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.

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.
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.