Research Areas

Stephen A. Edwards

Department of Computer Science,
Columbia University

www.cs.columbia.edu/~sedwards

sedwards@cs.columbia.edu

Embedded Systems

Computers masquerading as something else.

Casio
Camera
Watch

Nokia 7110
Browser
Phone

Sony
Playstation 2

Philips
DVD Player

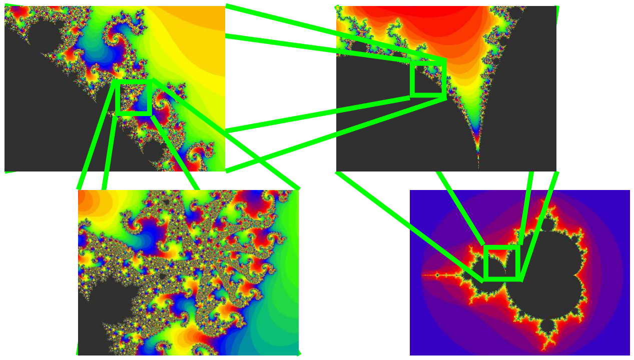
Philips
TiVo Recorder

Long-Term Goal

Supplying tools that speed the development of embedded systems.

Embedded Systems Challenges

Real-time



Complexity

Photo by Thomas Danoghue

Concurrency

Legacy Languages

Software complexity growing

Size of Typical Embedded System

1985	13 kLOC	
1989	21 kLOC	↓ 44 % per year
1998	1 MLOC	
2000	2 MLOC	
2008	16 MLOC	≈ Windows NT 4.0
2010	32 MLOC	≈ Windows 2000

Source: “ESP: A 10-Year Retrospective,” Embedded Systems Programming, November 1998

Written in stone-age languages

“Which of the following programming languages have you used for embedded systems in the last 12 months?”

C	81%
Assembly	70%
C++	39%
Visual Basic	16%
Java	7%

Source: “ESP: A 10-Year Retrospective,” Embedded Systems Programming, November 1998

Domain-Specific Languages

Little languages that fit the problem

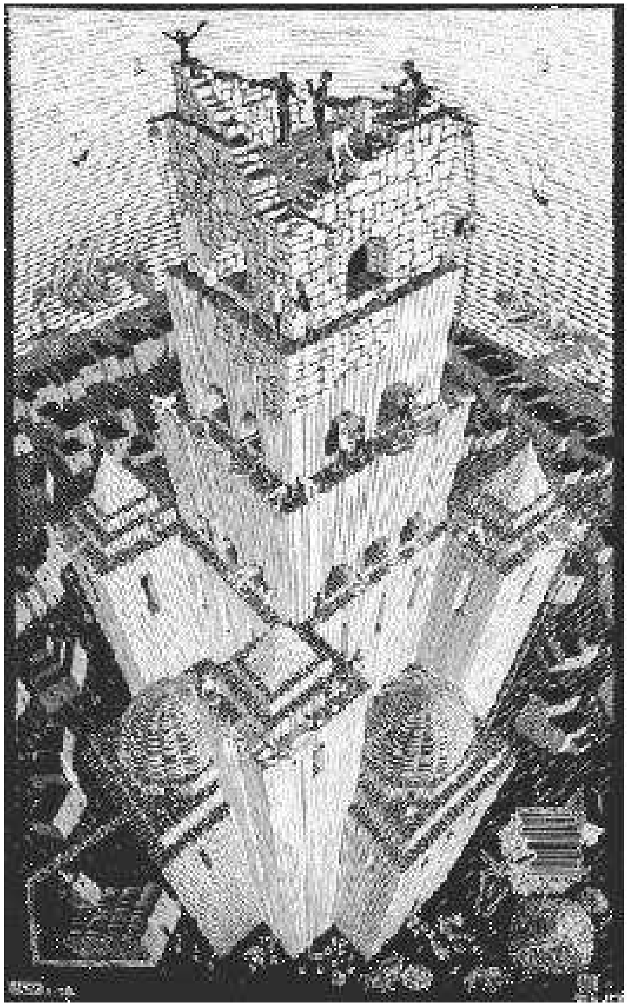
More succinct description that are

1. Quicker to create
2. Easier to get right

More opportunities for optimization and analysis

General-purpose languages hindered by undecidability

Domain-specific languages much simpler



Real-Time Languages

Esterel

The Esterel Real-Time Langauge

Synchronous language developed by Gérard Berry in France

Basic idea: use global clock for synchronization in software like that in synchronous digital hardware.

Challenge: How to combine concurrency, synchronization, and instantaneous communication

Esterel

Restart when RESET present

Infinite loop

Wait for next cycle with A present

Run concurrently

Same-cycle bidirectional communication

```
every RESET do
  loop
    await A;
    emit B;
    present C then
      emit D
    end;
    pause
  end
||
  loop
    present B then
      emit C
    end;
    pause
  end
end
```

Previous Esterel Work

- Thesis on Esterel semantics in a heterogeneous environment (Ptolemy/Edward Lee/Berkeley)
- To appear in Science of Computer Programming

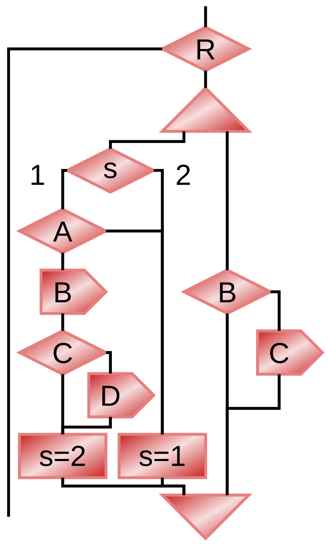
- Compiler that speeds up certain large programs 100$\times$
- Used inside Synopsys' CoCentric System Studio
- Has limitations (e.g., owned by former employer)
- Published in IEEE Transactions on Computer-Aided Design 21(2), 2002.

Previous Esterel Compiler

```
every R do
  loop
    await A;
    emit B;
    present C then
      emit D end;
    pause
  end
||
  loop
    present B then
      emit C end;
    pause
  end
end
```

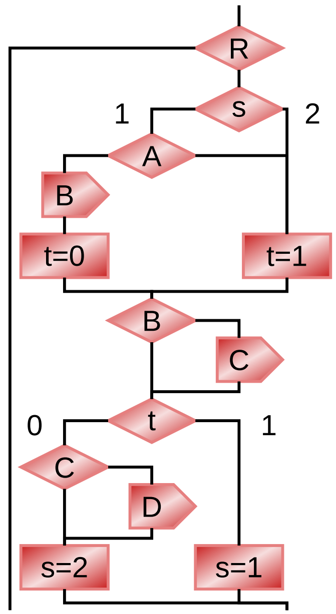
Esterel

Source



Concurrent

CFG



Sequential

CFG

```
if ((s0 & 3) == 1) {
  if (S) {
    s3 = 1; s2 = 1; s1 = 1;
  } else
    if (s1 >> 1)
      s1 = 3;
    else {
      if ((s3 & 3) == 1) {
         s3 = 2; t3 = L1;
      } else {
         t3 = L2;
      }
```

C code

Ongoing Esterel Work

- New compiler infrastructure designed for research
- Better circuits from Esterel programs (Cristian Soviani)
- Faster code from PDGs (Jia Zeng)
- Event-driven code (Vimal Kapadia, Michael Halas)
- An interpreter for small-footprint applications

The Hardware/Software Boundary

Device Drivers

Languages for Device Drivers

Device drivers are those pieces of software that you absolutely need that never seem to work

Big security/reliability hole: run in Kernel mode

Responsible for 80% of all Windows crashes

Tedious, difficult-to-write

Ever more important as customized hardware proliferates

Work by Others

Thibault, Marlet, and Consel

IEEE Transactions Software Engineering, 1999

Developed the Graphics Adaptor Language for writing XFree86 video card drivers

Report GAL drivers are 1/9th the size of their C counterparts

No performance penalty

Ongoing Work

Develop language for network card drivers under Linux (Chris Conway)

Sharing drivers between Linux and FreeBSD (Tom Heydt-Benjamin)

Ultimate vision: compiler takes two programs: device spec. and OS spec. and synthesizes appropriate driver.

OS vendor makes sure OS spec. is correct; Hardware designer makes sure hardware spec. is correct.

NE2000 Ethernet driver (fragment)

```
ioports ne2000 {
  bits cr {
    bit stop, sta, transmit;
    enum:3 { 001=remRead, 010=remWrite,
             011=sendPacket, 1**=DMAdone }
    enum:2 { 00=page0, 01=page1, 10=page2 }
  }
  paged p {
    page0 { cr.page0; } {
      twobyte clda;
      byte bnry;
      bits tsr {
        bit ptx, 1, col, abt, crs, 0, cdh, owc;
      }
    page1 { cr.page1; } {
      byte:6 par;
      byte curr;
      byte:8 mar;
    }
```

Program Correctness

Verification Library Language

Verification Library Language

Joint work with Al Aho

Language extensions to support verification libraries for Java

Traditional Libraries	Provide functionality
Verification Libraries	Provide improved confidence in program correctness

Vision is a new methodology: verification as part of the development process, part of the same toolbox as adding functionality.

“Hello World” Example

Require Java class names to start with capital letters.

```
enforce vll.capitalIdentifiers;
public class MyExample {
 public int nothing;
}
```

```
vllpackage vll.capitalIdentifiers;

AST() {
  find "class <name>" in ast
    if (name[0] < ’A’ || name[0] > ’Z’)
      warning("Uncapitalized class name: ", name);
}
```

Example 2: Locks

Ensuring locks are acquired in a consistent order.

```
public class MyClass {
  private static final Object l1 = new Integer(0);
  private static final Object l2 = new Integer(1);

  public void method1() {
    synchronized (l1) {
      synchronized (l2) {
      }
    }
  }

  public void method2() {
    synchronized (l2) {    // l2 first makes this
      synchronized (l1) {  // a possible source of deadlock
      }
    }
  }
}
```

Example 2: Locks Implementation

```
vllpackage vll.orderedLocks;

AST {
  Digraph g; // g is a user-defined directed graph object
  find "synchronized (<obj1>) {
             ...
             synchronized (<obj2>) { ... }
             ...
           }" in ast
    if ( !g.addEdgeWithoutCycle(obj1,obj2) )
      warning("Object ", obj2, " locked after ", obj1);
}
```

Example 3: Enforcing the Visitor Pattern

Illustrates desire for application-specific verification libraries.

```
enforce vll.visitor(MyVisitorClass, [Object1, Object2]);

public class MyVisitorClass {
 void visit(Object1 o) { }
 void visit(Object2 o) { }
}
```

Example 3: Enforcing the Visitor Pattern

```
vllpackage vll.visitor;

AST(Class visitorClass, vector<Class> objectClasses) {
  find "class #visitorClass" in ast then {
    foreach (Class objClass in objectClasses) {
      find "void accept(#visitorClass <arg>)
            { <arg>.visit(this); }" in objClass else
        warning("Missing or erroneous accept() in ",
                objClass);
      find "void visit(#objClass <arg>) { ... }"
        in visitorClass else
        warning("Missing visit(", objClass, ")");
    }
  } else {
    warning("visitor class ", visitorClass, " not defined");
  }
}
```

Other Verification Libraries

- Lint-like function call chekers
- Library that assumes the program is an FSM and can be checked using standard FSM tools
- Library that statically checks if a Java program uses a particular set of methods (e.g., deprecated ones)
- Library that removes array-bounds-checking code that can be proven unecessary

Think of a language mechanism that can supply **`-Wall`**, lint, purify, Spin, SLAM, Prefix, etc. as libraries as easy to use as those for I/O, GUIs, etc.

Porting Tools

Type inference for C

Type Inference for C

Intended use: porting C code from one environment to another.

Assume that old header files are not available or difficult to use.

Identifies missing function declarations and proposes prototypes.

Type Inference for C: Example

```
void main()
{
  if (today_is_wednesday()) {
    double a = sin(1.23);
  }
  printf("Hello World");
}
```

would report

```
double sin(double);
bool today_is_wednesday();
void printf(char *);
```

Porting Tools

“One Long Strand”

One Long Strand

Distinguishes active and dead lines in C source.

Dead code, dead functions, dead declarations, dead header file inclusions.

Uses:

- Cleaning up a large software project
- Removing unwanted features from reused software
- Understanding relationships among software features

One Long Strand: Example

```
#include <stdio.h>
#include <math.h>

void main()
{
  if (0) {
    double a = sin(1.23);
  }
  printf("Hello World");
}

void foo()
{
}
```


Thank you