Roger B. Dannenberg, Paul McAvinney, and Dean Rubine
Computer Science Department and
Center for Art and Technology
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213 USA

Arctic: A Functional Language for Real-Time Systems

Introduction

In the past, real-time control via digital computer has been achieved more through ad hoc techniques than through a formal theory. Languages for real-time control have emphasized concurrency, access to hardware input/output (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 (Wirth 1977a; 1977b; 1982), Ada (DOD 1980), CSP (Hoare 1978), and OCCAM (May 1983). These languages all provide support for concurrency through multiple sequential threads of control. Programmers must work hard to make sure their processes execute the right instructions at the appropriate times, and real-time control is regarded as the most difficult form of programming (Glass 1980). In contrast, our approach (Dannenberg 1984; 1986) is based on a nonsequential 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. The programming language Arctic is based on the nonsequential model and was designed for use in real-time computer music programs. It should be emphasized that our efforts have concentrated on the development of a notation for specifying desired real-time behavior. Any implementation only approximates 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 response; however, our current work is focused on reimplementing our language to achieve real-time performance capabilities for music applications.

Computer Music Journal, Vol. 10, No. 4, Winter 1986,

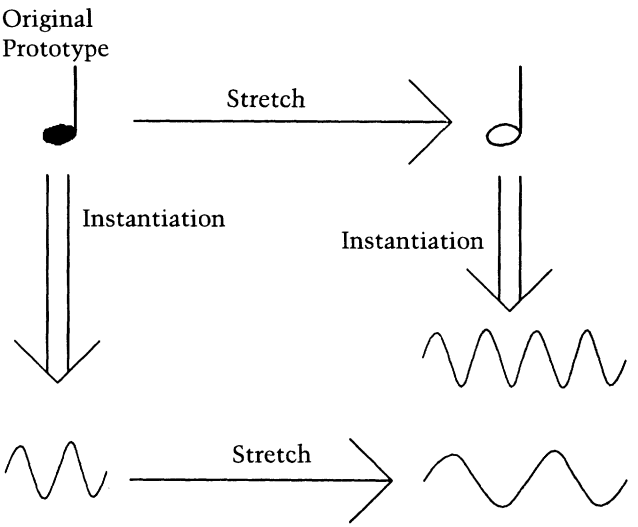
The Model

Our model is based on the idea that real-time systems can be described in terms of responses to events (discrete inputs) and functions (continuous inputs), and that the appropriate response may involve a complex behavior that is extended over an interval of time. The response may even be affected by events that occur as the response is in progress. We use higher-order functions called *prototypes* to represent a set of appropriate responses to a type of event. (A higher-order function is a function whose value is itself a function.) 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 at least two other arguments, called the *duration factor* and the *terminate,* on which instances may also depend. The duration factor usually affects the overall duration of the response, and the terminate is a time at which a response should be discontinued due to the occurrence 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

Fig. 1. Stretching a prototype is not necessarily equivalent to the stretching of an instance.



models for envelopes, audio signals, and control inputs. 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 the performer 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 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 sec. 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, 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.

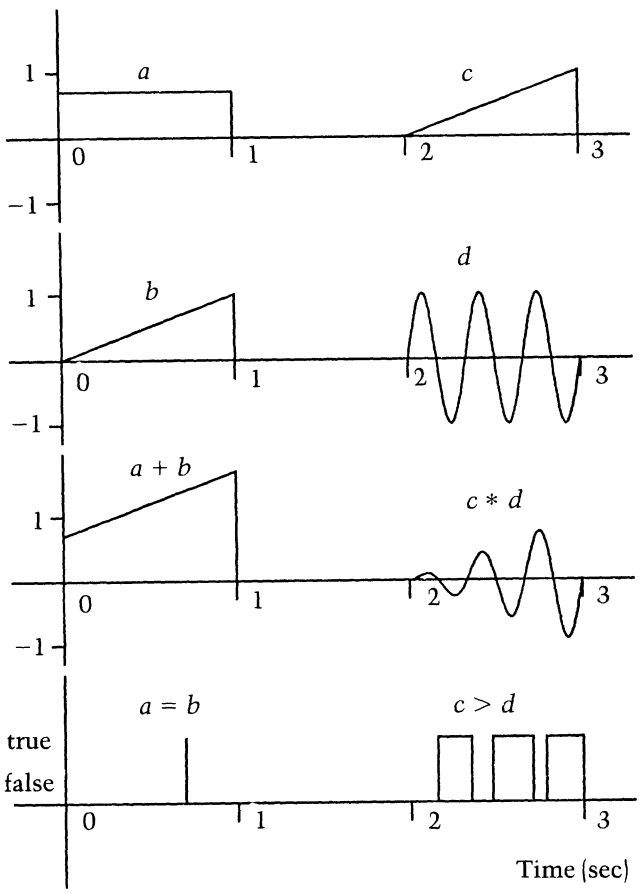
As a third example, a prototype can represent a musical phrase. Suppose we want to model a sustained tone preceded by a grace note of constant duration. Notice that if we simply took a representation of two notes and scaled time uniformly, then the grace note would lengthen along with the other note. (The pitch might drop as well!) In contrast, prototypes allow us to describe the desired response precisely without 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.

Notice that timing is explicitly specified when a prototype is instantiated. This is in contrast to more conventional languages in which timing is only implied by control structures and is usually obtained by explicit synchronization. We find that in most cases it is much easier to specify timing (when something should take place) than to specify synchronization (timing constraints among different processes or events). We will discuss the issue of synchronization further after describing the Arctic language.

A Description of Arctic

Arctic is a language for specifying systems whose inputs and outputs may be time-dependent and asynchronous. The model presented above forms the basis for Arctic. Artic also includes facilities for combining and naming prototypes and instances.

Fig. 2. Instances of primitive prototypes and operations on them.



Arctic is an *untyped* language like Lisp (McCarthy 1965) or APL (Iverson 1962), but its syntax has been influenced strongly by Pascal. The choice of syntax is strictly a matter of taste, and a Lisp-like or even a music-notation-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 not yet frozen! In order to reduce the length of this presentation, we will concentrate on the interesting aspects of Arctic and not attempt a complete definition. We describe the essential elements of the language through a series of examples, and the output generated by each example is illustrated graphically.

Primitive Prototypes and Operations

Arctic has a few primitive, or built-in prototypes. Figure 2 illustrates an instance (labeled *a*) of a *unit* prototype multiplied by 0.7. The *unit* prototype itself generates a function whose value is one on an interval defined by the start time and duration factor of the instance. The function is zero everywhere else. For example, in instance *a* in Fig. 2, the start time is zero and the duration factor is one. In the same figure, *b* and *c* are instances of the *ramp* prototype, and *d* is an instance of a *sin* (sine) prototype. Instances are just functions of time, and they can be combined using various mathematical operators. The third graph illustrates the values of the Arctic expressions $a + b$ and $c * d$ ("*" denotes multiplication). The fourth graph plots the values of the expressions $a = b$ and $c > d$. Notice that these last expressions denote Boolean functions of time.

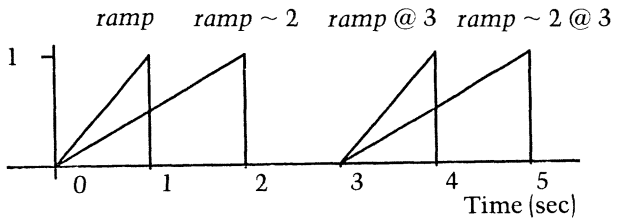
Any of these functions could be declared to be inputs or outputs. For example, *c* could be an instance of *ramp* as described above, *d* could be an input function, and $c * d$ could be the output of a real-time system.

Shift and Stretch

For convenience, a prototype instantiation inherits its starting time and duration factor from the expression in which it is embedded. However, operators are provided to change the inherited parameters wherever necessary. The *stretch* operator (~) is used to change the duration factor. Letting *P* be a prototype and *x* be a real number, instantiating $P \sim x$ with a duration factor *d* is equivalent to instantiating *P* with a duration factor *xd*. Informally, $P \sim x$ means "do behavior *P*, but make it last *x* times as long."

The *shift* operator (@) is used to offset the starting time. Instantiating $P @ x$ at time *t* and duration factor *d* is equivalent to instantiating *P* at time $t +$

Fig. 3. Several instances of ramp *showing the effect of* shift *(@) and* stretch *(~) operations.*



xd. Informally, *P* @ *x* means "do behavior *P*, but start it *xd* sec later."

Figure 3 illustrates four instances of the built-in ramp prototype, showing the use of shift, stretch, and a combination of the two to achieve different functions. Although most primitives shift and stretch in a linear manner, it is possible to define prototypes that change shape arbitrarily according to the time and duration factor as described earlier. A simple example is the built-in *sin* prototype. The function labeled *d* in Fig. 2 is an instance of *sin*(3) @ 2 (with the default duration factor of 1). If we were to change the duration factor, for example *sin*(3) ~ 5 @ 2, we would still get a 3-Hz sine curve, but there would be 15 periods to fill the 5-sec duration.

Variables and Assignment

Variables are allowed in Arctic, but they are not like more familiar variables in procedural languages like Pascal and C. In Arctic, a variable can be assigned a value one time at most, but the value can be a function of time. The assignment operation is denoted by the symbol ":=."

To illustrate variables and assignment, we present the expressions that were used to generate Fig. 2. First, we write expressions to declare and define *a*, *b*, *c*, and *d*:

```
value a, b, c, d;
a := unit * 0.7;
b := ramp;
c := ramp @ 2;
d := sin(3) @ 2;
```

Now we write four expressions describing the four graphs. Notice how nonoverlapping functions can be added to combine them into one:

```
a + c
b + d
a + b + c * d
a = b or c > d
```

Collections and Sequences

Up to this point, we have only considered prototypes that yield instances that are real-valued functions. However, a response to an event can consist of a number of parallel activities; *collections* allow us to group several prototype instantiations together. Figure 4 illustrates the following collection:

```
value x, y;
[x := sin(1); y := ramp]
```

The syntax of collections is a list of expressions separated by semicolons and enclosed between square brackets. The expressions are evaluated with the same start time and duration factor, so in this case, the *sin* and *ramp* instances overlap in time. The value of the collection expression is just the value of the first prototype in the collection, so we have plotted *x* and *y* instead.

A *sequence* is a special kind of collection that is used to specify sequential behavior. Figure 5 illustrates the sequence:

```
[sin(1)|ramp]
```

Notice that the syntax is similar to that of a collection, but the prototypes are separated by vertical bars (|) rather than semicolons. The prototypes are instantiated in sequence rather than in parallel, and the result is the sum of the instances.

Asynchronous Behavior

So far, we have seen how complex behaviors can be specified by combining prototypes. All of the behaviors looked at so far have had the property that,

Fig. 4. An instance of the collection [x := *sin*(1); y := *ramp*].

Fig. 5. An instance of the sequence [*sin*(1)||*ramp*].

Fig. 6. An instance of the **do-until-then** *construct. The first prototype is terminated when it reaches 1.5, at which time a new prototype is instantiated.*

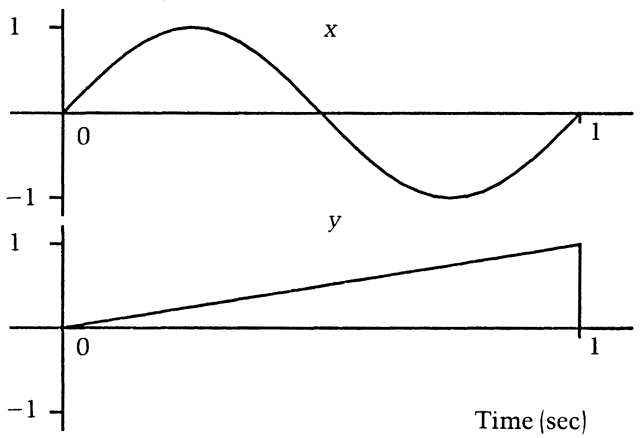
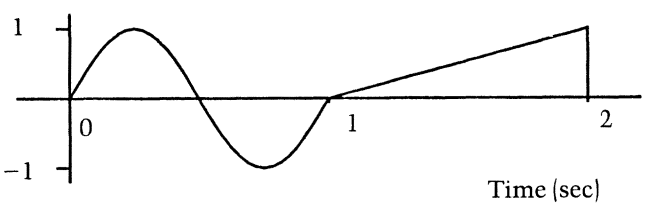


Fig. 5



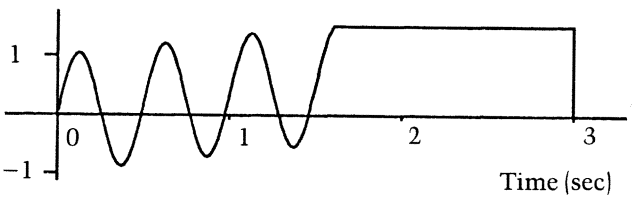
once instantiated, they run to completion in a prespecified manner. Arctic also allows one to define prototypes that can change in response to future asynchronous events. The idea is fairly simple: an instance is started and allowed to run until some designated event occurs. At that time, the instance is terminated and replaced by an instance of another prototype.

As an example, suppose we want a function consisting of an increasing sinusoidlike curve that rises until it reaches the value 1.5 at which time it holds the value 1.5. The entire function should last 3 sec. Let us use the prototype

$$x := (sin(2) + ramp) \sim 3$$

as the increasing function. Since it is not obvious when x will reach the value 1.5, we will write an Arctic expression that becomes true at that time. The expression is simply

$$x > 1.5$$



Now we are ready to write the desired prototype:

do $x := (sin(2) + ramp) \sim 3$
until $x > 1.5$
then $(unit * x(time)) \sim (3 - time)$;

The expression following **do** computes the first part of the function, and the expression following **until** is evaluated in parallel. When this latter function becomes true, the first expression is terminated and the prototype following **then** is instantiated. In this expression, *time* is a special variable whose value is the instantiation time (the time at which $x > 1.5$), and $x(time)$ is the value of x at *time*. We multiply a *unit* by $x(time)$ to make the function continuous. (In this case we know $x(time)$ will be 1.5, but if we had used a different **until** condition, it might not be possible to know the value in advance. Using $x(time)$ is a more general way to make sure the function is continuous.) The *unit* prototype is stretched by $3 - time$ to achieve a total time of 3 sec. The value returned by the **do-until-then** construct is the sum of the **do** expression and the **then** expression, as shown in Fig. 6.

A few more details are necessary to complete the description of this little program. It is important to note that the value of an assignment is the value of the expression being assigned, so the **do** expression is used not only to define x but also to yield a value. Also, notice that $x(time)$ is being evaluated at a discontinuity because x goes to zero at *time*. Intervals in Arctic are always open on the left and closed on the right, so $x(time)$ evaluates to 1.5, not 0.

Prototype Definition

In Arctic, new prototypes can be named and defined in terms of existing prototypes. To illustrate this, we will define a new prototype called *Note* and give

examples of its use. The *Note* prototype takes advantage of a special form of assignment (+=) which can only apply to variables declared as **sum** variables. The following program

```
sum x;
[x += ramp; x += sin(2)];
```

is equivalent to

```
value x;
x := ramp + sin(2);
```

More specifically, if *x* is a **sum** variable, its value is the sum of every expression appearing on the right side of assignments of the form "*x* += *e*." As we will see, this type of assignment is useful for combining the effects of a number of different prototype instantiations.

The *Note* prototype computes amplitude and frequency controls named *amp0* and *freq0* for a synthesizer. (*amp0* and *freq0* are used for illustration and are ordinary Arctic functions. Arctic itself does not dictate how control functions are interpreted to generate sound.) The definition is:

```
out sum amp0, freq0;
Env is 90 * [ramp ~ 0.2|(1 - ramp) ~ 0.6|zero ~
  0.2];
Note(in pitch) causes [
  amp0 += Env;
  freq0 += unit * pitch;
  ];
```

To instantiate *Note*, the expression given after **causes** in the definition is instantiated, resulting in functions being added to *amp0* and *freq0*. The **out** keyword indicates that *amp0* and *freq0* are system outputs. The function added to *amp0* results from an instantiation of *Env*, another prototype defined as a sequence. The first part of the *Env* sequence uses the *ramp* prototype to compute an attack, the second part uses 1-*ramp* to compute a decay, and the third part causes the function to remain at zero for a short interval. Stretch (~) operations are used within the sequence to adjust the length of each part, and so that the total length of the result will be the duration factor. This sequence is used as an amplitude envelope function. The envelope is multiplied by 90 in order to get a medium loud sound from our synthesizer. The keyword "**is**" in the definition of *Env* indicates that *Env* returns a value.

In this example, pitch is held constant through the duration of each note. This is accomplished by using *unit* to generate a constant function whose length is the duration factor. The constant is multiplied by *pitch*, which is a parameter to *Note*. For example, the following collection:

```
[Note(48) @ 0; Note(60) @ 1;]
```

results in the *amp0* and *freq0* illustrated in Fig. 7. Notice how the desired pitches (48 corresponds to C4, and 60 corresponds to C5) are provided as parameters to *Note*. The duration of the notes is one because the default duration factor is one.

Example: A Compositional Algorithm

The next example uses Arctic to compute a sequence of notes, each with random pitch. While this example will win no prizes for composition, it illustrates how Arctic programs can span many levels of musical structure, from overall structural to details of envelopes or even acoustic signals themselves. The prototype is defined as follows:

```
repeat(Note(irnd(50) + 30), 40) ~ 0.1;
```

Let us examine this prototype expression from the inside out. The parameter passed to *Note* is *irnd*(50) + 30, which means "take a random integer from 0 to 49 and add 30." The resulting number represents a random pitch between 30 (F♯2) and 79 (G6). The random pitch is passed as a parameter to *Note*, which was defined previously.

Note is in turn a parameter to *repeat*, a special Arctic function that takes two parameters: the first is a prototype and the second is a repeat count. The prototype is instantiated repeatedly in a sequence whose length is given by the repeat count. In this case, *Note* is instantiated 40 times. This would or-

Fig. 7. The result of two instantiations of the Note *prototype. The top graph represents amplitude and the lower one represents pitch.*

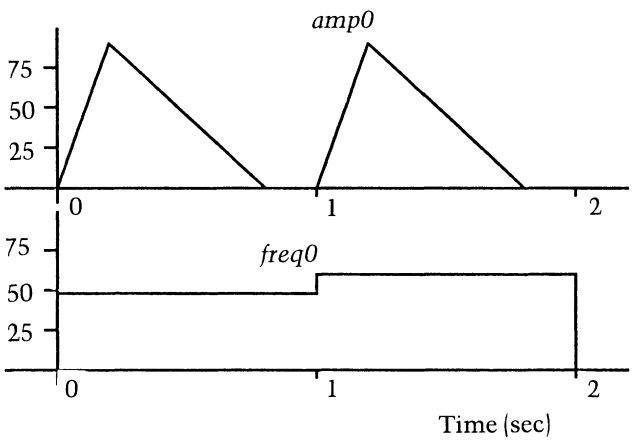
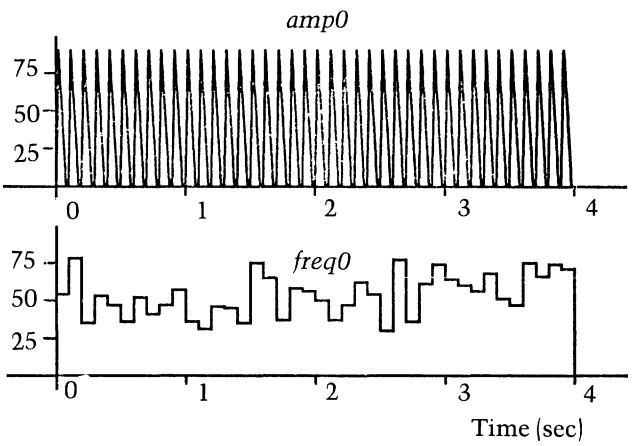


Fig. 8. The result of a prototype that computes a random pitch sequence. The notes are generated by the same Note *prototype used in Fig. 7.*



dinarily result in a behavior of length 40 sec (1 sec per note), but the whole prototype is stretched by 0.1, making each note last 0.1 sec. The overall behavior is 4 sec long, and a graph of the resulting amplitude and frequency functions is given in Fig. 8.

Let us take a more detailed look at how the *stretch* operation (~) works in this example. Recall that the duration factor is an implicit parameter to every prototype. The duration factor at the outermost level of expressions is one. This is multiplied by the stretch factor, so the duration factor passed to the *repeat* prototype is 0.1. The duration factor is passed on without modification to each instance of *Note*, from there to *Env*, and from there to the sequence prototype in the definition of *Env*. The sequence consists of three parts, each with its own additional *stretch* operation:

$$[ramp \sim 0.2 | (1 - ramp) \sim 0.6 | zero \sim 0.2]$$

The duration factor passed to *ramp* in the first part of the sequence will be 0.2 × 0.1, or 0.02 sec. The other parts of the sequence will have durations of 0.06 and 0.02 sec, respectively, for a total duration of 0.1 sec. This total duration is passed back up through *Env* and *Note* back to *repeat*, which uses the duration to place the next instance of *Note*.

A very similar mechanism is at work with the implicit start parameter. The *repeat* prototype gets the default start time of zero, which is passed through *Note* and *Env* to the envelope sequence. There, *ramp* is instantiated at time 0, 1 − *ramp* at time 0.02, and *zero* at time 0.08. The next instance of *Note* is instantiated by *repeat* with a start time of 0.1 and the same envelope sequence is recomputed 0.1 sec later. If we wanted the entire *repeat* expression to start at 2 instead of 0 sec, we could follow the expression by "@ 2":

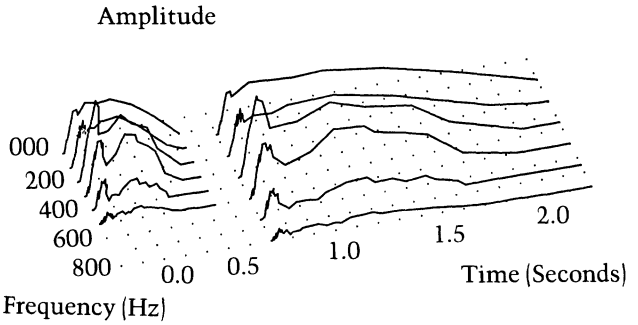
$$repeat(Note(irnd(50) + 30), 40) \sim 0.1 \ @ \ 2;$$

Then, the first *Note*, *Env*, and envelope sequence would inherit a start time of 2 and a duration factor of 0.1.

Arctic also includes a control construct for conditional instantiation (**if-then-else**). With a little more work, we could have defined the behavior in Fig. 8 using recursion instead of the special *repeat* function:

```
Random(in N) causes [
  if N > 0
    then [Note(irnd(50 + 30)|
      Random(N − 1)]
    else zero ~ 0];
Random(40) ~ 0.1;
```

Fig. 9. An original tone and a tone "stretched" nonuniformly by an Arctic program. The plotted functions represent the combined amplitude and frequency functions for 6 harmonics. Note that the attack portion is not stretched and the pitch has been transposed up a fifth.



Random generates a sequence of length N by instantiating a *Note* and following it by a sequence of length $N - 1$. If N is not greater than zero, then the prototype *zero* ~ 0 is instantiated in order to provide an end to the sequence.

Spectral Surface Manipulation

As a final example, we describe a more complex use of Arctic. We begin with a number of functions representing the amplitude envelopes of a series of partials. These envelopes result from the analysis of a tone from an acoustic instrument. The functions serve as input to an Arctic program that extracts the decay part of each harmonic and stretches it, leaving the attack portion unaltered. Other manipulations, such as the addition of vibrato or amplitude scaling are also possible. Nonlinear stretching of arbitrary functions is not a primitive in Arctic, but it can be simulated in the current implementation by breaking a function into a number of "slices," stretching the slices by different amounts, and then splicing them back together. The original and the resulting functions are plotted in Fig. 9. The Arctic program that transformed the functions on the left hand side to those on the right is less than 100 lines long.

Discussion

Arctic borrows from the GROOVE system (Mathews and Moore 1970) and from Music V (Mathews et al. 1969) in that behavior is represented as the combination of functions of time, and new behaviors can be created by combining and modifying functions. Arctic is also indebted to the 4CED program (Abbott 1981), in which actions can be triggered by events, and in which timing is notated explicitly. In Arctic, 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, Arctic uses functions to denote time-varying responses, and our resulting language will be of the applicative, or functional style (Backus 1978) as opposed to the imperative style.

Several object-oriented languages have been developed to solve the same problems of real-time control that we address. In the language FORMES (Rodet and Cointe 1984; Cointe and Rodet 1984), 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. Related object-oriented systems include those for animation described by Kahn (1979) and Reynolds (1982).

Other languages have integrated functional ideas with more traditional process-oriented implementations. The OWL language (Donner 1983), designed to program multilegged robots, has processes with implicit looping and supports an event- and condition-driven style of programming. The Formula sys-

tem implemented by Anderson and Kuivila (1985; 1986) includes a unique facility for describing continuous functions procedurally. These functions can be "attached" to other processes and are evaluated only at points in time where the value is needed.

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. Programmers must take great pains to prevent sequential execution from interfering with the real-time behavior of their 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 sequential execution and state change; these only make sense within the time domain.

Another advantage of the functional style is that synchronization of expression evaluation is implicit. For example, if the value of an expression is assigned to a variable, then the expression must be evaluated *before* the variable is used in some other expression. This follows from the rule that variables can only be assigned a value once. When many expressions are being evaluated to compute a number of functions, implicit synchronization is an important factor in keeping programs small and understandable. Implicit synchronization is also interesting because of its implications for designing and programming highly parallel computers called *dataflow machines* (Dennis 1980; Wadge and Ashcroft 1985).

Arctic in Real Time

The current implementation of Arctic is an interpreter and is not a real-time system. The interpreter reads inputs and programs from files, computes output functions and events, and writes them to another file. The file can be used as input to a program that drives a synthesizer, plotter, or some other device in real time. An interactive version of the interpreter evaluates expressions as they are typed and displays output functions graphically. The interpreter operates on entire functions, one operation at a time. For example, to compute $[a + b; x * y]$, the interpreter forms the sum of a and b and then takes the product of x and y. Thus, the value of $a + b$ at time 100 is computed before the value of $x * y$ at time 0. This is, of course, not the sort of execution order one would want in a real-time system.

We are investigating the problems of a real-time implementation of Arctic. Such an implementation will have to interleave the computation of functions that overlap in time. Although this could be accomplished by using many processes (one for each function), the overhead of context switching would be too high in most cases. Another problem is that execution order is implicit in Arctic and must be computed at run time. The way in which we intend to deal with these issues is outlined next.

In a conventional multiple process system, switching from one process to another involves saving processor registers associated with the current process, finding the new process, and restoring registers for its use. This often takes hundreds of instructions. In Arctic, "processes" correspond to expressions and assignments, which usually take only a few instructions to evaluate. If Arctic were to be implemented with conventional processes, it would spend nearly all of its time switching contexts, leaving little time for computation.

Fortunately, evaluating Arctic expressions does not require a mechanism as elaborate as conventional processes. All that is required is a way to keep track of what expressions are to be executed. The expressions themselves can be compiled into instruction lists that save and restore little or no context from one execution to the next. The idea is to simulate continuous change by reevaluating all expressions periodically. Only the current value of the functions denoted by the expressions will be kept in memory.

One implementation technique we have used in another real-time system is to dynamically compile

instructions to execute the desired expressions. For each expression to be executed, the following three machine instructions are generated:

```
load context-address
call expression-implementation
jump next
```

Each instance of a prototype has a corresponding block of memory to hold its local variables, parameters, and other context information. The first instruction loads a register with the address of this block. The second instruction calls a precompiled subroutine that implements the expression. When the call returns, the third instruction transfers control to execute the next expression.

An executing Arctic program contains a list with one of these instruction triples for each active expression. The triples are linked together through the address fields of the jump instructions, so the list is simultaneously a data structure and executable code. To instantiate a prototype, a context is first created and initialized. Then, a triple of instructions is generated for each expression and linked into the list of active expressions. This corresponds to process creation in a conventional system.

The second implementation problem is to make sure that expressions are executed in the proper order. In particular, the implementation should guarantee that assignments to variables always precede any use of that variable in the list of active expressions. Thus, the list of active expressions must be sorted accordingly. Unfortunately, there are cases where the insertion of just one expression can force the list to be completely reordered. We plan to experiment with different heuristics and compromises, and we believe that good performance can be achieved on "real" programs.

Use in the Musician's Workbench

It is our hope that Arctic will become a practical language for real-time control, and we plan to make Arctic an integral part of the Musician's Workbench, a computer music system we are constructing at Carnegie Mellon University. Although space will not allow a complete description of the project, it may be useful to show how Arctic will fit into the scheme of things.

The highest level interface to the Musician's Workbench will be an interactive score editor, which will be used for both producing printed scores and controlling synthesizers. The editor will support conventional music notation as well as extensions defined by composers. Here is where Arctic comes in. If composers define their own notation, how will they describe to the system what the notation means? Our solution is to translate graphic scores into Arctic programs. (In this case it is not necessary that the Arctic program run in real time.)

For example, a composer might design an Arctic prototype that takes a pitch and two other parameters called *Attack* and *Color.* Using the score editor, the composer can create a score in conventional notation and specify values for *Attack* and *Color* for each note as desired. To perform the score, each note is translated by the system into a prototype instance. The starting time and duration of each note are used as the starting time and duration of the prototype instance, and the pitch, *Attack,* and *Color* attributes of each note are passed as parameters.

It should be mentioned that, in keeping with Arctic concepts, the score editor will allow one to create and edit functions of time in addition to discrete events like notes. These functions can also be input to Arctic programs as in the spectral surface example.

For sound output, we will extend Arctic so that external routines (probably written in the C language) can be substituted for Arctic prototypes. This will make it easy to interface Arctic to various input and output devices without adding extensive facilities to the language. We are currently working with a device called the Bradford Musical Instrument Simulator, or BMIS, developed by Peter Comerford at the University of Bradford. The BMIS provides 64 table-lookup oscillators with interpolation between tables. We are writing software for the BMIS that will allow us to view amplitude, pitch, and interpolation controls as functions to be computed by Arctic programs.

With a real-time implementation, composers will be able to program the Musician's Workbench as a

performance device. The positions of various sensors will be input to Arctic programs as functions of time, and discrete input events such as keys will cause the instantiation of composer-defined prototypes which will in turn produce the desired response. Real-time outputs will be used to control the BMIS or other real-time devices.

Summary and Conclusions

We have outlined a new approach to the problem of real-time control. Our approach is based on the concept of the *prototype*, an abstraction of behavior, and on the use of functional programming. Arctic programs are characterized by the lack of sequential execution, the absence of side effects, and inherent parallelism. We know of no other functional language that can specify real-time system response or deal with asynchronous events. In contrast to sequential programming languages, the *evaluation* of Arctic programs is not time-dependent. This makes it much simpler to manipulate values that are time-dependent.

We have used Arctic to design and manipulate individual sounds as well as entire compositions. Arctic makes it possible to express rather elaborate behaviors with a concise, high-level notation, enabling composers and researchers to concentrate on creation and problem solving rather than programming and debugging.

Several aspects of Arctic require more thought and experiment. The use of starting times and durations solves just part of the more general problem of *temporal alignment*. One possible extension to Arctic is to allow the *shift* operation (@) to take a time map (Jaffe 1985) as its right argument. Second, Arctic provides an elegant, but not entirely practical 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. Finally, Arctic must be extended with data structures such as arrays and lists, in order to more easily perform arbitrary computations. Perhaps an interface to a procedural language will alleviate this problem.

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.

Acknowledgment

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.

References

Abbott, C. 1981. "The 4CED Program." *Computer Music Journal* 5(1):13–33.

Anderson, D. P., and R. Kuivila. 1985. "Continuous Abstractions for Discrete Event Languages." Unpublished.

Anderson, D. P., and R. Kuivila. 1986. "Accurately Timed Generation of Discrete Events." *Computer Music Journal* 10(3):48–56.

Backus, J. 1978. "Can Programming Be Liberated from the von Neumann Style?" *Communications of the Association for Computing Machinery* 21(8):613–641.

Cointe, P., and X. Rodet. 1984. "FORMES: An Object and Time Oriented System for Music Composition and Synthesis." In *1984 ACM Symposium on LISP and Functional Programming*. New York: Association for Computing Machinery, pp. 85–95.

Dannenberg, R. B. 1984. "Arctic: A Functional Language for Real-Time Control." In *1984 ACM Symposium on LISP and Functional Programming*. New York: Association for Computing Machinery, pp. 96–103.

Dannenberg, R. 1986. "Arctic: Functional Programming for Real-Time Systems." In *Proceedings of the Nineteenth Hawaii International Conference on System Sciences*. New York: IEEE, pp. 216–226.

Dennis, J. B. 1980. "Dataflow Supercomputers." *Computer* 13(11):48–56.

DOD. 1980. *Reference Manual for the Ada Programming Language*. Washington, D.C.: United States Department of Defense.

Donner, M. 1983. "The Design of OWL—A Language for Walking." In *Sigplan Symposium on Programming Language Issues In Software Systems*. New York: Association for Computing Machinery, pp. 158–165.

Glass, R. L. 1980. "Real-Time: The 'Lost World' Of Software Debugging and Testing." *Communications of the Association for Computing Machinery* 23(5):264–271.

Hoare, C. A. R. 1978. "Communicating Sequential Processes." *Communications of the Association for Computing Machinery* 21(8):666–677.
Iverson, K. 1962. *A Programming Language.* New York: Wiley.
Jaffe, D. 1985. "Ensemble Timing in Computer Music." *Computer Music Journal* 9(4):38–48.
Kahn, K. 1979. "DIRECTOR Guide." Memo 482B. Cambridge, Massachusetts: M.I.T. Artificial Intelligence Laboratory.
Mathews, M. V. et al. 1969. *The Technology of Computer Music.* Cambridge, Massachusetts: MIT Press.
Mathews, M. V., and F. R. Moore. 1970. "A Program to Compose, Store, and Edit Functions of Time." *Communications of the Association for Computing Machinery* 13(12):715–721.
May, D. 1983. "OCCAM." *Sigplan Notices* 18(4):69–79.
McCarthy, J. 1965. *LISP 1.5 Programmer's Manual.* Cambridge, Massachusetts: MIT Press.
Reynolds, C. U. 1982. "Computer Animation with Scripts and Actors." In *Proceedings of SIGGRAPH Conference.* New York: Association for Computing Machinery.
Rodet, X., and P. Cointe. 1984. "FORMES: Composition and Scheduling of Processes." *Computer Music Journal* 8(3):32–50.
Wadge, W. W., and E. A. Ashcroft. 1985. *Lucid, the Dataflow Programming Language.* New York: Academic Press.
Wirth, N. 1977a. "Modula: A Programming Language for Modular Multiprogramming." *Software, Practice and Experience* 7(1):3–35.
Wirth, N. 1977b. "Design and Implementation of Modula." *Software, Practice and Experience* 7(1):67–84.
Wirth, N. 1982. *Programming in Modula-2.* New York: Springer-Verlag.