Using and Compiling Esterel

Prof. Stephen A. Edwards
Columbia University
Department of Computer Science
sedwards@cs.columbia.edu
http://www.cs.columbia.edu/~sedwards/
Presented at CCU, August 17, 2004

The Esterel Language

Developed by Gérard Berry
starting 1983
Originally for robotics applications
Imperative, textual language
Synchronous model of time like that in digital circuits
Concurrent
Deterministic

A Simple Example

The specification:
The output O should occur when inputs A and B have both arrived. The R input should restart this behavior.

A First Try: An FSM

The Esterel Version

module ABRO:
  input A, B, R;
  output O;
  loop
    [ await A || await B ];
    emit O
each R
end module

Much simpler since language includes notions of signals, waiting, and reset.

The Esterel Version

module ABRO:
  input A, B, R;
  output O;
  loop
    [ await A || await B ];
    emit O
each R
end module

Basic Ideas of Esterel

Imperative, textual language
Concurrent
Based on synchronous model of time:
  • Program execution synchronized to an external clock
  • Like synchronous digital logic
  • Suits the cyclic executive approach
Two types of statements:
  • Combinational statements, which take “zero time” (execute and terminate in same instant, e.g., emit)
  • Sequential statements, which delay one or more cycles (e.g., await)

Uses of Esterel

Wristwatch
  • Canonical example
  • Reactive, synchronous, hard real-time
Controllers, e.g., for communication protocols
Avionics
  • Fuel control system
  • Landing gear controller
  • Other user interface tasks
Processor components (cache controller, etc.)
**Advantages of Esterel**

Model of time gives programmer precise timing control
Concurrence convenient for specifying control systems
Completely deterministic
  - Guaranteed: no need for locks, semaphores, etc.
Finite-state language
  - Easy to analyze
  - Execution time predictable
  - Much easier to verify formally
Amenable to both hardware and software implementation

**Disadvantages of Esterel**

Finite-state nature of the language limits flexibility
- No dynamic memory allocation
- No dynamic creation of processes
Little support for handling data; limited to simple decision-dominated controllers
Synchronous model of time can lead to overspecification
Semantic challenges:
  - Avoiding causality violations often difficult
  - Difficult to compile
Limited number of users, tools, etc.

**The Esterel Language**

**Esterel’s Model of Time**

The standard CS model (e.g., Java’s) is asynchronous: threads run at their own rate. Synchronization is through calls to wait() and notify().

Esterel’s model of time is synchronous like that used in hardware. Threads march in lockstep to a global clock.

```
Semaphore Semaphore
process

clock

Semaphore Semaphore
process

clock
```

**Signals**

Esterel programs communicate through signals
These are like wires
Each signal is either present or absent in each cycle
Can’t take multiple values within a cycle
Presence/absence not held between cycles
Broadcast across the program
Any process can read or write a signal

**Basic Esterel Statements**

`emit S`
Make signal S present in the current cycle
A signal is absent unless emitted in that cycle.

`pause`
Stop for this cycle and resume in the next.

`present S then s1 else s2 end`
Run s1 immediately if signal S is present in the current cycle, otherwise run s2.

**Simple Example**

```
module Example1:
output A, B, C;
emit A;
present A then
  emit B
end;
pause;
emit C
end module
```

**Signal Coherence Rules**

Each signal is only present or absent in a cycle, never both
All writers run before any readers do
Thus

```
present A else
  emit A
end
```

is an erroneous program. (Deadlocks.)
The Esterel compiler rejects this program.

**Advantage of Synchrony**

Easy to regulate time
Synchronization is free (e.g., no Bakers’ algorithm)
Speed of actual computation nearly uncontrollable
Allows function and timing to be specified independently
Makes for deterministic concurrency
Explicit control of “before” “after” “at the same time”
Time Can Be Controlled Precisely

This guarantees every 60th S an M is emitted

every 60 S do every invokes its body every 60th S emit M
emit takes no time (cycles)
S S S S S S
M M
1 ... 59 60 61 ... 120

The || Operator

Groups of statements separated || by run concurrently and terminate when all groups have terminated
[ emit A; pause; emit B;
||
  pause; emit C; pause; emit D
];
emit E
A B C D E

Communication Is Instantaneous

A signal emitted in a cycle is visible immediately
[ pause; emit A; pause; emit A
||
pause; present A then emit B end ]
A A B

Bidirectional Communication

Processes can communicate back and forth in the same cycle
[
  pause; emit A;
present B then emit C end;
pause; emit A
||
pause; present A then emit B end ]
A A B C

Concurrency and Determinism

Signals are the only way for concurrent processes to communicate
Esterel does have variables, but they cannot be shared
Signal coherence rules ensure deterministic behavior
Language semantics clearly defines who must communicate with whom when

The Await Statement

The await statement waits for a particular cycle
await S waits for the next cycle in which S is present
[ emit A ; pause ; pause; emit A
||
  await A; emit B ]
A A B

The Await Statement

Await normally waits for a cycle before beginning to check
await immediate also checks the initial cycle
[
  emit A ; pause ; pause; emit A
||
  await immediate A; emit B
] A A B

Loops

Esterel has an infinite loop statement
Rule: loop body cannot terminate instantly
Needs at least one pause, await, etc.
Can’t do an infinite amount of work in a single cycle

Loops and Synchronization

Instantaneous nature of loops plus await provide very powerful synchronization mechanisms

Loops and Synchronization

loop
  await 60 S;
  emit M
end
S S S S S S M M
1 ... 59 60 61 ... 120
Preemption

Often want to stop doing something and start doing something else
E.g., Ctrl-C in Unix: stop the currently-running program
Esterel has many constructs for handling preemption

The Abort Statement

Basic preemption mechanism
General form:
\[
\text{abort} \\
\text{statement} \\
\text{when} \ \text{condition}
\]
Runs statement to completion. If condition ever holds, abort terminates immediately.

Strong vs. Weak Preemption

Strong preemption:
- The body does not run when the preemption condition holds
- The previous example illustrated strong preemption

Weak preemption:
- The body is allowed to run even when the preemption condition holds, but is terminated thereafter
- “weak abort” implements this in Esterel

The Trap Statement

Esterel provides an exception facility for weak preemption
Interacts nicely with concurrency
Rule: outermost trap takes precedence

The Trap Statement

Normal termination from first process
\[
\text{trap T in} \\
\text{[} \\
\text{pause; } \text{emit A;} \\
\text{pause; } \text{emit A;} \\
\text{pause when B;} \\
\text{emit C} \\
\text{||} \\
\text{await B;} \\
\text{emit C} \\
\text{]} \\
\text{exit T} \\
\text{||} \\
\text{emit D}
\]

Emitted also runs
\[
\text{trap T1 in} \\
\text{trap T2 in} \\
[ \\
\text{||} \\
\text{exit T1} \\
\text{||} \\
\text{exit T2} \\
\text{]} \\
\text{end trap; } \text{emit D}
\]

Second process allowed to run even though first process has exited

Strong vs. Weak Abort

Strong abort emit A does not run
Weak abort emit A runs
\[
\text{abort} \\
\text{pause; } \text{pause; } \text{emit A;} \\
\text{pause when B;} \\
\text{emit C}
\]

B A

C

A D

B C

Normal Termination

Aborted termination

Aborted termination; emit A preempted

B not checked in first cycle (like await)

Nested Traps

Outer trap takes precedence; control transferred directly to the outer trap statement.
emit A not allowed to run.

\[
\text{trap T1 in} \\
\text{trap T2 in} \\
\text{||} \\
\text{exit T1} \\
\text{||} \\
\text{exit T2} \\
\text{]} \\
\text{end; } \text{emit A} \\
\text{end; } \text{emit B}
\]
The Suspend Statement

Preemption (abort, trap) terminate something, but what if you want to resume it later?

Like the unix Ctrl-Z

Esterel's suspend statement pauses the execution of a group of statements

Only strong preemption: statement does not run when condition holds

Causality

Can be very complicated because of instantaneous communication

For example, this is also erroneous

```
abort
  emit B
  when A
  pause;
  present B then emit A end
```

Emission of B indirectly causes emission of A

Causality Example

```
emit A;
  present B then emit C end;
  present A else emit B end;
```

Considered acceptable to the latest compiler

After emit A runs, it is clear that B cannot be emitted because A’s presence runs the “then” branch of the second present

B declared absent, both present statements run

Esterel Programming Examples

People Counter Example

Construct an Esterel program that counts the number of people in a room. People enter the room from one door with a photocell that changes from 0 to 1 when the light is interrupted, and leave from a second door with a similar photocell. These inputs may be true for more than one clock cycle.

The two photocell inputs are called ENTER and LEAVE. There are two outputs: EMPTY and FULL, which are present when the room is empty and contains three people respectively.

Overall Structure

ENTER → Conditioner → ADD → Counter → EMPTY

LEAVE → Conditioner → SUB → Counter → FULL

Conditioner detects rising edges of signal from photocell.
Counter tracks number of people in the room.

Implementing the Conditioner

module Conditioner:
  input A;
  output Y;
  loop
    await A; emit Y;
    await ~A;
  end
endmodule

Testing the Conditioner

# esterel -simul cond.strl
# gcc -o cond cond.c -lcsimul # may need -L
# ./cond
Conditioner> ;
--- Output: A;  # Rising edge
Conditioner> A;  # Doesn’t generate a pulse
--- Output: Y
Conditioner> ;  # Reset
--- Output:
Conditioner> A;  # Another rising edge
--- Output: Y
Conditioner> ;
--- Output:
Conditioner> A;
--- Output: Y

Implementing the Counter: First Try

module Counter:
  input ADD, SUB;
  output FULL, EMPTY;
  var count := 0 : integer in
    loop
      present ADD then if count < 3 then
        count := count + 1 end end;
      present SUB then if count > 0 then
        count := count - 1 end end;
      if count = 0 then emit EMPTY end;
      if count = 3 then emit FULL end;
      pause
    end
end module

Testing the Counter

Counter> ;
--- Output: EMPTY
Counter> ADD SUB;
--- Output: EMPTY
Counter> ADD;
--- Output:
Counter> SUB;
--- Output: EMPTY
Counter> ADD;
--- Output:
Counter> ADD;
--- Output: FULL
Counter> ADD SUB;
--- Output: # Oops: still FULL

Counter, second try

module Counter:
  input ADD, SUB;
  output FULL, EMPTY;
  var c := 0 : integer in
    loop
      present ADD then
        present SUB else
          if c < 3 then c := c + 1 end end
        end
      else
        present SUB then
          if c > 0 then c := c - 1 end end
        end;
      if c = 0 then emit EMPTY end;
      if c = 3 then emit FULL end;
      pause
    end
end module

Testing the second counter

Counter> ;
--- Output: EMPTY
Counter> ADD SUB;
--- Output: EMPTY
Counter> ADD SUB;
--- Output: EMPTY
Counter> ADD;
--- Output:
Counter> ADD;
--- Output: FULL
Counter> ADD SUB;
--- Output: # Oops: still FULL

Assembling the People Counter

module PeopleCounter:
  input ENTER, LEAVE;
  output EMPTY, FULL;
  signal ADD, SUB in
    run Conditioner[signal ENTER / A, ADD / Y]
    run Conditioner[signal LEAVE / A, SUB / Y]
    run Counter
end module

Vending Machine Example

Design a vending machine controller that dispenses gum once. Two inputs, N and D, are present when a nickel and dime have been inserted, and a single output, GUM, should be present for a single cycle when the machine has been given fifteen cents. No change is returned.

N =  
D = 
GUM = 

Source: Katz, Contemporary Logic Design, 1994, p. 389
**Vending Machine Solution**

```plaintext
module Vending:
  input N, D;
  output GUM;

  loop
    var m := 0 : integer in
    trap WAIT in
      loop
        present N then m := m + 5; end;
        present D then m := m + 10; end;
        if m >= 15 then exit WAIT end;
      end
    end
    emit GUM; end
  end
endmodule
```

**Alternative Solution**

```plaintext
loop
  await
    case immediate N do await
      case N do nothing
      case N do nothing
      end
    end
    case immediate D do nothing
    case immediate D do nothing
    end
    case immediate D do nothing
    case D do nothing
    end
  end
  emit GUM; end
end
```

**Tail Lights Example**

Construct an Esterel program that controls the turn signals of a 1965 Ford Thunderbird.

**Tail Light Behavior**

![Tail Light Behavior](image)

**Tail Lights**

There are three inputs, LEFT, RIGHT, and HAZ, that initiate the sequences, and six outputs, LA, LB, LC, RA, RB, and RC. The flashing sequence is

<table>
<thead>
<tr>
<th>LC</th>
<th>LB</th>
<th>LA</th>
<th>step</th>
<th>RA</th>
<th>RB</th>
<th>RC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**A Single Tail Light**

```plaintext
module Lights:
  output A, B, C;

  loop
    emit A; pause;
    emit A; emit B; pause;
    emit A; emit B; emit C; pause;
    pause
  end
end module
```

**The T-Bird Controller Interface**

```plaintext
module Thunderbird :
  input LEFT, RIGHT, HAZ;
  output LA, LB, LC, RA, RB, RC;

  ... end module
```

**The T-Bird Controller Body**

```plaintext
loop
  await
    case immediate HAZ do await
      run Lights[signal LA/A, LB/B, LC/C]
      when [not HAZ]
      run Lights[signal RA/A, RB/B, RC/C]
      when [not HAZ]
      case immediate LEFT do
        run Lights[signal LA/A, LB/B, LC/C]
        when [not LEFT]
      case immediate RIGHT do
        run Lights[signal RA/A, RB/B, RC/C]
        when [not RIGHT]
      end
    end
  end
end
```

**Comments on the T-Bird**

I choose to use Esterel's innate ability to control the execution of processes, producing succinct easy-to-understand source but a somewhat larger executable.

An alternative: Use signals to control the execution of two processes, one for the left lights, one for the right.

A challenge: synchronizing hazards.

Most communication signals can be either level- or edge-sensitive.

Control can be done explicitly, or implicitly through signals.
Traffic-Light Controller Example

This controls a traffic light at the intersection of a busy highway and a farm road. Normally, the highway light is green but if a sensor detects a car on the farm road, the highway light turns yellow then red. The farm road light then turns green until there are no cars or after a long timeout. Then, the farm road light turns yellow then red, and the highway light returns to green. The inputs to the machine are the car sensor C, a short timeout signal S, and a long timeout signal L. The outputs are a timer start signal R, and the colors of the highway and farm road lights.


The Traffic Light Controller

module Fsm:
input C, L, S;
output R;
output HG, HY, FG, FY;
loop
emit HG ; emit R; await [C and L];
emit HY ; emit R; await S;
emit FG ; emit R; await [not C or L];
emit FY ; emit R; await S;
end module

The Traffic Light Controller

module Timer:
input R, SEC;
output L, S;
loop
weak abort
await 3 SEC;
[ sustain S ]
||
await 5 SEC;
sustain L
] when R;
end
end module

The Traffic Light Controller

module TLC:
input C, SEC;
output HG, HY, FG, FY;
signal S, L, S in
run Fsm
||
run Timer
end
end module

Compiling Esterel

Semantics of the language are formally defined and deterministic
It is the responsibility of the compiler to ensure the generated executable behaves correctly w.r.t. the semantics
Challenging for Esterel

Compilation Challenges

- Concurrency
- Interaction between exceptions and concurrency
- Preemption
- Resumption (pause, await, etc.)
- Checking causality
- Reincarnation
  Loop restriction prevents most statements from executing more than once in a cycle
  Complex interaction between concurrency, traps, and loops allows certain statements to execute twice or more

Automata-Based Compilation

Key insight: Esterel is a finite-state language
Each state is a set of program counter values where the program has paused between cycles
Signals are not part of these states because they do not hold their values between cycles
Esterel has variables, but these are not considered part of the state

Automata Compiler Example

```java
void tick() {
    static int s = 0;
    A = B = 0;
    switch (s) {
    case 0:
        A = 1;
        s = 1;
        break;
    case 1:
        if (C) {
            B = 1; s = 0;
        }
        break;
    }
}
```
### Automata Compiler Example

```c
emit A;
emit B;
await C;
emit D;
present E then emit B end
```

```c
switch (s) {
    case 0:
        A=1;
        B=1;
        s=1;
        break;
    case 1:
        if (C) {
            D=1;
            if (E) B=1;
            s=2;
        }
        break;
    case 2:
}
```

### Automata Compilation Considered

- Very fast code (Internal signaling can be compiled away)
- Can generate a lot of code because concurrency can cause exponential state growth
- $n$-state machine interacting with another $n$-state machine can produce $n^2$ states
- Language provides input constraints for reducing states
  - “these inputs are mutually exclusive”
    
  \[ \text{relation } A \# B \# C; \]
  - “if this input arrives, this one does, too”
    
  \[ \text{relation } D => E; \]

### Automata Compilation

- Not practical for large programs
- Theoretically interesting, but don’t work for most programs longer than 1000 lines
- All other techniques produce slower code

### Netlist-Based Compilation

- Key insight: Esterel programs can be translated into Boolean logic circuits
- Netlist-based compiler:
  - Translate each statement into a small number of logic gates, a straightforward, mechanical process
  - Generate code that simulates the netlist

### Netlist Example

```c
emit A; emit B; await C;
emit D; present E then emit B end
```

### Netlist Compilation Considered

- Scales very well
  - Netlist generation roughly linear in program size
  - Generated code roughly linear in program size
  - Good framework for analyzing causality
  - Semantics of netlists straightforward
  - Constructive reasoning equivalent to three-valued simulation
- Terribly inefficient code
  - Lots of time wasted computing irrelevant values
  - Can be hundreds of time slower than automata
  - Little use of conditionals

### Netlist Compilation

- Currently the only solution for large programs that appear to have causality problems
- Scalability attractive for industrial users
- Currently the most widely-used technique

### Control-Flow Graphs

- Key insight: Esterel looks like a imperative language, so treat it as such
- Esterel has a fairly natural translation into a concurrent control-flow graph
- Trick is simulating the concurrency
- Concurrent instructions in most Esterel programs can be scheduled statically
- Use this schedule to build code with explicit context switches in it
Average Cycle Times (Pentium)

Control-flow Approach Considered

Scales as well as the netlist compiler, but produces much faster code, almost as fast as automata.

Not an easy framework for checking causality.

Static scheduling requirement more restrictive than netlist compiler.

This compiler rejects some programs the others accept.

Only implementation hiding within Synopsys' CoCentric System Studio. Will probably never be used industrially.


Our Technique 2: Static Discrete Events

Event-driven C back end

module Example:
input I, S;
output O, Q;
signal R, A in
\begin{verbatim}
every S do
  weak abort sustain R when immediate A;
  emit O
  loop
    pause; pause;
    present R then emit A
    end present
  end loop
end every
end signal
end module
Generated code (1)
#define sched1a next1 = head1, head1 = &&C1a
#define sched1b next1 = head1, head1 = &&C1b
#define sched2 next2 = head1, head1 = &&C2
#define sched3a next3 = head1, head1 = &&C3a
#define sched3b next3 = head1, head1 = &&C3b
#define sched4 next4 = head2, head2 = &&C4
#define sched5a next5 = head3, head3 = &&C5a
#define sched5b next5 = head3, head3 = &&C5b
#define sched5c next5 = head3, head3 = &&C5c
#define sched6a next6 = head4, head4 = &&C6a
#define sched6b next6 = head4, head4 = &&C6b
#define sched6c next6 = head4, head4 = &&C6c
#define sched7a next7 = head5, head5 = &&C7a
#define sched7b next7 = head5, head5 = &&C7b

Generated code (2)
int cycle() {
  void *next1;
  void *next2;
  void *next3;
  /* other next pointers */
  void *head1 = &&END_LEVEL_1;
  void *head2 = &&END_LEVEL_2;
  /* other level pointers */
  if (s1) { s1 = 0; goto N26; }
  else {
    s1 = 0;
    if (S) {
      s2 = 1; code0 = -1;
      sched7a; sched1a; sched3b;
      s3 = 2; sched6b;
      switch (s3) {
        case 0: sched6c; break;
        case 1:
          s3 = 1; code1 = -1;
          sched7a; sched1b; goto N38;
          break;
        default: s3 = 2; sched6b;
        break;
      }
    } else {
      s2 = 0; sched7b;
      N26: s2 = 0; sched7b;
    }
  }
  goto *head1;
}

Generated code (3)
if (s2) {
  s2 = 1;
  code0 = -1;
  sched7a; sched1a; sched3a;
  switch (s3) {
    case 0: sched6c; break;
    case 1:
      s3 = 1; code1 = -1;
      sched6a; sched2; goto N38;
      case 2:
        if (I) {
          s3 = 1; code1 = -1;
          sched7a; sched1b; sched3b; goto N38;
          if (s5) Q = 1;
          C1b: if (R) s5 = 1;
          else s5 = 0;
          code0 &= -(1 << 1);
          goto *next1;
        } else {
          if (R) A = 1;
          C3b: s4 = 1;
          else {
            code0 &= -(1 << 1);
            goto *next3;
          }
          END_LEVEL1: goto *head2;
        } else {
          N26: s2 = 0; sched7b;
          goto *head1;
        }
  }
  goto *head1;
}

Generated code (4)
C1a: if (s5) Q = 1;
C1b: if (R) s5 = 1;
else s5 = 0;
  code0 &= -(1 << 1);
  goto *next1;
C2: if (s6) sched4;
else s6 = 0;
  goto *next2;
C3a: if (s4) s4 = 0;
else {
  if (R) A = 1;
  C3b: s4 = 1;
  else {
    code0 &= -(1 << 1);
    goto *next3;
  }
  END_LEVEL1: goto *head2;
}

Linked Lists — initial state

Linked Lists – schedule C3a

Linked Lists – schedule C1b
Linked Lists – schedule C4

Level 0
/* Cluster 0 */
...goto*head1;

Level 1
C1a: C1b: ... goto*next1;
C2: ... goto*next2;
C3a: C3b: ... goto*next3;
END
LEVEL1:
... goto *head2;

Level 2
C4: ... goto*next4;
END
LEVEL2:
... goto *head3;

Results (seconds/1 000 000 cycles)

<table>
<thead>
<tr>
<th>Example</th>
<th>Size</th>
<th>Clusters</th>
<th>Levels</th>
<th>C/L</th>
<th>Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>atds</td>
<td>622</td>
<td>156</td>
<td>16</td>
<td>9.8</td>
<td>138</td>
</tr>
<tr>
<td>Chorus</td>
<td>3893</td>
<td>662</td>
<td>22</td>
<td>30.1</td>
<td>563</td>
</tr>
<tr>
<td>mca200</td>
<td>5354</td>
<td>148</td>
<td>15</td>
<td>9.9</td>
<td>135</td>
</tr>
<tr>
<td>tcint</td>
<td>357</td>
<td>101</td>
<td>19</td>
<td>5.3</td>
<td>85</td>
</tr>
<tr>
<td>Wristwatch</td>
<td>360</td>
<td>87</td>
<td>13</td>
<td>6.7</td>
<td>87</td>
</tr>
</tbody>
</table>

Program Dependence Graphs

- Ferrante, Mace & Simons, 1984: Using PDG
- Cytron et al., 1991: Generating PDG
- Simons & Ferrante, 1993: External Edge
- Our approach: Natural Concurrent Programs

From PDG to SCFG: Trivial?

Make it sequential directly
Execute one by one

From PDG to SCFG: Non-trivial

No way to be sequential unless
to add guard variable or copy

An Example: Reconstructing PDG

Our Technique 3: Program Dependence Graphs
More complex situations:

```c
if (B) {
    V = 1;
    A = 1;
} else
    V = 0;
if (A)
    C = 1;
else
    C = C + 1;
```
More complex situations:
more forks & more data flow

Experimental Results

Summary

What To Understand About Esterel
Synchronous model of time
  • Time divided into sequence of discrete instants
  • Instructions either run and terminate in the same instant or explicitly in later instants

Idea of signals and broadcast
  • “Variables” that take exactly one value each instant and don’t persist
  • Coherence rule: all writers run before any readers

Causality Issues
  • Contradictory programs
  • How Esterel decides whether a program is correct

What To Understand About Esterel
Compilation techniques
  • Automata: Fast code, Doesn’t scale
  • Netlists: Scales well, Slow code, Good for causality
  • Control-flow: Scales well, Fast code, Bad at causality
  • Discrete Events: Scales well, Fast code, Better with more concurrency
  • PDG: Scales well, best yet for many examples

Generated C code for examples running on 2.5 GHz Pentium 4, Linux