Design Languages for Embedded Systems

Stephen A. Edwards
Columbia University, New York
sedwards@cs.columbia.edu

May, 2003

Abstract

Embedded systems are application-specific computers that interact with the physical world. Each has a diverse set of tasks to perform, and although a very flexible language might be able to handle all of them, instead a variety of problem-domain-specific languages have evolved that are easier to write, analyze, and compile.

This paper surveys some of the more important languages, introducing their central ideas quickly without going into detail. A small example of each is included.

1 Introduction

An embedded system is a computer masquerading as a non-computer that must perform a small set of tasks cheaply and efficiently. A typical system might have communication, signal processing, and user interface tasks to perform.

Because the tasks must solve diverse problems, a language general-purpose enough to solve them all would be difficult to write, analyze, and compile. Instead, a variety of languages have evolved, each best suited to a particular problem domain. For example, a language for signal-processing is often more convenient for a particular problem than, say, assembly, but might be poor for control-dominated behavior.

This paper describes popular hardware, software, dataflow, and hybrid languages, each of which excels a certain problems. Dataflow languages are good for signal processing, and hybrid languages combine ideas from the other three classes.

Due to space, this paper only describes the main features of each language. The author's book on the subject [9] provides many more details on all of these languages.

2 Hardware Languages

Verilog [14, 25] and VHDL [13, 23, 8, 2] are the most popular languages for hardware description and modeling (Figure 1, 2). Both model systems with discrete-event semantics that ignore idle portions of the design for efficient simulation. Both describe systems with structural hierarchy: a system consists of blocks that contain instances of primitives, other blocks, or concurrent processes. Connections are listed explicitly.

Verilog provides more primitives geared specifically toward hardware simulation. VHDL's primitive are assignments such as $a = b + c$ or procedural code. Verilog adds transistor and logic gate primitives, and allows new ones to be defined with truth tables.

Both languages allow concurrent processes to be described procedurally. Such processes sleep until awakened by an event that causes them to run, read and write variables, and suspend. Processes may wait for a period of time (e.g., `#10` in Verilog, `wait for 10ns` in VHDL), a value change (`@(a or b)`, `wait on a,b`), or an event (`@(posedge clk)`, `wait on clk until clk='1'`).

VHDL communication is more disciplined and flexible. Verilog communicates through *wires* or *regs*: shared memory locations that can cause race conditions. VHDL's signals behave like wires but the resolution function may be user-defined. VHDL's variables are local to a single process unless declared shared.

Verilog's type system models hardware with four-valued bit vectors and arrays for modeling memory. VHDL does not include four-valued vectors, but its type system allows them to be added. Furthermore, composite types such as C *structs* can be defined.

Overall, Verilog is the leaner language more directly geared toward simulating digital integrated circuits. VHDL is a much larger, more verbose language capable of handing a wider class of simulation and modeling tasks.

3 Software Languages

Software languages describe sequences of instructions for a processor to execute. As such, most consist of sequences of imperative instructions that communicate through memory: an array of numbers that hold their values until changed.

Each machine instruction typically does little more than, say, add two numbers, so high-level languages aim to specify many instructions concisely and intuitively. Arithmetic expressions are typical: coding an expression such as $ax^2 + bx + c$ in machine code is straightforward, tedious, and best done by a compiler. The C language provides such expressions, control-flow constructs such as loops and conditionals, and recursive functions. The C++ language adds classes as a way to build new data types, templates for polymorphic code, exceptions for error handling, and a standard

```
module fulladd(ai, bi, ci, o, co);
  input ai, bi, ci;
  output o, co;
  wire s1;

  xor x1(s1, ai, bi), x2(o, s1, ci);

  assign co = (ai + bi + ci) >= 2;
endmodule

module testadd;
  reg [2:0] y;
  wire      o, co;
  reg       clk;

  fulladd a1(y[0], y[1], y[2], o, co);

  initial begin
    y = 0;  clk = 0;
    $monitor($time,,"%d%d%d %d%d",
             y[2], y[1], y[0], co, o);
  end

  always #10 clk = ~clk;
  always @(posedge clk) y <= y + 1;
endmodule
```

Figure 1: A Verilog model for a full adder. This uses primitive gates, continuous assignment, and procedural code.

```
entity XOR2 is
  port (o: out Bit; a, b: in Bit);
end XOR2;

architecture arch1 of XOR2 is
begin
  A1: o <= (a xor b);
end arch1;

entity fulladd is
  port (ai, bi, ci: in Bit;
        o, co: out Bit);
end fulladd;

architecture arch1 of fulladd is
  signal s1 : Bit;
  component XOR2
    port(o: out Bit; a, b: in Bit);
  end component;
  for X1, X2: XOR2 use entity Work.XOR2;
begin
  X1: XOR2 port map (s1, ai, bi);
  X2: XOR2 port map (o, s1, ci);
  A1: co <= (ai and bi) or (ai and ci)
            or (bi and ci);
end arch1;
```

Figure 2: A VHDL model for a full adder that does not include a test bench; compare with Figure 1.

	C	C++	Java
Expressions	●	●	●
Control-flow	●	●	●
Recursive functions	●	●	●
Exceptions	❍	●	●
Classes & Inheritance		●	●
Templates	●		
Namespaces	●	●	
Multiple inheritance	●	❍	
Threads & Locks		●	●
Garbage collection	❍	●	

● full support ❍ partial support

Table 1: Software language features compared

```
  jmp   L2           # go to L2
L1:                  # label
  movl  %ebx, %eax   # n -> m
  movl  %ecx, %ebx   # r -> n
L2:
  xorl  %edx, %edx   # clear %edx
  divl  %ebx         # m / n
  movl  %edx, %ecx   # rem -> r
  testl %ecx, %ecx   # if r = 0,
  jne   L1           # go to L1
```

Figure 3: Euclid's algorithm in i386 assembly language. Symbols like `%ebx` represent registers. `movl` means "move long value." `divl %ebx` divides `%eax` by `%ebx` and puts the remainder in `%edx`.

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  char *c;
  while (++argv, --argc > 0) {
    c = argv[0] + strlen(argv[0]);
    while (--c >= argv[0])
      putchar(*c);
    putchar('\n');
  }
  return 0;
}
```

Figure 4: A C program that prints its arguments backwards. The outermost while loop iterates through the arguments (count in argc, array of strings in argv), while the inner loop starts a pointer at the end of the current argument and walks it backwards, printing each character along the way.

library of common data structures. Java is a still higher-level language that provides automatic garbage collection, threads, and monitors to synchronize them.

3.1 Assembly Languages

An assembly language program (Figure 3) is a list of processor instructions written in a symbolic, human-readable form. Each instruction consists of an operation such as addition along with some operands. E.g., `add r5,r2,r4` might add the contents of registers `r2` and `r4` and write the result to `r5`. Such arithmetic instructions are executed in order, but branch instructions can perform conditionals and loops by changing the processor's program counter—the address of the instruction being executed.

A processor's assembly language is defined by its opcodes, addressing modes, registers, and memories. The opcode distinguishes, say, addition from conditional branch, and an addressing mode defines how and where data is gathered and stored (e.g., from a register or from a particular memory location). Registers can be thought of as small, fast, easy-to-access pieces of memory.

3.2 The C Language

A C program (Figure 4) contains functions built from arithmetic expressions structured with loops and conditionals. Instructions in a C program run sequentially, but control-

```
class Cplx {
  double re, im;
public:
  Cplx(double v) : re(v), im(0) {}
  Cplx(double r, double i)
    : re(r), im(i) {}
  double abs() const {
    return sqrt(re*re + im*im);
  }
  void operator+= (const Cplx& a) {
    re += a.re; im += a.im;
  }
};

int main() {
  Cplx a(5), b(3,4);
  b += a;
  cout << b.abs() << '\n';
  return 0;
}
```

Figure 5: A C++ fragment illustrating a partial complex number type (the C++ library has a complete version).

flow constructs such as loops of conditionals can affect the order in which instructions execute. When control reaches a function call in an expression, control is passed to the called function, which runs until it produces a result, and control returns to continue evaluating the expression that called the function.

C derives its types from those a processor manipulates directly: signed and unsigned integers ranging from bytes to words, floating point numbers, and pointers. These can be further aggregated into arrays and structures—groups of named fields.

C programs use three types of memory. Space for global data is allocated when the program is compiled, the stack stores automatic variables allocated and released when their function is called and returns, and the heap supplies arbitrarily-sized regions of memory that can be deallocated in any order.

The C language is an ISO standard, but most people consult the book by Kernighan and Ritchie [17]. Ritchie designed the language.

3.3 C++

C++ (Figure 5) [24] extends C with structuring mechanisms for big programs: user-defined data types, a way to reuse code with different types, namespaces to group objects and avoid accidental name collisions when program pieces are assembled, and exceptions to handle errors. The C++ standard library includes a collection of efficient polymorphic data types such as arrays, trees, strings for which the compiler generates custom implementations.

A class defines a new data type by specifying its representation and the operations that may access and modify it. Classes may be defined by inheritance, which extends and modifies existing classes. For example, a rectangle class might add length and width fields and an area method to a shape class.

A template is a function or class that can work with multiple types. The compiler generates custom code for each different use of the template. For example, the same *min* template could be used for both integers and floating-point numbers.

```
import java.io.*;
class Counter {
    int value = 0;
    boolean present = false;
    public synchronized void count() {
        try { while (present) wait(); }
        catch (InterruptedException e) {}
        value++; present = true; notifyAll();
    }
    public synchronized int read() {
        try { while (!present) wait(); }
        catch (InterruptedException e) {}
        present = false; notifyAll();
        return value;
    }
}
class Count extends Thread {
    Counter cnt;
    public Count(Counter c) { cnt = c; start(); }
    public void run() { for (;;) cnt.count(); }
}
class Mod5 {
    public static void main(String args[]) {
        Counter c = new Counter();
        Count count = new Count(c);
        int v;
        for (;;) if ( (v = c.read()) % 5 == 0 )
            System.out.println(v);
    }
}
```

Figure 6: A contrived Java program that spawns a counting thread to print all numbers divisible by 5.

3.4 Java

Sun's Java language [1, 11, 20] resembles C++ but is incompatible. Like C++, Java is object-oriented, providing classes and inheritance. It is a higher-level language than C++ since it uses object references, arrays, and strings instead of pointers. Java's automatic garbage collection frees the programmer from memory management.

Java provides concurrent threads (Figure 6). Creating a thread involves extending the *Thread* class, creating instances of these objects, and calling their *start* methods to start a new thread of control that executes the objects' *run* methods.

Synchronizing a method or block uses a per-object lock to resolve contention when two or more threads attempt to access the same object simultaneously. A thread that attempts to gain a lock owned by another thread will block until the lock is released, which can be used to grant a thread exclusive access to a particular object.

3.5 RTOS

Many embedded systems use a real-time operating system (RTOS) to simulate concurrency on a single processor. An RTOS manages multiple running processes, each written in sequential language such as C. The processes perform the system's computation and the RTOS schedules them—attempts to meet deadlines by deciding which process runs when. Labrosse [18] describes the implementation of a particular RTOS.

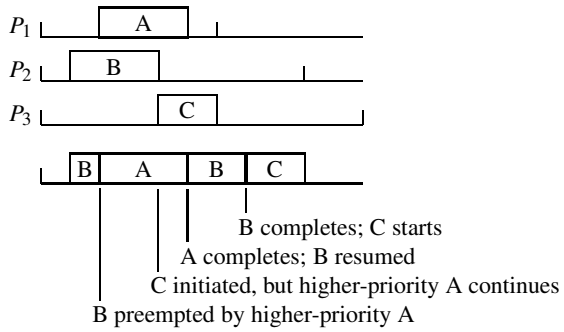

Figure 7: The behavior of an RTOS with fixed-priority preemptive scheduling. Rate-monotonic analysis gives process P_1 the highest priority since it has the shortest period; P_3 has the lowest. A would have missed its deadline (the tick mark) had it not preempted B.

Most RTOSes uses fixed-priority preemptive scheduling in which each process is given a particular priority (a small integer) when the system is designed. At any time, the RTOS runs the highest-priority running process, which is expected to run for a short period of time before suspending itself to wait for more data. Priorities are usually assigned using rate-monotonic analysis [6] (due to Liu and Layland [21]), which assigns higher priorities to processes that must meet more frequent deadlines.

4 Dataflow Languages

Dataflow languages describe systems of procedural processes that run concurrently and communicate through queues. Although clumsy for general applications, dataflow languages are a perfect fit for signal-processing algorithms, which use vast quantities of arithmetic derived from linear system theory to decode, compress, or filter data streams that represent periodic samples of continuously-changing values such as sound or video. Dataflow semantics are natural for expressing the block diagrams typically used to describe signal-processing algorithms, and their regularity makes dataflow implementations very efficient because otherwise costly run-time scheduling decisions can be made at compile time, even in systems containing multiple sampling rates.

4.1 Kahn Process Networks

Kahn Process Networks [16] form a formal basis for dataflow computation. Kahn's systems consist of processes that communicate exclusively through unbounded point-to-point first-in, first-out queues. Reading from a port makes a process wait until data is available, so the behavior of Kahn's networks does not depend on execution speeds.

Balancing processes' relative execution rates to avoid an unbounded accumulation of tokens is the challenge in scheduling a Kahn network. One general approach, proposed in Parks' thesis [22] places artificial limits on the size of each buffer. Any process that writes to a full buffer blocks until space is available, but if the system deadlocks because all buffers are full, the scheduler increases the capacity of the smallest buffer.

```
process f(in int u, v; out int w)
{
  int i; bool b = true;
  for (;;) {
    i = b ? wait(u) : wait(w);
    printf("%i\n", i);
    send(i, w);
    b = !b;
  }
}

process g(in int u; out int v, w)
{
  for (;;) {
    send(wait(u), v); send(wait(u), w);
  }
}

process h(in int u, out int v, int init)
{
  send(v, init);
  for(;;)
    send(wait(u), v);
}

channel int X, Y, Z, T1, T2;
f(Y, Z, X);
g(X, T1, T2);
h(T1, Y, 0);
h(T2, Z, 1);
```

Figure 8: A Kahn Process Network [16]. The f process alternately copies from its u and v ports to its w port; the g process does the opposite, copying its u port to alternately v and w; and h simply copies its input to its output.

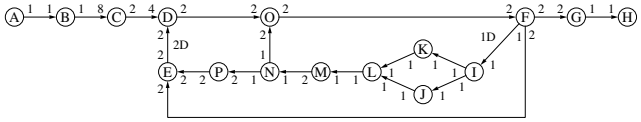

Figure 9: A modem in SDF (from Bhattacharyya et al. [5])

4.2 Synchronous Dataflow

Lee and Messerschmitt's Synchronous Dataflow [19] fix the communication patterns of the blocks in a Kahn network. Each time a block runs, it consumes and produces a fixed number of data tokens on each of its ports. This predictability allows SDF to be scheduled completely at compile-time, producing very efficient code.

Scheduling operates in two steps. First, the rate at which each block fires is established by considering the production and consumption rates of each block at the source and sink of each queue. For example, one of the arcs in Figure 9 implies $2C = 4D$. Once the rates are established, any algorithm that simulates the execution of the network without buffer underflow will produce a correct schedule if one exists. However, more sophisticated techniques reduce generated code and buffer sizes by better ordering the execution of the blocks (see Bhattacharyya et al. [4]).

5 Hybrid Languages

Hybrid languages combine ideas from others to solve different types of problems. Esterel excels at discrete control by blending software-like control flow with the synchrony and concurrency of hardware. Communication protocols are SDL's forte; it uses extended finite-state machines with single input queues. SystemC provides a flexible discrete-event simulation environment built on C++. CoCentric™ System Studio combines dataflow with Esterel-like finite-state machine semantics to simulate and synthesize dataflow applications that also require control.

5.1 Esterel

Intended for specifying control-dominated reactive systems, Esterel [3] combines the control constructs of an imperative software language with concurrency, preemption, and a synchronous model of time like that used in synchronous digital circuits. In each clock cycle, the program awakens, reads its inputs, produces outputs, and suspends.

An Esterel program communicates through signals that are either present or absent each cycle. In each cycle, each signal is absent unless an emit statement for the signal runs and makes the signal present for that cycle only. Esterel guarantees determinism by requiring each emitter of a signal to run before any statement that tests the signal.

5.2 SDL

SDL is a graphical specification language developed for describing telecommunication protocols defined by the ITU [15] (Ellsberger [10] is more readable). A system consists of concurrently-running FSMs, each with a single input queue, connected by channels that define which messages they carry. Each FSM consumes the most recent message in its queue, reacts to it by changing internal state or sending messages to other FSMs, changes to its next state, and repeats the process. Each FSM is deterministic, but because messages from other FSMs may arrive in any order because of varying execution speed and communication delays, an SDL system may behave nondeterministically.

5.3 SystemC

The SystemC language is a C++ subset for system modeling. A SystemC specification is simulated by compiling it with a standard C++ compiler and linking in freely-distributed class libraries from www.systemc.org.

The SystemC language builds systems from Verilog- and VHDL-like modules. Each has a collection of I/O ports and may contain instances of other modules or processes defined by a block of C++ code.

SystemC uses a discrete-event simulation model. The SystemC scheduler executes the code in a process in response to an event such as a clock signal, or a delay. This model resembles that used in Verilog and VHDL, but has the flexibility of operating with a general-purpose programming language.

	Esterel	SDL	SystemC	CCSS
Concurrent	●	●	●	●
Hierarchy	●	●	●	●
Preemption	●		●	●
Deterministic	●		❍	●
Synchronous communication	●		●	●
Buffered communication		●	●	●
FIFO communication		●	❍	●
Procedural	●	❍	●	❍
Finite-state machines	●	●	❍	●
Dataflow		●	●	●
Multi-rate dataflow			❍	●
Software implementation	●	●	●	●
Hardware implementation	●		●	●

● full support ❍ partial support.

Table 2: Hybrid language features compared.

```
module Example:

input  S, I;
output O;

signal R, A in
  every S do
    await I;
    weak abort
      sustain R
    when immediate A;
    emit O
  ||
    loop
      pause; pause;
      present R then emit A end;
    end
  end
end

end module
```

Figure 10: An Esterel program modeling a shared resource.

```
#include "systemc.h"

struct complex_mult : sc_module {
  sc_in<int>  a, b;
  sc_in<int>  c, d;
  sc_out<int> x, y;
  sc_in_clk   clock;

  void do_mult() {
    for (;;) {
      x = a * c - b * d;
      wait();
      y = a * d + b * c;
      wait();
    }
  }

  SC_CTOR(complex_mult) {
    SC_CTHREAD(do_mult, clock.pos());
  }
};
```

Figure 11: A SystemC model for a complex multiplier.

5.4 CoCentric System Studio

CoCentric System Studio™ [7] uses a hierarchical formalism that combines Kahn-like dataflow and hierarchical, concurrent FSMs. The FSMs resemble Harel's Statecharts [12], but use Esterel's synchronous semantics to ensure determinism.

A CCSS model is built hierarchically from Dataflow, AND, OR, and Gated models. Dataflow models are Kahn Process networks. The blocks may be dataflow primitives written in a C++ subset or other hierarchical models. AND models run concurrently and communicate with Esterel-like synchronous semantics. OR models are finite-state machines that may manipulate data and whose states may contain other models. Gated models contain sub-models whose execution can be temporarily suspended under external control.

References

[1] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Addison-Wesley, Reading, Massachusetts, third edition, 2000.

[2] Peter J. Ashenden. *The Designer's Guide to VHDL*. Morgan Kaufmann, San Francisco, California, 1996.

[3] Gérard Berry and Georges Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, November 1992.

[4] Shuvra S. Bhattacharyya, Ranier Leupers, and Peter Marwedel. Software synthesis and code generation for signal processing systems. *IEEE Transactions on Circuits and Systems—II: Analog and Digital Signal Processing*, 47(9):849–875, September 2000.

[5] Shuvra S. Bhattacharyya, Praveen K. Murthy, and Edward A. Lee. Synthesis of embedded software from synchronous dataflow specifications. *Journal of VLSI Signal Processing Systems*, 21(2):151–166, June 1999.

[6] Loic P. Briand and Daniel M. Roy. *Meeting Deadlines in Hard Real-Time Systems: The Rate Monotonic Approach*. IEEE Computer Society Press, New York, New York, 1999.

[7] Joseph Buck and Radha Vaidyanathan. Heterogeneous modeling and simulation of embedded systems in El Greco. In *Proceedings of the Eighth International Workshop on Hardware/Software Codesign (CODES)*, San Diego, California, May 2000.

[8] Ben Cohen. *VHDL Coding Styles and Methodologies*. Kluwer, Boston, Massachusetts, second edition, 1999.

[9] Stephen A. Edwards. *Languages for Digital Embedded Systems*. Kluwer, Boston, Massachusetts, September 2000.

[10] Jan Ellsberger, Dieter Hogrefe, and Amardeo Sarma. *SDL: Formal Object-Oriented Language for Communicating Systems*. Prentice Hall, Upper Saddle River, New Jersey, second edition, 1997.

[11] James Gosling, Bill Joy, Guy Steele, , and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, Reading, Massachusetts, second edition, 2000.

[12] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[13] IEEE Computer Society, 345 East 47th Street, New York, New York. *IEEE Standard VHDL Language Reference Manual (1076-1993)*, 1994.

[14] IEEE Computer Society, 345 East 47th Street, New York, New York. *IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language (1364-1995)*, 1996.

[15] International Telecommunication Union, Place des Nations, CH-1221, Geneva 20, Switzerland. *ITU-T Recommendation Z.100: Specification and Description Language*, 1999.

[16] Gilles Kahn. The semantics of a simple language for parallel programming. In *Information Processing 74: Proceedings of IFIP Congress 74*, pages 471–475, Stockholm, Sweden, August 1974. North-Holland.

[17] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Langage*. Prentice Hall, Upper Saddle River, New Jersey, second edition, 1988.

[18] Jean Labrosse. *MicroC/OS-II*. CMP Books, Lawrence, Kansas, 1998.

[19] Edward A. Lee and David G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987.

[20] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, Massachusetts, 1999.

[21] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, January 1973.

[22] Thomas M. Parks. *Bounded Scheduling of Process Networks*. PhD thesis, University of California, Berkeley, 1995. Available as UCB/ERL M95/105.

[23] Douglas L. Perry. *VHDL*. McGraw-Hill, New York, third edition, 1998.

[24] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Massachusetts, third edition, 1997.

[25] Donald E. Thomas and Philip R. Moorby. *The Verilog Hardware Description Language*. Kluwer, Boston, Massachusetts, fourth edition, 1998.