Expressing Temporal Behavior Declaratively[1]

Roger B. Dannenberg
School of Computer Science
Carnegie Mellon University

ABSTRACT

The programming language Arctic specifies real-time behavior declaratively by using temporal control constructs and by indicating starting times and durations explicitly, much the way timing is specified in a cue sheet or a musical score. Values in Arctic are functions of time, which may be combined with various arithmetic and logical operators. Since Arctic is a single assignment language, the execution order is implied by data dependencies, simplifying synchronization problems for the programmer. Arctic supports behavioral abstraction, in which a single program module gives rise, through various transformations, to a class of behaviors. An implementation of Arctic is described, and experience with the declarative approach to real-time control is discussed.

1. Introduction

For the most part, traditional programming languages have been designed for applications where control over the timing of execution is not a primary concern. Users usually want their programs to run as fast as possible, but program correctness is not affected by program speed. In contrast, for some applications it is necessary to specify not only output values, but the times at which output values are to be produced. A ‘‘correct’’ value that is early or late is not acceptable.

The most common examples of applications where timing is central are real-time programs which control physical devices. Real-time systems are commonly used for controlling appliances, robots, vehicles, power plants, and manufacturing processes. Time is also important in non-real-time applications such as some computer animation and computer music systems. In these applications, images or sounds are computed and stored for real-time playback at a later time. Even though this computation is not in real-time, temporal behavior in the ‘‘virtual time’’ of the output medium (frames of film or samples of audio) is still of the utmost importance.

---

[1]A version of this article appears in ***CMU Computer Science, A 25th Anniversary Commemorative***, edited by Richard F. Rashid, ACM Press Anthology Series (Chapter 3), pp. 47-68. Unfortunately, many typographical errors were introduced in that version of the text. This printing contains the corrections.

Simulation is another example of a non-real-time application area in which (virtual) time is still important.

Research in computer music has resulted in a number of innovative languages [2, 4, 6, 9, 19] and systems [7, 12, 13, 18] for dealing with virtual and real time. This is not too surprising given that music can be defined as sound organized in time. Long before electronic computers, traditional music notation allowed composers to create temporal specifications which include loops, counters, conditionals and other constructs familiar to modern-day programmers. [8]

In the following sections, I will describe the language Arctic [17], for expressing temporal behavior. Two variants of Arctic called Canon [10] and Fugue [11], have similar semantics but deal with different domains. A real-time implementation of Arctic is in progress and will also be described.

2. Related Work

Arctic is based on earlier work in the computer music field including the Music V language [14] by Max Mathews and 4CED [1] by Curtis Abbot. Music V was designed to synthesize music in non-real-time and consists of two parts called the *score language* and the *orchestra language*. Music V was innovative in several ways:

- First, the Music V score language allows users to specify a list of notes and when they should occur. The idea of associating an explicit time with program statements is quite different from the use of clocks and synchronization primitives in more traditional languages.
- Second, statements in the Music V score language serves as ‘‘triggers’’ that starts instances of instruments described in the orchestra language. This provides a natural form of parallelism and also serves to link discrete events to continuous-time[2] signals. (We will use the term *signal* throughout to mean a real-valued function of time. Do not confuse *signals* in Arctic with synchronization mechanisms of other languages by the same name.)
- Third, the orchestra language is essentially a data-flow language in which operators are connected by data streams. (Music V predated data-flow architectures and streams by several years.) This turns out to be a natural way to describe operations on continuous-time signals.

The 4CED language was influenced by Music V but is more dynamic. 4CED is a real-time language for controlling a digital audio signal processor, hence external sensors can trigger timed sequences similar to Music V scores.

Assuming that these continuous-time signals in Arctic are actually implemented as sequences of discrete values, Arctic bears a similarity to the languages Lucid [3] and Val [16] which are also functional or data-flow languages for manipulating sequences. One major difference, however, is that in Arctic, variables are functions of a single global time, while in Lucid and Val,

---

[2]Here, *continuous-time* refers to functions of time in which time is thought of as continuous (real). In contrast, discrete events occur at some countable set of time points. In practice, even continuous-time signals require a finite (and therefore discrete) representation such as samples or piece-wise linear representations.

sequences can be computed and indexed without any connection to time. Another related language is L.0 [5], which treats time as a sequence of discrete steps and uses a very general naming and scoping mechanism, but is otherwise much like Arctic.

3. The Programming Language Arctic

Arctic has two primary ingredients that distinguish it from other functional languages. First, there is a mechanism for specifying *when* things are to be computed. A variety of control constructs and operators make it possible to give explicit specifications of temporal behavior. Second, expressions manipulate functions of time. For example, writing

a := *b* + *c*

defines the signal $a(t)$ to be the sum of two signals $b(t)+c(t)$.

The primary abstraction mechanism in Arctic is the *prototype*, a named behavior that can be instantiated any number of times. A *prototype* in Arctic is analogous to a function or procedure in other languages, but it is important to note that prototypes specify behaviors that exist over time. Let us illustrate some simple Arctic expressions using prototypes. The expression

ramp

denotes an instance of *ramp*, a built-in prototype representing a linear interpolation from 0 to 1, nominally over the time interval (0, 1]. (Signals are always computed over some interval that is open on the left and closed on the right.)

The ability give explicit timing specifications is illustrated by the following expression:

ramp @ 3

which denotes an instance of *ramp* that begins at time 3 and ends at time 4.

Notice in these examples how expressions for discrete-time events (e.g. ‘‘start the ramp at time 3’’) can give rise to continuous-time signals (e.g. interpolate from 0 to 1 on the interval 3 to 4). This mechanism is an important bridge between the worlds of discrete time and continuous time, both of which are important in real-time systems.

4. Temporal Specification

This section expands upon the facilities in Arctic for specifying temporal behavior of discrete-time computation. Control constructs are provided to support parallelism, sequential execution, shifting, and stretching, as well as temporal versions of conditional and iterative expressions.

4.1. Parallel Collections

Arctic has constructs for parallel and sequential collections of behaviors. The parallel collection is a list of expressions enclosed in brackets and separated by semicolons: the expression

[*A*; *B*; *C*]

denotes the simultaneous instantiation of prototypes *A*, *B*, and *C*.

4.2. Sequential Collections

The sequential collection uses the vertical bar as a separator: the expression

[*A* | *B* | *C*]

denotes the sequence *A* followed by *B* followed by *C*. In Arctic, all instances of prototypes have durations in addition to starting times so that sequential behavior is well-defined.

4.3. Transformations

When a prototype is instantiated, it is passed two implicit parameters, called the *start time* and *duration factor*. These tell the prototype instance when to start and by how much to stretch the nominal duration. These implicit parameters can be altered by various operators including @, @@, ~, and ~~. To explain these operators, it is best to show the start time and duration factor explicitly within braces:

$(A \text{ @ } t)\{s, d\} = A\{s + td, d\}$
$(A \sim x)\{s, d\} = A\{x, dx\}$
$(A \text{ @@ } t)\{s, d\} = A\{t, d\}$
$(A \sim\sim x)\{s, d\} = A\{s, x\}$

Thus, @ (called *shift*) provides relative time-shift and @@ provides absolute time-shifting. Similarly, ~ (called *stretch*) provides relative duration scaling and ~~ provides absolute duration scaling. As an example, consider the expression *A @ 5 ~ 2* evaluated at the top-level where the default starting time is 0 and duration factor is 1:

$(A \text{ @ } 5 \sim 2)\{0, 1\} = (A \text{ @ } 5)\{0, 2\}$
$= A\{10, 2\}$

So *A @ 5 ~ 2* starting at time 0 with duration factor 1 is equivalent to *A* starting at time 10 with duration factor 2.

Most Arctic expressions return two values: a signal and a stop time. Signals will be discussed in the next section. The stop time conveys the duration of the behavior, which is needed in sequential collections. The programmer can override the stop time by providing an explicit one: *A $ x*{*s, d*} has a stop time of *s + dx* and *A $$ x*{*s, d*} has a stop time of *x*.

4.4. Defining Prototypes

Transformations are dynamically scoped. If there is a definition

Name **is** *Body*;

then *Name*{*s, d*} denotes *Body*{*s, d*}. For example, a new prototype can be defined as follows:

MyProto **is** [*A*; *B* @ 1; *C* @ 2];

We can then write an expression such as *MyProto ~ 2*, which means:

[*A*; *B* @ 1; *C* @ 2] ~ 2

4.5. Conditionals and Iteration

Conditional execution can be obtained with a conventional looking **if** construct:

if *C* **then** *A* **else** *B*;

where *C* is a conditional expression and *A* and *B* are prototype expressions. The condition *C* is tested at the start time of the **if** and then either *A* or *B* are instantiated.

Iteration can be accomplished using the **while** construct:

while *C* **do** *A*

which is equivalent to an infinitely nested **if**:

if *C* **then** [*A* |
if *C* **then** [*A* |
...
]];

The result is a sequence of *A*'s which is terminated when *C* becomes false. The condition *C* is only tested before each instantiation of *A*.

5. Signals

The most interesting data type in Arctic is the signal, which represents a real-valued function of time over some interval. Signals are returned from instances of prototypes (including many built-in prototypes), signals may be assigned to variables, and signals may be combined using various arithmetic and logical operators.

5.1. Prototypes

The *ramp* prototype seen earlier is a good example of a built-in prototype that returns a signal and stop time. We can now give a more formal description of *ramp*. Just as we introduced the extended notation ''{*s*, *d*}'' to make start times and duration factors explicit, we will now introduce the notation ''(*t*)'' so that we can talk about signals as a function of time. If *s* is a signal, then *s*(*t*) is the value of the signal at time *t*. The stop time is indicated by the extended notation ''.stop''; for example, if *I* is an instance of a prototype, then *I*.stop denotes the stop time of the instance.

ramp{*s*, *d*}(*t*) =
0 for *t* <= *s* or *t* > *s* + *d*
(*t* – *s*) / *d* for *s* < *t* <= *s* + *d*

ramp{*s*, *d*}.stop = *s* + *d*

We can now give a more formal description of parallel collections:

[*A*; *B*; ...]{*s*, *d*} = [*A*{*s*, *d*}; *B*{*s*, *d*}; ...]

[*A*; *B*; ...]{*s*, *d*}(*t*) = *A*{*s*, *d*}(*t*)

[*A*; *B*; ...]{*s*, *d*}.stop = *max*(*A*{*s*, *d*}.stop,
B{*s*, *d*}.stop,
...)

The description is in three parts. The first part indicates that an instance of a collection is simply a collection of instances with the same start time and duration factor. The second part indicates that the signal returned by the collection is the signal returned by the first instance of the collection. The third part indicates that the stop time of the collection is the maximum of the

stop times of the instances of the collection. This description is still not complete because it omits any specification of naming, parameter passing, and effects on global variables. More detail will be presented in section 5.2, and a complete description can be found in the Arctic reference manual [17].

We can also describe sequential collections in terms of the parallel collection:

$$[\ A \mid B \mid C \mid \ldots\ ]\{s, d\} = [\ A\{s, d\};\ B\{A\{s, d\}.\text{stop}, d\};\ C\{B\{A\{s, d\}.\text{stop}, d\}.\text{stop}, d\};\ \ldots\ ]$$

$$[\ A \mid B \mid C \mid \ldots\ ]\{s, d\}(t) = A\{s, d\}(t) + B\{A\{s, d\}.\text{stop}), d\}(t) + C\{B\{A\{s, d\}.\text{stop}), d\}.\text{stop}), d\}(t) + \ldots$$

$$[\ A \mid B \mid C \mid \ldots \mid Y \mid Z\ ]\{s, d\}.\text{stop} = Z\{Y\{\ \ldots, d\}.\text{stop}, d\}.\text{stop}$$

This indicates that *B* is instantiated at the stop time of *B* and *C* is instantiated at the stop time of *C*, etc. The definition is not quite correct because it calls for more than one instance of each prototype except the last one; however, the implementation must insure that there is only one instance of each prototype in the sequence. The second part indicates that the signal returned is the sum of the signals returned by the elements of the sequence, and the third part indicates that the stop time of the sequence is the stop time of the last element of the sequence.

5.2. Values, Variables, and Assignment

Values in Arctic may be either real or boolean and may be scalars or signals, giving a cross-product of four types. In mixed expressions, scalars are coerced to constant signals; for example, in the expression 1 – *ramp*, the scalar 1 is coerced to *step*(1), which is defined as follows:

$$\textit{step}(c)\{s, d\}(t) = \begin{cases} 0 & \text{for } t <= s \\ c & \text{for } t > s \end{cases}$$

In this definition, constants are coerced to signals that are zero previous to the start time. This definition is a desirable because we do not want the creation of an instance to affect history, making programs non-causal.

Variables may be declared in Arctic and used in expressions. Arctic is a single assignment language, so an instance of a variable may only be assigned once; however, the signal being assigned to a variable may be a function of time. For example, we can write

output := *x* * *input*;
x := 1 – *ramp*;

Here, *x*, *input*, and *output* are variables. Assignments in Arctic establish one-way constraints:

var := *expr*{*s*, *d*} implies *var*(*t*) = *expr*(*t*) for *t* > *s*

This in turn imposes a partial ordering on the evaluation order of Arctic assignments and expressions. In the preceding example, it is necessary to evaluate the second assignment to obtain *x* before evaluating the first assignment which uses *x*. (A real-time implementation would evaluate both assignments at every time-step in the implied partial order.)

6. Termination of Expressions

So far, we have seen how discrete events can give rise to continuous signals. In Arctic, events can also terminate computations and replace them with new ones. The **do-until** control construct is used for this purpose. For example,

do *A* := *ramp* **until** *A* >= *input* **then** *input*

means to compute a linear ramp and assign it to *A* as long as *A* is less than *input*. At that point, stop computing the ramp (*A* will subsequently be zero) and return the value of *input*. Assuming *input* is between 0 and 1, the value of the **do-until** expression makes a smooth transition from 0 to *input*.

If the **do** part of the **do-until** expression finishes before (or even when) the condition becomes true, then the entire expression finishes immediately. Otherwise, the expression finishes when the **then** part finishes.

It is possible to wait for the instantiation of a prototype using an **event** expression:

do *P* **until** (**event** *ButtonPress*(*b*) **where** *b* = 1)

means to terminate *P* if *ButtonPress(1)* is instantiated. Whenever *ButtonPress* is instantiated, **event** *ButtonPress*(*b*) becomes true and *b* is bound to the current value of the parameter passed to the *ButtonPress* prototype. The scope of *b* is the expression following the **where** keyword.

Note the similarity between **event** expressions and the rendezvous in Ada. Both are a means of momentarily synchronizing the execution of two independent but communicating threads of control. In Arctic the joining is ‘‘instantaneous’’ and does not cause blocking or queueing.

7. Application in Various Domains

Variants of Arctic have been constructed for three different domains. All of the variants are interpreters that operate out of real time on data describing temporal behavior.

7.1. Interpreted Arctic

The first variant computes piece-wise linear functions which are typically used to control synthesis processes. For example, control functions might specify the pitch, amplitude and harmonic content of an audio oscillator. This version of Arctic [17] has been used to compose music, to synthesize warning tones for nuclear reactors, and to generate test signals for laboratory instrumentation.

This is a ‘‘variant’’ of Arctic in the sense that functions are not computed in real time or even in time order. For example, the expression *(a + b) * c* would be interpreted by constructing the sum of signals *a* and *b* and then multiplying[3] by *c*. Because signals are piece-wise linear, they can be stored compactly as a list of breakpoints. The interpreter uses static analysis of the parse tree to determine the order of expression evaluation.

---

[3]Since piece-wise linear functions are not closed under multiplication, we conveniently define multiplication in the Arctic interpreter to yield correct breakpoints wherever breakpoints occur in the factors. All other points in the result are defined by the linear interpolation of these breakpoints.

7.2. Canon

Unlike the first variant, which implements Arctic syntax, the second variant, named Canon [10], is embedded in Lisp. Canon operates on sequences of musical notes which have several properties including pitch, loudness, starting time, and duration. These properties can be altered using transformations analogous to the shift and stretch transformations already discussed. For example,

```
(defun triad ()
    (seq (note C4) (note E4) (note G4)))

(play (transpose 3 (loud -20 (triad))))
```

defines **`triad`** to be a sequence of three notes with pitches C4, E4, and G4. (This sequence is called a major triad). The **`triad`** behavior is transposed by 3 semitones and softened through the use of a negative loudness offset.

Canon does not include signals as an ordinary data type as does Arctic. However, Canon does allow transformations to be time-varying. For example, it is often useful to gradually increase the loudness of a phrase of music. This can be accomplished in Canon by specifying a loudness contour to the **`loud`** transformation:

```
(loud (env (0 0) (1000 30) (2000 10))
      (myscore))
```

In this example, the **`env`** function constructs a piece-wise linear contour that increases from zero to 30 over the first 10 seconds and then decreases to 10 after a total elapsed time of 20 seconds. The contour is evaluated at the note starting-times to determine the loudness offset for each note in **`myscore`**.

As with ordinary transformations, time-varying transformations can be nested. The most interesting time-varying transformation is **`warp`**, which implements time mapping. The time map transforms ‘‘program time’’ or ‘‘virtual time’’ into real time according to an arbitrary (but monotonically increasing) function, allowing easy specification of music that accelerates or slows down. Unfortunately, the contours used to specify time-varying transformation are not supported by a rich set of operations as are signals in Arctic. This is a limitation of the implementation but not of the general approach.

The Lisp environment used by Canon allows composers to take advantage of a rich language for expressions and data structures; it also minimizes the implementation effort. In fact, Canon control constructs are implemented as macros. The implementation is simplified by the fact that there are no signals or (single) assignments supported by Canon. Canon illustrates that the timing and transformation semantics of Arctic can be put to good use in other language contexts.

7.3. Fugue

The third variant, called Fugue, is also embedded in Lisp. Fugue is much like Canon, but rather than computing note-level information, Fugue is intended to compute sampled audio and control signals. Fugue expressions return values of type Sound. The implementation uses a modified Lisp where Sounds and various operations on Sounds are primitives. Operations include reading and writing Sounds from files and adding, scaling, multiplying, shifting, filtering and stretching Sounds. Various Sound synthesis operations such as software oscillators are also

present.

Although our modified Lisp could be used directly for sound synthesis by writing programs that call upon the primitive Sound operations, Fugue provides a higher-level language in which time and other transformations can be used to modify behaviors. For example, a programmer could develop a way of combining oscillators and filters to obtain an interesting sound. This technique could be encapsulated in a function called **mysound**. Then, using the Arctic-like semantics of Fugue, **mysound** could be instantiated with different transformations to construct a musical phrase:

```
(defun phrase ()
    (sim (at 0.0 (mysound))
         (at 1.0 (stretch 0.5 (transpose 3 (mysound))))
         (at 1.5 (stretch 0.5 (transpose 2 (mysound))))
         (at 2.0 (transpose 10 (mysound)))
         (at 2.5 (transpose  9 (mysound)))))
```

In practice, **mysound** would probably take a pitch parameter to eliminate the need for the wordier **transpose** operations. Notice how easily (and declaratively) the **sim** (for ‘‘simultaneous’’) construct combined with the **at** (shift) operation allows the programmer to arrange instances of **mysound** in time. The **sim** construct sums (mixes) the resulting sounds so that **phrase** returns a single composite sound.

To illustrate how **mysound** might be written, here is a simple function that uses an oscillator scaled by an envelope to generate a sound:

```
(defun mysound ()
    (s-mult (osc) (env (0 0) (0.1 1) (0.7 1) (1 0))))
```

Both the **osc** and **env** primitives respond appropriately to various transformations so **transpose**, **at**, and **stretch** transformations will have the desired effect.

An interesting aspect of the Fugue implementation is that many operations on sounds are implemented using lazy evaluation. This is possible because, in keeping with the functional style of Arctic, sounds are immutable values. When two sounds are added together, the sum is represented by a small node containing pointers to the two sounds. This reduces storage requirements (digital audio requires up to 200 KBytes per second) and allows multiple operations such as shifting, scaling, and adding to be performed in a single pass over the data.

8. A Real-Time Implementation

One of the great challenges of Arctic has been to devise a reasonable real-time implementation. A real-time implementation must solve two problems: how to manage instantiation and timing of Arctic prototypes (discrete-time computation) and how to compute time-varying signals (continuous-time computation).

8.1. Data Dependencies

For the time being, let us assume that real-time Arctic signals will be represented as a stream of samples. For example, the signal denoted by the variable *A* will be represented by a single memory location that is updated regularly at an interval called the time step. At any given instant during the execution of an Arctic program, there are a number of active primitives that

generate signals as well as operators that combine signals, creating a data-flow graph. For example, the following expressions:

```
[ A := B + 5;
  B := (ramp * 3);
  C := B * input ]
```

would give rise to the graph in figure 8-1.

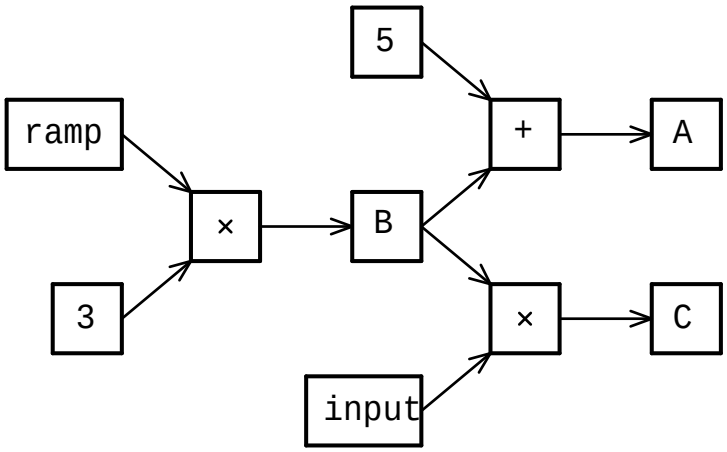

Figure 8-1: Data flow graph for an Arctic program.

If this graph were in effect for the duration of the program execution, it would be a simple matter to perform a topological sort on the data dependencies to derive an order of evaluation. This could then be used to produce a sequential program to be executed every sample time.

Unfortunately, this approach will not work because instantiation of prototypes and termination of instances modify the flow graph dynamically. The sequential collection and **do-until** constructs are language mechanisms that can give rise to dynamically changing graphs. Furthermore, instantiation can be driven by input data that is is unknown at compile-time, and parameter-passing allows essentially arbitrary (and non-hierarchical) flow graphs to be realized.

Because of these complications, the current implementation approach uses ‘‘lazy’’ or demand-driven evaluation to compute signals. Instances of expressions have at least three components: a method called ‘‘update’’ for computing the current value, a storage location for holding a value, and a time-stamp denoting when the value was computed. The algorithm for computing the current value of an expression is the following:

```
access(exp)
  begin
    if exp.timestamp < globaltime
    then
      begin
        exp.timestamp := globaltime;
        exp.value := (exp.update)(exp);
      end
    return exp.value;
  end
```

To get the value of an expression, the time-stamp is compared to the global time. If less, the

update method is called to produce a new value. Typically, the update method will recursively *access* subexpressions and variables. The effect is to bring values up-to-date in an order that is consistent with data dependencies. The order is computed dynamically. Values are cached so that expressions are recomputed at most once per time step.

As in other lazy evaluation systems, computation is driven by the demand for output values. Arctic variables can be designated as output signals. The declaration ‘‘**out** *X*;’’ indicates that one of the system outputs is the value of the variable *X*. The main program calls *access* to get the value of each output variable on each time step. This serves to drive all necessary signal computation.

Lazy evaluation is not appropriate in prototypes that compute their signals incrementally. For example, the *ramp* prototype adds a slope to an accumulator on every time step to avoid a more expensive interpolation operation. The update method of *ramp* must be called on every sample to insure correctness. For this reason, a list of active expressions is kept at run time and expressions on the list are updated on every time step along with output variables.

8.2. Instantiation

One of the features of Arctic is that prototypes can be instantiated dynamically and in parallel. Instances exist over time and computation is interleaved among active instances. The current implementation solves this problem using object-oriented programming techniques. Prototypes are compiled into classes, and the run-time instantiation of an Arctic prototype is implemented by creating a new instance of the class at run time.

When an Arctic prototype is specified in terms of a collection of other prototypes, the implementation will be a collection of corresponding objects. What should be the granularity of objects? Should each arithmetic operator be handled by a separate object? Alternatively, can several assignments be grouped into a single object? In general, since the overhead of the *access* procedure is fairly high, objects should encompass as many operations as possible for efficiency. On the other hand, it is not always possible to determine whether two assignments are dependent, so in the current implementation, a class is generated for each assignment in the Arctic source program. Also, whenever the right-hand side of an assignment includes a prototype, the prototype is implemented by a separate object rather than being expanded in-line.

One of the greatest difficulties in the implementation has been getting correct behavior at the start and stop times of instances. The first approach attempted to use a signaling mechanism whereby a child object (subexpression) would notify its parent at the stop time by invoking a pre-specified method. One of the problems with this approach is the race condition where the parent’s value for a particular time step might depend upon whether the value was updated before or after notification from a child.

The solution came with the realization that stop times can be treated as values rather than events and that data dependencies can be used to eliminate race conditions. This approach is supported by the existing lazy evaluation scheme. The view that stop time is a computed value is reflected in the descriptions of control constructs in section 5.1.

8.3. Performance

The real-time Arctic implementation is still in progress, so only preliminary performance measurements are available. In order to develop and test implementation techniques, programs are hand-compiled into an object-oriented dialect of the C programming language. Arctic primitives are provided by a run-time system and much of the behavior of collections, sequences, and assignments can be inherited from generic classes, so the translation effort is tedious but not exhausting.

A 16 MHz Motorola 68020 with a 68881 floating point coprocessor is used for real-time execution, and Arctic program outputs are used to control the pitch and amplitude of a bank of eight music synthesizers. Real-time input can be generated from a music keyboard as well as from buttons and knobs.

A single program was used to measure performance. The program responds to input from a music keyboard and a knob. When a key is pressed, the Arctic program generates time-varying pitch and amplitude contours for one of the eight synthesizers until the key is released.

The real-time requirements of Arctic are application dependent, but for music it is desirable to process input and compute several interesting control functions at rates of 100 to 200 samples per second [15]. This data rate represents approximately the bandwidth of a human interacting with a musical instrument. The determination of what is an ‘‘interesting’’ control function is very subjective: with the expressive power of Arctic, the computation per sample can become substantial with only a modest programming effort.

Rather than measure samples computed per second, it is perhaps more informative to measure evaluations per second. By running as fast as possible and eliminating calls that actually output data, the current implementation performs 12,800 value accesses per second. Of these evaluations, 6100 invoke an object’s update method and the other 6700 find the object up-to-date and simply use the current value. Evaluations include not only expressions that do the real work but also expressions such as sequential collections that merely channel signals from subexpressions. About 2800 ‘‘useful’’ expressions involving arithmetic operations are evaluated per second.

Another measure of performance is the latency from input to output. For signal input, input changes are received asynchronously and are read into Arctic variables at the beginning of every time step. This means that data arriving just after the beginning of a time step will not be seen until the next step. If the Arctic program takes an entire time step to compute outputs, then the latency from signal input to output is always less than two time steps. The variance of this latency could be made essentially zero by waiting for the full worst case delay before performing any output operations.

An alternative implementation approach would be to dispense with dependency checking and simply re-evaluate all active objects as fast as possible. This makes sense if the speedup due to lower overhead compensates for the additional latency introduced by a less-than-optimal evaluation order. In the program studied, the maximum depth of recursion of the update algorithm was measured as 5, indicating that signals propagate through 5 levels of expressions even in fairly simple programs. The dependency checking and dynamic ordering of expressions is cost-effective because the overhead does not approach the delay that would result from the

worst-case evaluation order.

Discrete event input is queued by the system and the queue is checked at the beginning of each time step. Discrete outputs will be generated in response during the time step, so the latency is again less than two time steps. When would an output signal be affected by a discrete input? Intervals in Arctic are closed on the right, so a signal that is terminated in time step *T* is not seen to change behavior until time *T*+1. Similarly, if an instance is created at time *T*, it does not affect signals until time *T*+1. Thus the latency from discrete input to an output signal is at most three time steps.

The current implementation is a ‘‘soft’’ real-time system; that is, it tries to deliver results as close to the specified output time as possible. When a deadline is not met, results are delivered late and in time order. Since Arctic programs respond to events by instantiating or terminating instances, it is important to know how long these operations take. A frequent cause of missing deadlines will be the extra load created many simultaneous events. In the current system, the key-down event immediately instantiates 12 prototypes. By simulating simultaneous key-down events, an average prototype instantiation time of 660μs (microseconds) was obtained.

All of these numbers reflect an implementation in C with a minimal amount of optimization. (One exception is that the already up-to-date case of *access* is compiled in-line.) In addition, the implementation includes extra code for instrumentation. Performance could probably be improved by a factor of 2 or 3, and a faster processor could be used for additional speedup.

9. Behavioral Abstraction

It is often easier to describe a class of similar behaviors than it is to describe each individual behavior separately. For example, if similar control sequences and signals of different durations are required, it is probably easier to parameterize one program than to write separate programs for each desired duration. A program or prototype that implements a class of similar behaviors is called a *behavioral abstraction*. Arctic, Fugue, and Canon all support the definition and use of behavioral abstractions.

A key idea behind behavioral abstraction is that behaviors do not always transform in a straightforward linear manner. For example, if a violinist were asked to play a note longer than its normal duration, what would be the expected result? Would the pitch be lowered so that the same number of oscillations would fill the allotted time? Would the violinist move the bow more slowly and thus play softer? Would the vibrato be slowed down as well? All of these suggestions are absurd because we understand ‘‘playing a note’’ to be a particular abstract behavior, and we know (or at least a violinist knows) what it means to apply various transformations to ‘‘playing a note’’ such as stretching and transposing.

It is important that Arctic and its cousins be able to express behavioral abstractions of this sort. As the musical example illustrates, it is generally not sufficient to simply shift and stretch all subexpressions in order to ‘‘shift’’ or ‘‘stretch’’ a behavioral abstraction. Arctic allows the programmer to override the default behavior of transformations in order to achieve the desired behavior. The transformation details are a part of the prototype specification. Abstraction is achieved because the implementation details are separated (and hidden) from the rest of the program.

To illustrate with a concrete example, suppose we have a large program for controlling a household robot, a part of which is used for opening doors:

```
openDoor(D) is
    [ moveToDoor(D) | turnDoorKnob(D) |
      swingOpen(D) | releaseDoorKnob(D) ];
```

so that instantiating *openDoor(D)* will cause the robot to open door *D*. Now suppose we want to instruct the robot to open a door slowly. Instantiating *openDoor(D) ~ 2* will do the job, but we notice that the robot approaches the door and turns the door knob slowly as well. This is not what we want, so the *openDoor* prototype is redefined as follows:

```
openDoor(D) is
    [ [ moveToDoor(D) | turnDoorKnob(D) ] ~~ 1 |
      swingOpen(D) | releaseDoorKnob(D) ~~ 1 ];
```

Here, the ‘‘absolute stretch’’ operator (~~) has been used to override the duration factor to all components of *openDoor* except for *swingOpen*. Thus, the duration factor supplied to *openDoor* is only passed on to *swingOpen* and the robot will now approach the door and turn the knob at normal speed.

Prototype specifications can access their own start time and duration factor using the symbols *time* and *dur*. Prototypes can also refer to global variables (signals). These mechanisms give the programmer complete access to the transformation environment and the ability to override the default transformation semantics with another behavior.

10. Discussion

After five years and four implementation efforts, much has been learned about Arctic, but there are still many open issues and problems to be solved. We will discuss the strengths and weaknesses of Arctic and suggest directions for future research.

10.1. Strengths of Arctic

Probably the most interesting feature of Arctic is its explicit and declarative approach to timing specification. Timing in procedural languages is implicit and arises as a consequence of sequential execution while timing in Arctic is explicitly specified through the use of the *shift* transformation and various temporal control constructs. On the other hand, procedural languages use explicit syntax to indicate synchronization while Arctic synchronization is implicit due to the single assignment rule and data dependencies.

A consequence of explicit timing in Arctic is that Arctic programs provide a very high-level declarative specification of temporal behavior that is similar in many ways to musical scores, scripts, and cue sheets used in the arts. The implicit synchronization provided by the single assignment rule has been the basis for a number of other languages and machines supporting parallelism, and it seems particularly appropriate for computing streams and signals.

10.2. Weaknesses of Arctic

Real-time systems have traditionally been programmed in fairly low-level procedural languages. Procedural languages provide a close match to the underlying hardware in terms of modifiable storage and sequential execution. When the synchronization support provided by Arctic is not needed and when discrete algorithms involving data structures are required, the synchronization and temporal specification features of Arctic are of little if any help.

Although techniques such as lazy evaluation, caching the results of functions, and various program transformations can often evaluate declarative programs as efficiently as their imperative counterparts, the complexity of the implementation techniques can be quite high. The Arctic run-time system is an example of the relative complexity required for a declarative approach to real-time control. It is also telling that the Arctic run-time system is implemented using an imperative object-oriented technique.

Arctic is not really a complete language in that it lacks data structures and data abstraction mechanisms. Arctic is also lacking any mechanism for mutual exclusion. These are not so much problems with the approach as a result of focusing research on the interesting aspects of Arctic.

10.3. Directions for Future Research

There are a number of open issues related to the design and implementation of Arctic. One of the major sources of inefficiency in the current language and implementation is the fact that the execution order of expressions must be determined at run-time. One could argue that the execution order should be computed only after the data-flow graph is modified by an instantiation. This would reduce the total computation but would probably increase the worst-case latency.

At least four approaches might be taken to reduce the overhead of determining execution order. First, static analysis might be able to determine that fairly large subgraphs can be compiled as in-line code. Second, language restrictions might make it possible to determine execution order without lazy evaluation. For example, if variables and parameters were eliminated, the resulting dependency graph would be a tree and a depth-first traversal of the tree would generate a legal execution order. Perhaps there are less restrictive alternatives. Third, special hardware support for checking time stamps might provide substantial speedup. Finally, in some situations, it makes sense to cache the execution order and reuse it until the graph changes. In general, this saves total execution time but increases the worst-case latency.

The present implementation spends a fair amount of time testing whether instances have stopped. Unlike signals which tend to change at every time step, the boolean flag indicating that an instance has stopped only changes once. An implementation that propagates this change to dependents could eliminate the need to test for a change on every time step.

The current implementation does not perform storage reclamation. Both reference counting and parallel garbage collection are viable strategies. An interesting problem is how to stop the computation of ‘‘orphan’’ signals that can never affect any output variables. A related problem is the dynamic optimization of Arctic signal computations. For example, if two signals are being multiplied and one of them permanently goes to zero, then the multiplication can be replaced by the constant zero. The present system can successfully perform dynamic optimization, but good

strategies for directing optimization effort have not been studied.

Another area for exploration is the use of multiple clocks and multiple time steps. In many real-time systems, periodic computations (corresponding to Arctic signals) are performed at different frequencies depending upon the latency requirements of each computation. Arctic has no mechanisms for specifying the sample rate of signals and it is not clear how signals computed at different sample rates would be combined. A related area is the specification of time mapping or time deformation, essentially allowing the rate of passage of time to be virtualized and controlled by the program. This is of particular interest in musical applications, but might also be relevant to testing and debugging in other applications.

A ‘‘pure’’ approach is useful for exploring the implications of declarative-style programming for language design and implementation, but this may not be the most effective way to put the resulting techniques into practice. Just as ‘‘pure’’ Lisp is rarely if ever used for serious applications, it may be too much to expect that ‘‘pure’’ Arctic will be used for the serious implementation of real-time systems.

An alternative might be to look for ways that the declarative programming style and behavioral abstraction mechanisms of Arctic can be supported by more conventional imperative languages, run-time systems, and real-time operating systems. One idea is to implement a set of parameterized behaviors that can be instantiated from within an imperative language as in the existing object-oriented implementation. One might then supply an expression and let the run-time system keep the value of the expression up-to-date. Another idea is to extend an imperative language with some degree of lazy evaluation or other mechanism for maintaining one-way constraints.

11. Conclusions

Arctic has introduced a new declarative approach to the specification of temporal behavior. In contrast to imperative languages, Arctic allows timing to be specified explicitly and synchronization to be specified implicitly. One of the interesting aspects of Arctic is its support of behavioral abstraction in which real-time behavior classes called prototypes can be created and instantiated under different transformations. The standard transformations are stretching and shifting in time.

Four implementations and variations of Arctic have now been developed. These are an interpreter that operates on piece-wise linear representations of signals, a language for describing and manipulating musical note sequences, a language for digital signal processing, and a real-time Arctic for processing data at moderate sample rates.

The Arctic approach seems particularly well-suited to systems that synthesize rather complex output. These are applications where there is a need to assemble and schedule complex behaviors. In contrast, systems that test for conditions and respond immediately or that perform fixed arithmetic operations on a stream of samples are easily implemented in almost any language. An application area that has not been explored yet is computer animation. It seems reasonable that a language based on Arctic could provide temporal control over object positions and viewing parameters. In addition, the concept of behavioral abstraction could be applied with additional transformations to alter an object’s size and position as well as the start time and

duration of animated behaviors.

Future work on Arctic will include further development of the real time implementation and a continued effort to bring Arctic concepts into the realm of practical software engineering.

12. Acknowledgments

I am grateful to the School of Computer Science for making it possible to develop Arctic and for soliciting this report. Many people have played an important role in encouraging and supporting my work on Computer Music at Carnegie Mellon, and Arctic could not have developed without their enthusiasm. These include Nico Habermann, Phil Miller, Dana Scott, Raj Reddy, and Marilyn Taft Thomas. Others have worked more directly on Arctic. Paul McAvinney and I spent innumerable hours talking about and designing the first version of Arctic. Dean Rubine implemented the Arctic interpreter, which forced us to nail down a number of fuzzy design details. Chris Fraley implemented the bulk of Fugue, and George Polly provided additional development effort. Hal Mukaino helped design and begin the implementation of real-time Arctic. Frances Dannenberg provided helpful comments on this article.

Support for Arctic has come from DARPA and the Hughes Aircraft Corporation. The real-time Arctic effort and this paper have been supported by a grant from Yamaha Music Technologies, Inc. Their support is greatly appreciated.

References

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

[2] Anderson, David P. and Ron Kuivila.
Accurately Timed Generation of Discrete Musical Events.
Computer Music Journal 10(3):48-56, Fall, 1986.

[3] Ashcroft, E. A. and W. W. Wadge.
Lucid, a Nonprocedural Language with Iteration.
Communications of the ACM 20(7):519-526, July, 1977.

[4] Balaban, Mira.
Music Structures: A Temporal-Hierarchical Representation For Music.
Technical Report FC-TR-021 MCS-313, Ben Gurion University Department of Mathematics and Computer Science, 1989.

[5] Cameron, E. J., D. M. Cohen, B. Gopinath, W. M. Keese II, L. Ness, P. Uppaluru, and J. R. Vollaro.
The IC* Model of Parallel Computation and Programming Environment.
IEEE Transactions on Software Engineering :317-326, March, 1988.

[6] Cointe, Pierre and Xavier Rodet.
Formes: an Object & Time Oriented System for Music Composition and Synthesis.
In *1984 Symposium on LISP and Functional Programming*, pages 85-95. ACM, 1984.

[7] Collinge, D. J.
MOXIE: A Language for Computer Music Performance.
In *Proceedings of the 1984 International Computer Music Conference*, pages 217-220. Computer Music Association, 1984.

[8] Dannenberg, Frances K., Roger B. Dannenberg, and Philip Miller.
Teaching Programming to Musicians.
In *1984 Proceedings of the Fourth Annual Symposium on Small Computers in the Arts*, pages 114-122. IEEE, October, 1984.

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

[10] Dannenberg, Roger B.
The Canon Score Language.
Computer Music Journal 13(1):47-56, Spring, 1989.

[11] Dannenberg, Roger B. and Christopher Lee Fraley.
Fugue: Composition and Sound Synthesis With Lazy Evaluation and Behavioral Abstraction.
In *Proceedings of the 1989 International Computer Music Conference*, pages 76-79. Computer Music Association, San Francisco, 1989.

[12] Greenberg, Gary.
Procedural Composition.
In *Proceedings of the 1987 International Computer Music Conference*, pages 25-32. Computer Music Association, 1987.

[13] Greenberg, Gary.
Composing With Performer Objects.
In *Proceedings of the 1988 International Computer Music Conference*, pages 142-149. Computer Music Association, 1988.

[14] Mathews, Max V.
The Technology of Computer Music.
MIT Press, Boston, 1969.

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

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

[17] Rubine, Dean and Roger B. Dannenberg.
Arctic Programmer's Manual and Tutorial.
Technical Report CMU-CS-87-110, Carnegie Mellon, 1987.

[18] Scaletti, C., and E. Johnson.
An Interactive Graphic Environment for Object-Oriented Music Composition and Sound Synthesis.
In *Proceedings of the 1988 Conference on Object-Oriented Languages and Systems*, pages 18-26. ACM, 1988.

[19] Schottstaedt, Bill.
Pla: A Composer's Idea of a Language.
Computer Music Journal 7(1):11-20, Spring, 1983.