Arctic: A Functional Language for Real-Time Control

Roger B. Dannenberg

Computer Science Department
Carnegie-Mellon University
Pittsburgh, PA 15213

Abstract

Arctic is a language for the specification and implementation of real-time control systems. Unlike more conventional languages for real-time control, which emphasize concurrency, Arctic is a stateless language in which the relationships between system inputs, outputs and intermediate terms are expressed as operations on time-varying functions. Arctic allows discrete events or conditions to invoke and modify responses asynchronously, but because programs have no state, synchronization problems are greatly simplified. Furthermore, Arctic programs are non-sequential, and the timing of system responses is notated explicitly. This eliminates the need for the programmer to be concerned with the execution sequence, which accounts for much of the difficulty in real-time programming.

1. Introduction

Real-time computer systems are generally regarded to be the most difficult to program [9]. We are especially interested in the class of real-time systems that must respond to inputs quickly (on the order of 1ms) but require a considerable amount of logic or decision-making. We can characterize this class roughly as falling between two extremes in the spectrum of response times. For systems that must respond much faster, special-purpose hardware is required, and programming is not the limiting factor. Signal processing falls into this category. At the other extreme are "soft" real-time systems that need not respond so quickly. Conventional multi-tasking systems or even general-purpose operating systems are adequate for the implementation of these systems. An inventory control system falls into this category.

Between these extremes lie a number of applications, including industrial automation, transportation systems, robots, consumer products, animation, and computer music. The present work was motivated directly by the demand for a flexible yet powerful language that could meet the real time constraints of computer music applications. I am currently looking at other possibilities, such as the control and instrumentation of physical experiments, animation and, job control. Typically, these systems must deal with many concurrent and interacting tasks, and the programmer is faced with several problems, including concurrency, synchronization, and timing.

The first of these problems is concurrency: the programmer must multiplex the processor(s) among a number of tasks. Often, the overhead of context switching between processes would be too high, so the programmer must coalesce his tasks into a loop within a single process. (See Figure 1). This transformation, of course, is not necessarily straightforward.

```
cobegin                    while true do
    Task1;                     begin
    Task2;        ------>        do_part_of_Task1;
    Task3                        save_Task1_state;
coend                            do_part_of_Task2;
                                 save_Task2_state;
                                 do_part_of_Task3;
                                 save_Task3_state
                               end
```

Figure 1 : Transformation of three parallel tasks to eliminate multiple processes.

In any case, the programmer must formulate his concurrent tasks to produce real-time responses in tiny steps so that the tasks can be multiplexed at a fine granularity.

The second problem is synchronization. Whether the programmer uses multiple processes, or simply performs tasks in sequence, he must make sure that data is available before it is used. For programs that sample inputs, perform operations, and produce sampled outputs, the appropriate model of computation is a data-flow graph rather than a collection of processes. Synchronization mechanisms are also needed so that tasks can be started and stopped by various events and conditions.

The third problem is timing. The real-time system program does more than merely compute values; it must also read and write them at the proper time. Proper timing is usually achieved as a side-effect of sequential execution. For example, if a program discovers it must do operation *X* at some future time *t*, it can either wait until *t*, or it can schedule *X* and do other things until time *t*. Either way, it is the sequential combination of a wait (or doing other things) *followed by X* that convinces us *X* will happen at time *t*. Often, the execution sequence with which a program could most directly compute values bears little relationship to the time sequence in which the values are needed. I conjecture that this is the largest single source of complexity in real-time programs.

The development of the programming language Arctic is motivated by all three of these problems. In Arctic, values have a time dimension; that is, an Arctic value can be viewed as a real-valued function of time.[1] Thus, *Arctic allows us to manage concurrently varying values as succinctly as conventional languages allow us to deal with several variables.* Concurrency is a natural by-product of the functional properties of the language, rather than a language control construct.

Arctic also simplifies the problem of synchronization for the programmer. *Since Arctic programs have no state or imperative commands* [3], *synchronization is unnecessary to govern the order of execution.* With regard to synchronization, Arctic has properties similar to those of data-flow languages [14].

The problems of timing are addressed in Arctic by a complete and fairly explicit specification of exactly when things happen. If we want an event *X* to take place at time *t*, we write *X@t*, meaning "do *X* at time *t*". The formal meaning of this expression will be explained further below.

So far, I have described some problems of real-time control and hinted at a language, Arctic, that offers some solutions to them. Below, I will discuss the principle components of Arctic, including how responses are defined, the specification of timing relationships, and asynchronous event handling. I conclude with a discussion of some properties of Arctic.

2. Previous Work

Arctic synthesizes ideas from functional programming languages and several languages and systems for computer music. The principal advantage of functional languages for real-time control is the simplicity of concurrent evaluation due to the absence of side-effects. Data-flow languages, which allow a restricted form of assignment operation, are also amenable to concurrent execution [14, 5]. These languages are interesting in that all parallelism and synchronization are implicit, as opposed to imperative languages, which introduce explicit processes and synchronization constructs for concurrency.

However, functional and data-flow languages achieve their nice properties in part by suppressing the notions of time and sequence. Arctic regains the ability to deal with time by virtue of its primitive values, which are themselves functions of time. This is reminiscent of streams in data-flow languages [6], and sequences in Lucid [2]. However, there is a fundamental difference between Arctic values and these streams or sequences. While Arctic values are conceptually continuous and are "indexed" or referenced by global time, streams and sequences are indexed and aligned by position relative to the beginning of the particular stream or sequence.

Arctic also borrows heavily from several previous music-oriented languages, including Music V [11], the GROOVE system [12], and 4CED [1]. Music V is not a real-time control language, but it does allow one to write programs whose output is a time-varying signal. Music V uses data flow-like programs to describe *instruments* and separate event lists (*scores*) to invoke instances of instruments at specified times. Instruments cannot invoke other instruments, however. The GROOVE system introduced the idea of function manipulation for real-time control, but had no notion of responding to discrete events. On the other hand, 4CED includes events (*attach points*), and the response to an event can in turn cause another event, but 4CED does not provide function manipulation as a primitive operation. The 4CED language allows the time of execution of statements to be notated explicitly; however, the language is largely sequential to simplify its implementation.

Several researchers have adopted the object-oriented programming style for the real-time control of music [16, 17] and animation [10, 15]. These systems typically build a structure at run-time containing a set of objects which respond to "tick" messages. The objects serve to encapsulate tasks, and each "tick" message causes a task to update its state to reflect the passage of one quantum of time. Thus, the concurrency problem illustrated in Figure 1 is still present, although it is mediated by the abstraction facilities of object-oriented languages. The programmer must be concerned with synchronization. For example, if one object uses data produced by another, then the order of "tick" messages is important.

The OWL language [8] falls somewhere between the functional approach of Arctic and the more traditional process-oriented languages [19, 7, 13]. OWL programs manipulate state like procedural languages, but OWL has implicit looping and processes that facilitate a non-sequential, event- and condition-driven style of programming.

3. Preliminaries

Before describing Arctic, it is helpful to consider the formal model of real-time systems on which Arctic is based. For any application, a real-time system is modeled as a "black box" that has a set of inputs and a set of outputs. (See Figure 3.) These may be continuous (representing, say, a temperature reading or a

[1] Arctic values have some additional attributes, but we can safely ignore them for the time being.

voltage output), in which case the input or output is modeled by a real-valued function of time. An input or output may also be a discrete event (perhaps representing a contact closure), in which case the model is a set of times at which the event occurs.

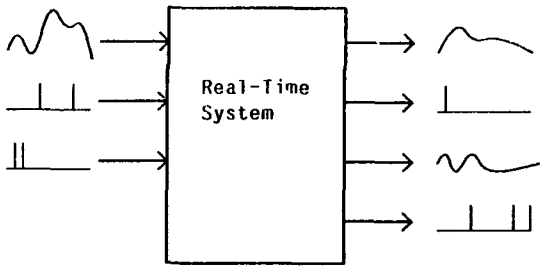

Figure 3 : Schematic of a real-time system having continuous and discrete event inputs and outputs.

An Arctic program is a description of the real-time response of a system to its continuous and discrete event inputs. Formally, an Arctic program denotes a higher-order function from the set of inputs to the set of outputs. Because it is impossible to measure or represent real input values or times with infinite precision, any Arctic implementation will only approximate the ideal, just as floating-point numbers and operations are only an approximation to real arithmetic.

4. Prototypes

The primary structuring mechanism in Arctic is the *prototype*, which is roughly analogous to a process, procedure, or function in other languages. A prototype is more properly regarded as a specification for a response to a class of events. An event causes an *instantiation* of the corresponding prototype, and the specified response ensues. The nature of this response can be a function of the *time* of instantiation (the time of the event) and the *duration* of the instantiation. If events happen at times t_1 and t_2, then we get two (possibly concurrent) responses, one parameterized by t_1 and the other by t_2. (The duration parameter, and a third parameter, *terminate* will be discussed later.)

Let us consider an example. Suppose we want to implement a doorbell that does not ring between the hours of midnight and 8 AM. The system will have one event input called *Push* which occurs whenever the button for the doorbell is pushed. The output of the system is an event, *RingBell*, which is interfaced to an electrical bell (see Figure 4).

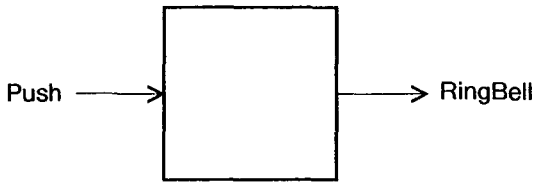

Figure 4 : The doorbell system.

The Arctic program that describes the response of this system is as follows:

```
Push causes [
    if (time mod 24 hours) > 8 hours
    then RingBell
    ]
```

The first line declares *Push* to be a prototype. Since *Push* is a system input, the *Push* prototype will be instantiated whenever a *Push* event occurs. The parameters *time* and *dur* are implicitly declared for every prototype. Within the body of *Push*, *time* denotes the real-valued time of instantiation of the *Push* prototype. Similarly, *dur* denotes the stretch factor, which is unity by default.

The next line of the example is simply a test to see if the time is after 8 AM. If so, the **then** part says to instantiate *RingBell*. Since *RingBell* is a system output, this causes an output event which, in this system, rings the doorbell.

Although the syntactic constructs of Arctic look like expressions and statements of more conventional languages, they actually denote prototypes, which yield Arctic values when instantiated. Thus, a *Push* event causes an instance of the *Push* prototype, which is defined to be an instance of the prototype enclosed in brackets ([...]). This happens to be an **if** prototype, which denotes an instance of either its **then** part or its **else** part, depending upon the value yielded by an instance of the prototype between **if** and **then**.

5. Shift and Stretch

By now, the reader is probably wondering about the purpose of the implicit *time* and *dur* parameters. These are used to specify when prototypes are instantiated, that is, when things happen. In the previous example, everything happens at the time of the *Push* event. I purposely glossed over the aspects of Arctic semantics that determine this. But now, suppose we want to implement a system where the response may involve a sequence of carefully timed outputs. To illustrate this, we will extend our doorbell to play a familiar tune. As with the previous example, we will first write the program, and then peek at the underlying semantics to see why it works.

To extend the doorbell, we need a set of four event outputs interfaced to four electrically operated tubular bells (chimes) as shown in Figure 5.

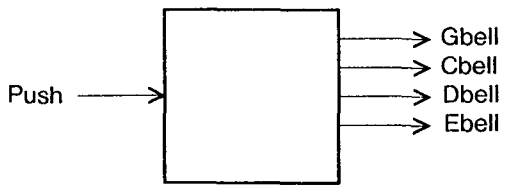

Figure 5 : The extended doorbell system.

The program is as follows:

```
Push causes [
    if (time mod 24 hours) > 8 hours
    then BigBen
    ]

BigBen causes [
    Ebell @ 0;
    Cbell @ 1;
    Dbell @ 2;
    Gbell @ 3;
    Gbell @ 8;
    Dbell @ 9;
    Ebell @ 10;
    Cbell @ 11
    ]
```

In this program, we have replaced the instantiation of *RingBell* within *Push* by an instantiation of *BigBen*, which is not an output event, but a program-defined prototype. When instantiated, *BigBen* plays a melody by generating a sequence of output events. An *Ebell* event will occur at 0 seconds, *relative* to the instantiation of *BigBen*. This is followed by a *Cbell* event at 1 second, a *Dbell* event at 2 seconds and so on. Because the timing is given explicitly, the lexical ordering of events is arbitrary.

Now let us look at why this sequence of outputs will occur. As mentioned above, all prototypes have an implicit time and duration parameter. Thus, the symbol *BigBen* within the *Push* prototype is really a syntactic shorthand for *BigBen(time, dur)*, where *time* and *dur* are the parameters of *Push*. Therefore, an *instantiation* ordinarily "inherits" the time and duration of its *instantiator*. However, the programmer can modify the time of instantiation in a straightforward way using the shift ("@") and stretch ("~") operators.

The shift operator is used to alter the instantiation time. If we write out the time and duration parameters explicitly, the following expression holds for any prototype *P*:

$$(P\ @X)(time, dur) = P(time + X{\cdot}dur, dur)$$

Notice that the time of instantiation is increased by the product of *X* and the duration factor. The rationale for this definition should become clear later.

Similarly, the stretch operator can be used to alter the duration parameter:

$$(P \sim X)(time, dur) = P(time, X{\cdot}dur)$$

Notice that $P\ @\ 0 = P \sim 1 = P$. Also, shift and stretch can be nested.

Returning to our example, observe that the form of the *BigBen* prototype is [*A*;*B*;*C*;...], where *A*,*B*,*C*,... are prototypes. This is a *collection* construct, and is instantiated by instantiating each of the component prototypes in parallel. The *time* and *dur* attributes for each component are the same as for the collection prototype. Thus, instantiation time and duration factors are "inherited". We can now see that the time of the bell events output by the *BigBen* prototype will be the sum of the *time* parameter of the instance of *BigBen* and the constants indicated after each bell event, which serve to shift the time parameter. Thus, the last *Cbell* event will occur 11 seconds after the instantiation of *BigBen*.

To illustrate the stretch operation, suppose we decide that an 11-second doorbell is a bit too long and we want to speed things up. We could modify the *BigBen* prototype by decreasing all of the shift values proportionally. Alternatively, we could replace the instantiation of *BigBen* in *Push* by *BigBen~0.5* which would have the effect of playing the melody in half the time. To convince yourself of this, notice that in the definition of *P@X* the time is shifted by *X*•*dur*, where *dur* is the duration factor (0.5 in this case).

Thus, prototypes like *BigBen* not only encapsulate behaviors, they also create a scope subject to time transformations. The temporal specification of events is very flexible and economic in Arctic because *time* and *dur* attributes are inherited by components of collections, and because *shift* (@) and *stretch* (~) compose.

6. Sequences and Attributes

Although I argued earlier against sequential *programming*, it is still important to be able to produce sequential *values* or to define responses as sequences of more primitive ones. In Arctic, prototypes can be instantiated sequentially using the *sequence* construct:

$$[A \mid B \mid C \ldots]$$

which means "instantiate *A*, then *B*, then *C*, and so on." The question is: how long after the instantiation of *A* do we instantiate *B*? To answer this, we must learn about attributes. Every instantiation yields a (possibly void) value and two attributes, *start* and *stop*. The start attribute is ordinarily equal to *time* and represents the starting time of the response produced by a prototype. The *stop* attribute indicates the time at which the response finishes. The *stop* attribute is defined for all primitive prototypes, some of which will be described later. For the *collection* construct described in the previous section, the *stop* attribute is the maximum *stop* attribute of any component prototype; however, this default can be overridden with the use of the **end** construct. For example, if *X* is some expression, and the following collection is instantiated at time *time* with duration factor *dur*:

$$[A; B; \textbf{end}\ @\ X]$$

then the *stop* attribute is (by definition) $time + X{\cdot}dur$.

Now we are prepared to specify the semantics of the sequence construct: In a sequence, the time of instantiation of the (lexically) first prototype is the instantiation time of the sequence. The time of instantiation for each successive prototype is the *stop* attribute of the instantiation of the previous prototype in the sequence. The *stop* attribute of the sequence is the *stop* attribute of the instantiation of the (lexically) final prototype in the sequence. The duration factor for each prototype in the sequence is the duration factor of the sequence itself.

7. Values

Thus far, only discrete inputs and outputs have been described. Arctic also allows the specification of real-time systems with continuous, time-varying values. Formally, a value is a triple ⟨f, s, e⟩, where f is either **void** or a function from a half-open interval of the reals (a time interval) into the reals, s is the start attribute, and e is the stop attribute. These values may be used in arithmetic expressions, assigned to variables (under the single assignment rule [14]), or passed as parameters to prototypes. For example, a simple linear amplifier may be implemented by the following:

$$[output := input * gain],$$

where *input* is a signal input, *gain* is the gain control input, and *output* is the system output.

A set of simple prototypes are predefined and serve as building blocks for defining more complex ones. For example, *ramp* yields a function defined on the interval (*time*, *time* + *dur*] and whose value at *t* is (*t* − *time*)/*dur*. The *sin* prototype takes two arguments, *f* and *p*, returning

$$sin(2\pi t \cdot \int_{time}^{t} f(x)dx + p(t))$$

for *t* in (*time*, *time* + *dur*]. Thus, *sin* returns a *sin* function whose instantaneous frequency and phase are given by functions *f* and *p*. The *extract* prototype takes three arguments, *f*, *a*, and *b*, and returns a function which is equal to *f*(*t*) for *t* in (*a*,*b*] and zero otherwise.

8. Conditionals

Arctic has three types of boolean expressions corresponding to three conditional constructs in the language. The type of expression is easily determined by context. The first type was seen in the *Push* prototype in connection with the **if-then-else** construct. This type returns a boolean constant and corresponds closely to conditionals in conventional programming languages.

The second type returns a boolean function of time, which can be used in the *conditional* prototype, which has the form:

$$(C\ ?\ A : B).$$

An instance of the conditional prototype is a function on the interval (*time*, *time* + *dur*] that is equal to *A*(*t*) if *C*(*t*) is true, and *B*(*t*) if *C*(*t*) is false. An example using this form of conditional will be given in Section 10.

The third type of boolean expression is the conditional event, which can used with the *until* construct described in the next section. In this context, a boolean function is evaluated to find the first moment after *time* at which the function is true. This is then taken as the event time. In this way, conditions can give rise to events and prototype instantiations, making demons trivial to implement.

9. Asynchronous Events

An asynchronous event is one that occurs at an arbitrary time and has some effect on a response already in progress. Arctic uses a special construct to define prototypes that depend upon events or conditions. Suppose we want a function that increases smoothly from 0 to 1 and then holds constant until the *quit* event occurs. This could be written:

Go **causes** [*ramp* | (*1* **until** *quit* **then** *0*)].

Instances of *Go* are illustrated in Figure (9). The **until** construct is defined as follows: consider the prototype *P* **until** *C* **then** *Q*, where *P* and *Q* are prototypes, *C* is an event, and let *P.stop* be the stop attribute of an instance of *P*. Now, if *C* occurs at time *t*, and *t* is between *time* and *P.stop*, then *P* **until** *C* **then** *Q* is equivalent to:

$$extract(P, -\infty, t) + Q\ @\ ((t - time)/dur).$$

Thus, the effect is to combine (by addition) the value of an instance of P up to time *t* with an instance of Q at time *t*. In the definition, *extract* refers to the prototype defined at the end of Section 7, and the expression ((*t* − *time*) / *dur*) is used to instantiate Q at *absolute* time *t* (substitute (*t* − *time*) / *dur* for X in the definition of "@" in Section 5). On the other hand, if *C* does not occur before *P.stop*, the **until** construct is equivalent to *P*, and *Q* is not instantiated.

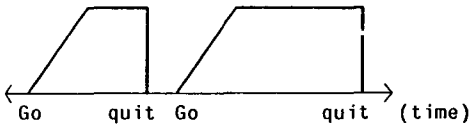

Figure 9 : Instances of the *Go* prototype, illustrating asynchronous responses.

This informal definition of **until** can be formalized and extended to cover cases where *P* in turn causes many instantiations. This is accomplished by attaching yet another implicit parameter, or inherited attribute, to each instance to specify a termination time. Primitives are then defined to end at the termination time if it occurs earlier than *time* + *dur*. For example, the ramp primitive is now defined in terms of three (implicit) parameters: *time*, *dur*, and *terminate*. If *time* < *terminate* < *time* + *dur* then ramp is defined on the interval (*time*, *terminate*], and its value at *t* is (as before) (*t* − *time*) / *dur*. Otherwise, ramp is defined by the same expression, but on the interval (*time*, *time* + *dur*].

Other primitives are defined in a similar manner, and the collection and sequence constructs are defined to pass this *terminate* attribute on to their component prototypes.

10. Example

The following program implements the "musical sieve" of Cointe and Rodet [4], whose original program is presented in the object-oriented language Formes. The task is to produce a time-varying pattern in parallel with time-varying data. The output of the program is *f1*, a function of time. We will use a sequence of prototypes to implement the pattern and data:

```
out f1;
value data;

C is [100 * unit ~ 0.2]
E is [130 * unit ~ 0.2]

noC is [(data = 100 ? 140 : data) ~ 0.3]

noE is [(data = 130 ? 140 : data) ~ 0.3]

Go causes [data := [E | E | C];
           f1 := [noC | noE | noC]];
```

The main prototype, *Go*, assigns *data* to the sequence of prototypes E, E, and C, which in turn are defined to be rectangular pulses of two different heights, and 0.2 sec in duration. In parallel, a sequence of pattern prototypes is instantiated to compute the output *f1*. The pattern *noC* examines *data*, returning 140 if *data* is equal to 100 (the height of the pulse made by the prototype *C*), and otherwise returning the current value of data. The *noC* prototype operates on an interval that is 0.3 seconds in duration, achieved by instantiating the conditional expression with duration factor 0.3. The *noE* prototype is similar to *noC* except that it rejects values of 130 (the height of the *E* pulse) rather than 100. Figure 10 illustrates *data* and the intervals of time during which *E* and *C* prototypes are instantiated. The lower half of the figure illustrates the output *f1* and the intervals during which *noC* and *noE* prototypes are instantiated.

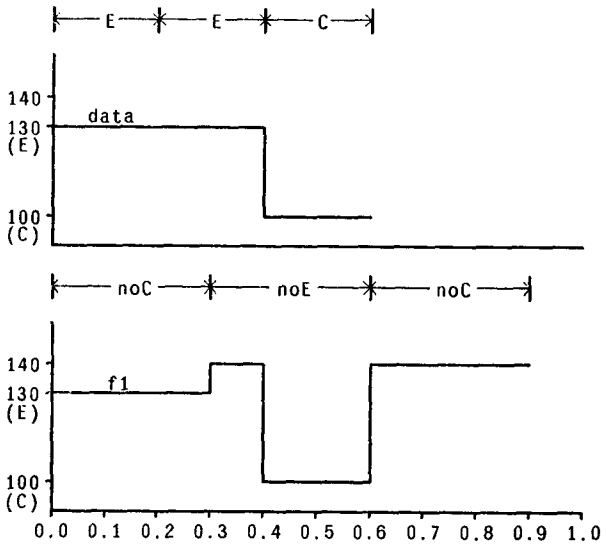

Figure 10 : Graph of the intervals of *E* and *C* prototype instantiation, *data* the intervals of *noC* and *noE* instantiations, and the output *f1*.

10.1. Functions vs. Prototypes

This example illustrates the difference between Arctic values (functions of time) such as *data* and *f1* and Arctic prototypes such as *unit* and *C*. Wherever the name of a value is mentioned in the program, the name denotes a single function of time. If value names are changed to eliminate conflicts ordinarily handled by static scope rules, then a value name denotes the same function everywhere in the program, independent of implicit *time*, *dur*, and *terminate* parameters. In other words, shift and stretch operators have no effect upon Arctic values.

Prototypes, on the other hand, are higher order functions of the implicit parameters *time*, *dur*, and *terminate*, as well as possible explicit parameters. Each occurrence of a prototype name in the program denotes a new instance of the named prototype. For example, in the sieve program above, the symbol "*unit*" denotes an instance of the *unit* prototype. Each occurrence of "*unit*" can denote a potentially new function since *time*, *dur*, and *terminate* may be different for each. An instance may produce an Arctic value.

11. Discussion

Arctic derives expressive power from several sources. First, functions of time can be combined and manipulated as a whole, as opposed to viewing functions as streams of values that must be dealt with individually. Second, there is an instantiation mechanism through which discrete events or conditions can give rise to complex responses. Another source of expressive power is the fact that the time of instantiation can be explicitly controlled. Timing is achieved by direct specification in the program, in contrast to languages where timing is a consequence of the sequential execution of many instruction streams. Specification is streamlined by the use of implicit parameters, which are "inherited" by components of constructs and can be transformed using shift and stretch operators. Finally, Arctic "variables" obey the single assignment rule; thus, synchronization between concurrent prototypes that define values and those that use them is implicit and need not concern the programmer.

11.1. Side Effects

In some cases, however, a pure side-effect free language is awkward. Consider the case where many instances of prototypes contribute to an output, such as the first doorbell example. Here, the *RingBell* output of the system must include one event for each instance of *RingBell* caused by *Push*. Thus, we have to combine an arbitrary number of instances to form a single output. The problem also occurs with Arctic values and continuous outputs. For example, in the signal processing aspects of computer music, one often wants to synthesize several instruments independently and add their sounds together. For efficiency, the synthesis algorithm should instantiate instrument prototypes dynamically, so the number of signals to sum will vary with time. A static expression such as "A + B + C" cannot express this summation. Restricted forms of side effects are used to solve these problems.

For the problem of discrete events, exemplified by *RingBell*, the combination of events is implicit, and two instances of *Push* (after 8 AM) will ring the bell twice, just as two occurrences of *C* in [*C* | *C*] denote two sequential instances of the prototype *C*. This may be considered a form of side effect.

In the case of continuous functions, the problem is solved by declaring a value or output to be a **sum** or **product**. Then, an instance can contribute an addend or factor using a construct like:

out + = *expression*.

The value of *out* at time *t* is the sum of all *x(t)*, such that *x(t)* is defined, and *out* + = *x* is instantiated. Thus, *out* has the same value throughout the program, and it is not possible for a prototype to look at *out* and then modify it in any way[2]. This is what one would expect from a single-assignment language. On the other hand, the prototype *[P; P]* may not mean the same thing as *[P]*, because each instance of *P* may contribute to some **sum** variable. Thus, several restricted forms of side effects are possible in Arctic.

11.2. Implementation

Arctic programs have been hand compiled to calls on function manipulation procedures, and a non-real-time interpreter is under development. These initial implementations assume that the inputs are completely known before program execution begins. An interactive graphics program can be used to draw and manipulate functions of time before sending them as inputs to the Arctic interpreter. For these programs, continuous functions are approximated by piece-wise linear functions, which are represented by lists of coordinates of the inflection points.

I am currently designing a true real-time implementation, in which a sequential processor will be multiplexed rapidly among a large number of concurrent instances. For this implementation, Arctic values will be represented by their current value, and primitive prototype instances will be represented by a few words of state and parameter information. The calculations to be performed will be represented as a linked list of primitive operations, which will be ordered according to precedence constraints implied by the single assignment rule. The instantiation of a prototype will add operations to the list. Secondary structures will be necessary to support the **until** construct which has the effect of removing operations from the list asynchronously. I believe that Arctic programs can be executed without conventional processes or context switches, which would be prohibitively expensive for the applications of interest. This will make Arctic competitive with more conventional real-time program structures.

I am also considering the problems of mapping Arctic programs onto highly parallel, reconfigurable arrays of processors, such as the UPE (Universal Processing Element) proposed by Carver Mead [18]. In this scheme, a small UPE processor would be allocated to each arithmetic or logical operation that resulted from an Arctic instantiation. Data would move synchronously through an interconnection network from one UPE to the next. Although many problems still need to be solved, a single board VLSI implementation could perform several hundred million operations per second and make Arctic applicable in the realm of signal processing.

12. Conclusion

Arctic is a powerful language for the *description* of real-time systems. Its development has led to a clearer understanding of how real-time systems should be specified; for example, it is useful to model a time-varying input as a real-valued function of time even if the input is implemented as a stream of discrete samples. It is my belief that Arctic will also be a powerful tool for the *implementation* of real-time systems. My colleagues and I are currently building experimental systems to test this hypothesis.

13. Acknowledgments

I would like to thank Paul McAvinney, who contributed to the formulation of Arctic and who has implemented a large part of our experimental system. I am also happy to acknowledge the assistance of Dean Rubine, who is writing the Arctic interpreter.

[2] Of course, the value of *out* at some time greater than *t* may be a function of the value of *out* at time *t*.

References

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

2. E. A. Ashcroft and W. W. Wadge. "Lucid, a Nonprocedural Language with Iteration." *Communications of the ACM 20*, 7 (July 1977), 519-526.

3. John Backus. "Can Programming Be Liberated from the von Neumann Style?" *Communications of the ACM 21*, 8 (August 1978), 613-641.

4. Pierre Cointe and Xavier Rodet. Formes: an Object & Time Oriented System for Music Composition and Synthesis. 1984 Symposium on LISP and Functional Programming, ACM, 1984.

5. Jack B. Dennis. "Data flow supercomputers." *Computer 13*, 11 (November 1980), 48-56.

6. Jack B. Dennis and Ken K.-S. Weng. An Abstract Implementation For Concurrent Computation With Streams. Proceedings of the 1979 International Conference on Parallel Processing, IEEE Computer Society, 1979.

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

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

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

10. K. Kahn. DIRECTOR Guide. Tech. Rept. MIT AI Laboratory Memo 482B, MIT, December, 1979.

11. Max V. Mathews. *The Technology of Computer Music.* MIT Press, Boston, 1969.

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

13. David May. "OCCAM." *Sigplan Notices 18*, 4 (April 1983), 69-79.

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

15. C. U. Reynolds. Computer Animation with Scripts and Actors. Proceedings of ACM SIGGRAPH Conference, ACM, July, 1982.

16. X. Rodet, P. Cointe, J. B. Barriere, Y. Potard, B. Serpette, and J. P. Briot. Applications and Developments of the FORMES programming environment. Proceedings of the 1983 International Computer Music Conference, Computer Music Association, 1983.

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

18. John Wawrzynek and Tzu-Mu Lin. A Bit Serial Architecture for Multiplication and Interpolation. Tech. Rept. 5067:DF:83, California Institute of Technology, January, 1983.

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