A Deterministic Concurrent Language for Embedded Systems

Stephen A. Edwards
Columbia University

Joint work with Olivier Tardieu
Definition

**shim** \( 	ext{ˈʃɪm} \) *n*

1: a thin often tapered piece of material (as wood, metal, or stone) used to fill in space between things (as for support, leveling, or adjustment of fit).

2: *Software/Hardware Integration Medium*, a model for describing hardware/software systems
Robby Roto

(Bally/Midway 1981)
Robby Roto Block Diagram

- Z80
- Bus Bridge (Mux)
- ROM 40K
- SRAM 6K
- NV SRAM 2K
- blitter
- Custom Address
- Custom Data
- DRAM 16K
- Video
- Switches Audio Left
- Sound I/O
- Audio Right
- Sound I/O
<table>
<thead>
<tr>
<th>Software</th>
<th>Blitter</th>
<th>Memory</th>
<th>Video</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Blit</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Pixels</td>
<td>Line</td>
</tr>
<tr>
<td></td>
<td>Blit</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Pixels</td>
<td>Line</td>
</tr>
<tr>
<td></td>
<td>Blit</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Pixels</td>
<td>Line</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Interrupt</td>
<td>Line</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
SHIM Wishlist

- **Concurrent**
  Hardware always concurrent

- **Mixes synchronous and asynchronous styles**
  Need multi-rate for hardware/software systems

- **Only requires bounded resources**
  Hardware resources fundamentally bounded

- **Formal semantics**
  Do not want arguments about what something means

- **Scheduling-independent**
  Want the functionality of a program to be definitive
  Always want simulated behavior to reflect reality
  Verify functionality and performance separately
The SHIM Model

Sequential processes

Unbuffered one-to-many communication channels exchange data tokens

Dynamic but statically predictable topology

Asynchronous

Synchronous communication events

Delay-insensitive: sequence of data through any channel independent of scheduling policy (the Kahn principle)

“Kahn networks with rendezvous communication”
Basic SHIM

An imperative language with familiar C/Java-like syntax

```c
int32 gcd(int32 a, int32 b) {
    while (a != b) {
        if (a > b)
            a -= b;
        else
            b -= a;
    }
    return a;
}
```

```c
struct foo {  // Composite types
    int x;
    bool y;
    uint15 z;
    int<-3,5> w;  // Explicit-width integers
    int8 p[10];  // Explicit-range integers
    bar q;  // Arrays
};// Recursive types
```
Four Additional Constructs

\[ stmt_1 \text{ par } stmt_2 \]  \quad \text{Run } stmt_1 \text{ and } stmt_2 \text{ concurrently}

\text{next } var \quad \text{Send or receive next value of } var

\text{try } stmt_1 \text{ catch( } exc \text{ ) } stmt_2 \quad \text{Define and handle an exception}

\text{throw } exc \quad \text{Raise an exception}
Concurrenty & \textit{par}

\textit{Par} statements run concurrently and asynchronously

Terminate when all terminate

Each thread gets private copies of variables; no sharing

Writing thread sets the variable’s final value

```c
void main() {
    int a = 3, b = 7, c = 1;
    {
        a = a + c;       // a ← 4, b = 7, c = 1
        a = a + b;       // a ← 11, b = 7, c = 1
    } par {
        b = b - c;       // a ← 3, b ← 6, c = 1
        b = b + a;       // a ← 3, b ← 9, c = 1
    }
    // a ← 11, b ← 9, c = 1
}
```
Restrictions

Both pass-by-reference and pass-by-value arguments
Simple syntactic constraints avoid nondeterministic races

```c
void f(int &x) { x = 1; } // x passed by reference
void g(int x) { x = 2; } // x passed by value

void main() {
    int a = 0, b = 0;
    a = 1; par b = a; // OK: a and b modified separately
    a = 1; par a = 2; // Error: a modified by both
    f(a); par f(b); // OK: a and b modified separately
    f(a); par g(a); // OK: a modified by f only
    g(a); par g(a); // OK: a not modified
    f(a); par f(a); // Error: a passed by reference twice
}
```
Communication & \textit{next}

“next a” reads or writes next value of variable $a$

Blocking: thread waits for all processes that know about $a$

```c
void f(int a) { // a is a copy of c
    a = 3; // change local copy
    next a; // receive (wait for g)
    // a now 5
}
void g(int &b) { // b is an alias of c
    next b = 5; // sets c and send (wait for f)
    // b now 5
}
void main() {
    int c = 0;
    f(c); par g(c);
}
```
Synchronization and Deadlocks

Blocking communication makes for potential deadlock

```
{ next a; next b; } par { next b; next a; }  // deadlocks
```

Only threads responsible for a variable must synchronize

```
{ next a; next b; } par next b; par next a;  // OK
```

When a thread terminates, it is no longer responsible

```
{ next a; next a; } par next a;  // OK
```

Philosophy: deadlocks easy to detect; races are too subtle

SHIM prefers deadlocks to races (always reproducible)
void main() {
    uint8 A, B, C;
    {
        // source: generate four values
        next A = 17;
        next A = 42;
        next A = 157;
        next A = 8;
    } par {
        // buf1: copy from input to output
        for (;;)
            next B = next A;
    } par {
        // buf2: copy, add 1 alternately
        for (; ;)
            next C = next B;
            next C = next B + 1;
    } par {
        // sink
        for (; ;)
            next C;
    }
}
Message Sequence Chart

```
{  
  d = 0;
  for (;;) {  
    e = d;
    while (e > 0) {  
      next c = 1;
      next c = e;
      e = e - 1;
    }  
    next c = 0;
    next d = d + 1;
  }
}
par {  
  a = b = 0;
  for (;;) {  
    do {  
      if (next c != 0)  
        a = a + next c;
    } while (c);
    b = b + 1;
  }
}
par {  
  for (;;) next d;
}
```
A bounded FIFO: compiler will analyze & expand

```c
void buffer1(int input, int &output) {
    for (;;)
        next output = next input;
}

void fifo(int n, int input, int &output) {
    if (n == 1)
        buffer1(input, output);
    else {
        int channel;
        buffer1(input, channel);
        par
            fifo(n-1, channel, output);
    }
}
```
Robby Roto in SHIM

while (player is alive)
  next start-of-frame;
  ...game logic...
  next more = true;
  next command = ...;
  ...game logic...
  next more = false;

for (; ;)
  while (next more)
    next command;
    Write to buffer
    next frame = buffer;

for (; ;)
  next start-of-frame;
  for each line
    next sync = ...;
    for each pixel
      next clock
      Read pixel
      next pixel = ...;
      buffer = next frame;
Exceptions

Sequential semantics are classical

```c
void main() {
    int i = 1;
    try {
        throw T;
        i = i * 2;  // Not executed
    } catch (T) {
        i = i * 3;  // Executed by throw T
    }
    // i = 3 on exit
}
```
void main() {
    int i = 0, j = 0;
    try {
        while (i < 5)
            next i = i + 1;
        throw T;
    } par {
        for (;;) {
            next j =
                next i + 1;
        }
    } par {
        for (;;) {
            next j;
        }
    }
    catch (T) {}
}
Generating Software from SHIM
Faking Concurrency in C

One function

```c
void run() {
    for (;;) {
        switch (pc1) {
            case 0:  // block A
                pc1 = 1;
                break;
            case 1:  // block C
                pc1 = 1;
                break;
        }
        switch (pc2) {
            case 0:  // block B
                pc2 = 1;
                break;
            case 1:  // block D
                pc2 = 1;
                break;
        }
    }
}
```

Multiple Functions

```c
void run() {
    for (;;)
        run1(), run2();
}
```

```c
void run1() {
    static pc1;
    switch (pc1) {
        case 0:  // block A
            pc1 = 1;
            return;
        case 1:  // block C
            pc1 = 1;
            return;
    }
}
```

```c
void run2() {
    static pc2;
    switch (pc2) {
        case 0:  // block B
            pc2 = 1;
            return;
        case 1:  // block D
            pc2 = 1;
            return;
    }
}
```

Tail Recursion

```c
void run1a() {
    block A
    *(sp++) = run2a;
    (*(­­sp))(); return;
}
```

```c
void run1b() {
    block C
    *(sp++) = run2b;
    (*(­­sp))(); return;
}
```

```c
void run2a() {
    block B
    *(sp++) = run1b;
    (*(­­sp))(); return;
}
```

```c
void run2b() {
    block D
    *(­­sp))(); return;
}
```
Faking Concurrency in C

One function

```c
void run() {
  for (;;) {
    switch (pc1) {
      case 0: block A
        pc1 = 1;
      break;
      case 1: block C
    }
    switch (pc2) {
      case 0: block B
        pc2 = 1;
      break;
      case 1: block D
    }
  }
}
```

Multiple Functions

```c
void run() {
  for (;;) {
    run1(), run2();
  }
}

void run1() {
  static pc1;
  switch (pc1) {
    case 0: block A
      pc1 = 1;
    return;
    case 1: block C
  }
}

void run2() {
  static pc2;
  switch (pc2) {
    case 0: block B
      pc2 = 1;
    return;
    case 1: block D
  }
}
```
Faking Concurrency in C

One function

```c
void run() {
    for (;;) {
        switch (pc1) {
            case 0: block A
                pc1 = 1;
                break;
            case 1: block C
        }
    }
    switch (pc2) {
        case 0: block B
            pc2 = 1;
            return;
        case 1: block D
    }
}
```

Multiple Functions

```c
void run1() {
    static pc1;
    switch (pc1) {
        case 0: block A
            pc1 = 1;
            return;
        case 1: block C
    }
}

void run2() {
    static pc2;
    switch (pc2) {
        case 0: block B
            pc2 = 1;
            return;
        case 1: block D
    }
}
```

Tail Recursion

```c
void run1a() {
    block A
    *(sp++) = run2a;
    (*(--sp))(); return;
}

void run1b() {
    block C
    *(sp++) = run2b;
    (*(--sp))(); return;
}

void run2a() {
    block B
    *(sp++) = run1b;
    (*(--sp))(); return;
}

void run2b() {
    block D
    (*(--sp))(); return;
}
```
void source(int32 &C) {
    bool b = 0;
    for (int32 a = 0 ; a < 42 ; ) {
        if (b) {
            next C = a;
        } else {
            for (int32 d = 0 ;
                 d < 10 ; ++d)
                a = a + 1;
        }
        b = ~b;
    }
}

Extended basic blocks...
Compiling Processes Together

Build an automaton through abstract simulation

State signature:

- Running/blocked status of each process
- Blocked on reading/writing status of each channel

*Trick: does not include control or data state of each process*
Abstract Simulation

```c
{ // buf1
    for (;;)
        next B = next A;
} par { // buf2
    for (;;)
        next C = next B;
        next C = next B + 1;
}
```
# Benchmarks

<table>
<thead>
<tr>
<th>Example</th>
<th>Lines</th>
<th>Processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Berkeley</td>
<td>36</td>
<td>3</td>
</tr>
<tr>
<td>Buffer2</td>
<td>25</td>
<td>4</td>
</tr>
<tr>
<td>Buffer3</td>
<td>26</td>
<td>5</td>
</tr>
<tr>
<td>Buffer10</td>
<td>33</td>
<td>12</td>
</tr>
<tr>
<td>Esterel1</td>
<td>144</td>
<td>5</td>
</tr>
<tr>
<td>Esterel2</td>
<td>127</td>
<td>5</td>
</tr>
<tr>
<td>FIR5</td>
<td>78</td>
<td>19</td>
</tr>
<tr>
<td>FIR19</td>
<td>190</td>
<td>75</td>
</tr>
</tbody>
</table>
## Executable Sizes

<table>
<thead>
<tr>
<th>Example</th>
<th>Switch</th>
<th>Tail-Recursive</th>
<th>Static (partial)</th>
<th>Static (full)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>size</td>
<td>states</td>
<td>size</td>
</tr>
<tr>
<td>Berkeley</td>
<td>860</td>
<td>1299</td>
<td>1033</td>
<td>5</td>
</tr>
<tr>
<td>Buffer2</td>
<td>832</td>
<td>1345</td>
<td>1407</td>
<td>10</td>
</tr>
<tr>
<td>Buffer3</td>
<td>996</td>
<td>1579</td>
<td>1771</td>
<td>20</td>
</tr>
<tr>
<td>Buffer10</td>
<td>2128</td>
<td>3249</td>
<td>5823</td>
<td>174</td>
</tr>
<tr>
<td>Esterel1</td>
<td>3640</td>
<td>5971</td>
<td>8371</td>
<td>49</td>
</tr>
<tr>
<td>Esterel2</td>
<td>4620</td>
<td>7303</td>
<td>6871</td>
<td>24</td>
</tr>
<tr>
<td>FIR5</td>
<td>4420</td>
<td>6863</td>
<td>6819</td>
<td>229</td>
</tr>
<tr>
<td>FIR19</td>
<td>17052</td>
<td>25967</td>
<td>67823</td>
<td>2819</td>
</tr>
</tbody>
</table>
## Speedups vs. Switch

<table>
<thead>
<tr>
<th>Example</th>
<th>Tail-Recursive</th>
<th>Static (partial)</th>
<th>Static (full)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Berkeley</td>
<td>2.9×</td>
<td>2.6</td>
<td>7.8</td>
</tr>
<tr>
<td>Buffer2</td>
<td>2.0</td>
<td>2.4</td>
<td>11</td>
</tr>
<tr>
<td>Buffer3</td>
<td>2.1</td>
<td>2.6</td>
<td>10</td>
</tr>
<tr>
<td>Buffer10</td>
<td>1.7</td>
<td>4.8</td>
<td>12</td>
</tr>
<tr>
<td>Esterel1</td>
<td>1.9</td>
<td>2.9</td>
<td>5.9</td>
</tr>
<tr>
<td>Esterel2</td>
<td>2.0</td>
<td>2.5</td>
<td>5.2</td>
</tr>
<tr>
<td>FIR5</td>
<td>0.92</td>
<td>4.8</td>
<td>7</td>
</tr>
<tr>
<td>FIR19</td>
<td>0.90</td>
<td>5.9</td>
<td>7.1</td>
</tr>
</tbody>
</table>
Conclusions

• The SHIM Model: Sequential processes communicating through rendezvous

• Sequential language plus
  • concurrency,
  • communication, and
  • exceptions.

• Scheduling-independent
  • Kahn networks with rendezvous
  • Nondeterministic scheduler produces deterministic behavior
Conclusions

• Software generation
  • Tail-recursion for simulating concurrency
  • Dynamic code maintains stack of function pointers to runnable processes
  • Processes compiled together w/ abstract simulation
Future Work

- Automata abstract communication patterns
  Useful for deadlock detection, protocol violation
Future Work

• Automata abstract communication patterns
  Useful for deadlock detection, protocol violation

• Synthesis for multicore processors
  Compile together the processes on each core
Future Work

- Automata abstract communication patterns
  Useful for deadlock detection, protocol violation

- Synthesis for multicore processors
  Compile together the processes on each core

- Hardware/software cosynthesis
  Bounded subset has reasonable hardware semantics
Future Work

- Automata abstract communication patterns
  Useful for deadlock detection, protocol violation

- Synthesis for multicore processors
  Compile together the processes on each core

- Hardware/software cosynthesis
  Bounded subset has reasonable hardware semantics

- Convince world: deterministic concurrency is good