempo variations. Finally, Arctic must be extended with data structures such as arrays and lists, in order to more easily perform arbitrary computations.

We feel that the problem of real-time control is largely a problem of language. To tackle the design and implementation of complex systems, we must have a powerful notation. By introducing such a notation, we feel that Arctic makes a significant contribution to the field of real-time control.

8. Acknowledgements

The authors wish to acknowledge the helpful comments and encouragement of Bill Scherlis, Dana Scott, and Curtis Abbott. Frank Wimberly's comments on an earlier version of this paper helped us to improve and clarify the material. Dean Rubine implemented the Arctic interpreter, enabling us to understand Arctic better and to correct some of our earlier misconceptions.

References

[1] Curtis Abbott.
The 4CED Program.
Computer Music Journal 5(1):13-33, Spring, 1981.

[2] John Backus.
Can Programming Be Liberated from the von Neumann Style?
Communications of the ACM 21(8):613-641, August, 1978.

[3] C. Roads.
Machine Tongues VI: Ada.
Compter Music Journal 3(4):6-8, Winter, 1980.

[4] P. Cointe and X. Rodet.
FORMES: An Object and Time Oriented System for Music Composition and Synthesis.
In *1984 ACM Symposium on LISP and Functional Programming*, pages 85-95. ACM, August, 1984.

[5] R. B. Dannenberg.
Arctic: A Functional Language for Real-Time Control.
In *1984 ACM Symposium on LISP and Functional Programming*, pages 96-103. ACM, August, 1984.

[6] *Reference Manual for the Ada Programming Language*
United States Department of Defense, 1980.

[7] Marc Donner.
The Design of OWL - A Language for Walking.
In *Sigplan Symposium on Prog. Lang. Issues In Software Systems.* Sigplan, 1983.

[8] Robert L. Glass.
Real-Time: The "Lost World" Of Software Debugging and Testing.
Communications of the ACM 23(5):264-271, May, 1980.

[9] Max V.Mathews and F. R. Moore.
A Program to Compose, Store, and Edit Functions of Time.
Communications of the ACM 13(12):715-721, December, 1970.

[10] David May.
OCCAM.
Sigplan Notices 18(4):69-79, April, 1983.

[11] James R. McGraw.
The VAL Language: Description and Analysis.
ACM Transactions on Programming Languages and Systems 4(1):44-82, January, 1982.

[12] X. Rodet and P. Cointe.
FORMES: Composition and Scheduling of Processes.
Computer Music Journal 8(3):32-50, Fall, 1984.

[13] N. Wirth.
Modula: A programming language for modular multiprogramming.
Software, Practice and Experience 7(1):3-35, 1977.

[14] N. Wirth.
Design and implementation of Modula.
Software, Practice and Experience 7(1):67-84, 1977.

[15] N. Wirth.
Programming in Modula-2.
Springer-Verlag, New York, 1982.