Multiparadigm Research: A Survey of Nine Projects

This issue of *IEEE Software* showcases six research projects studying multiparadigm programming languages and environments. This special section provides an overview of nine more projects in the field of multiparadigm systems. All the research is concerned with providing the "right" set of constructs for the programmer, allowing the programmer to use more than one mode of thinking (paradigm) for complex problems, but the approach varies widely from project to project.

The international collection of research projects (and paradigms) described in this section are

- Arctic (functional, real-time), Carnegie-Mellon University;
- C++ (imperative, object-oriented), AT&T Bell Laboratories;
- CaseDE (imperative, specification), Case Western Reserve University;
- Lore (object-oriented, set-based), CGE Research Center (France);
- Orient84/K (object-oriented, rule-based, access-oriented, parallel), Keio University (Japan);
- Smallworld (imperative, object-oriented), IBM Research;
- Tablog (functional, rule-based), IBM Research, Stanford University, Weizmann Institute (Israel), and SRI International;
- Algebraic specifications in Prolog (specifications, rule-based), GTE Laboratories; and
- Integrating functional and logic programming (functional, rule-based), University of Utah.

Each summary includes the name and address of someone who can be contacted for further information. Some also include references to published work in the field. We hope that this will serve as the beginning of a forum for the exchange of ideas in the field.

Brent Hailpern
IBM Thomas J. Watson Research Center
Yorktown Heights, NY 10598

Arctic: A Functional Language for Real-Time Control

Arctic is a language for the specification and implementation of real-time control systems. Its model of a real-time control system has both continuous and discrete inputs and outputs. Continuous inputs, outputs, and intermediate values are treated as functions of time. An Arctic program, which has no state, specifies a mapping from the input functions and discrete input events onto the output functions and discrete output events.

Arctic is intended for those real-time systems that must respond somewhat quickly to inputs (on the order of one millisecond) but require a considerable amount of logic or decision making. Possible applications include industrial automation, robots, transportation systems, animation, and computer music. A typical architecture for such systems is that of a general-purpose processor interfaced to special-purpose hardware.

Arctic programs do not specify the low-level processing and control that goes on inside the special-purpose hardware. This processing normally implements a rigid algorithm with stringent real-time requirements. Arctic programs do specify how the signals that control the hardware change in response to input from other hardware. An example of special-purpose hardware is a digital filter that must output a result every 20 microseconds. An Arctic program could map the setting of a certain knob to a value for each filter parameter.

The programmer of a real-time system is faced with several problems, including concurrency, synchronization, and timing. Arctic approaches each of these problems differently than a conventional real-time control language (such as Ada).

In a conventional language, concurrency is expressed by creating multiple processes that change state (conceptually) in parallel. Programmers of real-time systems often forgo the luxury of processes, merging code into a single process for greater efficiency. This tends to obscure the structure of the program and makes programming more difficult.

In Arctic, however, values have a time dimension—that is, they can be viewed as functions of time. Many functions can be defined on overlapping time intervals to express concurrency. Thus, Arctic allows us to manage concurrently varying values as succinctly as conventional languages allow us to deal with several scalar variables. Concurrency is a natural by-

Because of their symbolic significance, mythical beasts are favorite images for heraldry.

product of the functional properties of the language, rather than a language control construct.

The most important use of synchronization in multitasking languages is to make sure data has been computed by one task before it is used by another. Since Arctic programs have no state or imperative commands, synchronization is unnecessary to govern the order of execution. This order depends only on the dependencies in the Arctic program, not on any sequential ordering of statements. In this respect, Arctic is similar to dataflow languages.

Similarly, timing in conventional real-time languages is usually achieved as a side effect of sequential execution. To do *X* at some time *t*, we must wait until *t* or schedule *X* and do other things until time *t*. Time is much more explicit in Arctic. Informally, in Arctic we write "X@t" to have event *X* take place at time *t*. Similarly, "X ~ d" asks event *X* to take *d* times as long as usual.

Now that the principles underlying Arctic have been discussed, here are two examples aimed at giving the reader a feel for the language.

```
in LeftInput, RightInput, Gain;
out LeftOutput, RightOutput;

Go causes [ LeftOutput
          : = LeftInput*Gain;
        RightOutput
          : = RightInput*Gain ];
```

This example specifies a stereo amplifier. Three inputs—LeftInput, RightInput, and Gain—are declared to be functions of time. Go is a discrete event that occurs when an Arctic program is started. The line "Go causes..." is a prototype, a piece of code that specifies a response to a discrete event. A copy of the prototype is instantiated (executed) each time the event occurs. In this example, only one copy of Go is ever instantiated, and it causes the output functions LeftOutput and RightOutput to be computed as the product of the inputs Gain and LeftInput or RightInput, respectively. Since values denote functions of time, there is no need to write looping or multiplexing between tasks.

```
BigBen causes [Ebell@0;Cbell@1;
               Dbell@2;Gbell@3];
PushDoorbell causes BigBen;
```

The discrete input PushDoorbell causes the discrete even BigBen to be instantiated, which in turn causes the four events Ebell, Cbell, Dbell, and Gbell to be instantiated at times 0, 1, 2, and 3 seconds after BigBen was instantiated, respectively, thus playing a familiar tune. If instead we write (read " ~ " as "stretch")

```
PushDoorbell causes BigBen ~ 0.25 @ 2;
```

the tune will not begin until two seconds after the button is pushed, and then will be played four times as fast.

Artic derives its expressive power from several sources. First, functions of time can be combined and manipulated as a whole, as opposed to implementing functions as scalars whose values must be repeatedly updated individually. Second, there is an instantiation mechanism through which discrete events can give rise to complex responses. Third, the time of instantiation can be explicitly controlled. As hinted at in the BigBen example, specification can be streamlined by the use of implicit start time and duration parameters, which are inherited 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.

Roger Dannenberg and Dean Rubine
Computer Science Department
Carnegie-Mellon University
Pittsburgh, PA 15213

C++ Programming Language

The C++ programming language was designed to make the task of programming more enjoyable for the serious programmer. It consists of a traditional systems programming language and a set of mechanisms for defining new data types. The language, which is defined in the book *The C++ Programming Language*,[1] supports data abstraction and object-based programming in addition to traditional programming techniques.

C++ has the C programming language[2] as a subset, so it can be used immediately by C programmers. It can use C programs, libraries, and support tools and provides many facilities that make classical C programming simpler and safer without sacrificing runtime efficiency or increasing program size. For example, it provides function argument type checking, scoped manifest constants, and operators for handling free store.

The main strength of C++, however, lies in its facilities for defining new types. It provides a Simula-like[3] single-inheritance class concept with static and optional dynamic type checking. A programmer can define new general-purpose and application-specific types that can be used as elegantly, and often as efficiently, as the built-in types. Operators (including assignment and subscripting) can be overloaded, and initialization and cleanup functions can be specified. Examples of user-defined types are complex numbers, matrices, vectors with dynamic range checking, character strings with as substring operator, and linked lists.

More ambitious uses of C++ have involved defining sets of related classes (types), providing a significantly enhanced programming environment, and, in effect, embedding a new, higher level programming language in C++.