SHIM
A Deterministic Concurrent Language for Embedded Systems

Stephen A. Edwards
Columbia University

Joint work with Olivier Tardieu
Definition

shim  

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)
HW/SW Interaction

**Software** | **Blitter** | **Memory** | **Video**
---|---|---|---
Blit | Pixels | Line | Interrupt
Blit | Pixels | Line | Line
Blit | Pixels | Line | Line

Interrupt

**SHIM:** A Deterministic Concurrent Language for Embedded Systems – p. 5/38
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”
Modeling Time in SHIM

SHIM is timing-independent

Philosophy: separate functional requirements from performance requirements

Like synchronous digital logic: establish correct function independent of timing, then check and correct performance errors

Vision: clock processes impose execution rates, checked through static timing analysis
int32 gcd(int32 a, int32 b) {
    while (a != b) {
        if (a > b)
            a -= b;
        else
            b -= a;
    }
    return a;
}

struct foo {
    int x;
    bool y;
    uint15 z;
    int<-3,5> w;
    int8 p[10];
    bar q;
};
Four Additional Constructs

- \( stmt_1 \text{ par } stmt_2 \)  Run \( stmt_1 \) and \( stmt_2 \) concurrently
- next \( var \)  Send or receive next value of \( var \)
- try \( stmt_1 \) catch( \( exc \) ) \( stmt_2 \)  Define and handle an exception
- throw \( exc \)  Raise an exception
Concurrency & \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 \rightarrow 4, b = 7, c = 1
        a = a + b; // a \rightarrow 11, b = 7, c = 1
    } par {
        b = b - c; // a = 3, b \rightarrow 6, c = 1
        b = b + a; // a = 3, b \rightarrow 9, c = 1
    }
    // a \rightarrow 11, b \rightarrow 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 \textit{a}
Blocking: thread waits for all processes that know about \textit{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
    b = 5; // sets c
    next b; // send (wait for f)
    // b still 5
}
void main() {
    int c = 0;
    f(c); par g(c);
}
```
Synchronization and Deadlocks

Blocking communication makes for potential deadlock

\{ \text{next } a; \text{next } b; \} \text{ par } \{ \text{next } b; \text{next } a; \} \quad \text{// deadlocks}

Only threads responsible for a variable must synchronize

\{ \text{next } a; \text{next } b; \} \text{ par } \text{next } b; \text{par } \text{next } a; \quad \text{// OK}

When a thread terminates, it is no longer responsible

\{ \text{next } a; \text{next } a; \} \text{ par } \text{next } a; \quad \text{// OK}

Philosophy: deadlocks easy to detect; races are too subtle

SHIM prefers deadlocks to races (always reproducible)
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 i;
            j = i + 1;
            next j;
        }
    } par {
        for (;;) {
            next j;
        }
    } catch (T) {}
}

Exceptions propagate through communication actions to preserve determinism

Idea: “transitive poisoning”

Raising an exception “poisons” a process

Any process attempting to communicate with a poisoned process is itself poisoned (within exception scope)

“Best effort preemption”
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;
        }
        C = next B + 1;
    } par {    // sink
        for (; ;)
            next C;
    }
}
Communication Patterns

```c
{  
  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;
}
```
Recursion & Concurrency

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

Software

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

Blitter

for (;;) 
start-of-frame 
next start-of-frame;
next command;
Write to buffer
next frame = buffer;

Video out

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

        switch (pc2) {
            case 0: block B
                pc2 = 1;
                break;
            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;
                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
    }
}
```

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;
}
```
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 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
    }
}
```

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;
    }
    Exit
}

Extended basic blocks...
Global Data

- Stack of pointers to runnable functions
- Each channel holds pointer to function (process) to resume after communication
- **Blocked** flag for each process
- Each process broken into tail-recursive atomic functions

```c
void (*stack[3])(void);
void (**sp)(void);

struct channel {
    void (*reader)(void);
    void (*writer)(void);
};

struct channel A_ch = { 0, 0 };
struct channel B_ch = { 0, 0 };
struct channel C_ch = { 0, 0 };

unsigned char A, B, C;

struct {
    char blocked;
} source = { 0 };

struct {
    char blocked;
    unsigned char tmp;
} buf1 = { 0 };

struct {
    char blocked;
    unsigned char tmp;
} buf2 = { 0 };

struct {
    char blocked;
} sink = { 0 };
```
void source_0() {  
    A = 17;
    if (buf1.blocked && A_ch.reader) {    
        buf1.blocked = 0;
        *(sp++) = A_ch.reader;
        A_ch.reader = 0;
    }
    source.blocked = 1;
    A_ch.writer = source_1;
    (*(­­sp++)()); return;
}

void source_1() {  
      // Continue here after the write
    /* ... */
}

// Write to channel
// If buffer is blocked reading,
// Unblock A
// schedule the reader, and
// clear the channel.
// Block us
// to continue below
// Run next process
void buf1_0() {
    if (source.blocked && A_ch.writer) { // If source is blocked,
        buf1_1(); return; // “goto” buf1_1
    }
    buf1.blocked = 1; // Block us
    A_ch.reader = buf1_1; // to continue below
    (*(­­sp))(); return; // Run next process
}

void buf1_1() {
    buf1.tmp = A; // Read from the channel
    source.blocked = 0; // Unblock the source,
    *(sp++) = A_ch.writer; // schedule it, and
    A_ch.writer = 0; // clear the channel.
}
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

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

buf1 ready
buf2 blocked
A clear
C waiting for writer
B waiting for reader

buf1 PCs
buf2 PCs

{1} {4}
Abstract Simulation

```cpp
{ // buf1
  for (;;) {
    next B = next A;
  }
} par {
  // buf2
  for (;;) {
    next C = next B;
    next C = next B + 1;
  }
}
```
Abstract Simulation

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

buf1 ready
buf2 blocked
A clear
B waiting for reader
C waiting for writer

buf1 PCs
buf2 PCs
{1, 2} {3}
Abstract Simulation

{ // buf1
    1 for (;;) {
        2 next B = 3 next A;
    }
} par { // buf2
    4 for (;;) {
        5 next C = 6 next B;
        7 next C = 8 next B + 1;
    }
}

buf1 ready
buf2 blocked
buf1 PCs
buf2 PCs

A clear
B waiting for reader
C waiting for writer
Abstract Simulation

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

buf1 ready
buf2 blocked
buf1 PCs
buf2 PCs
{1, 2} {3}
A clear
C waiting for writer
B waiting for reader

{1} {4}
buf2

{1} {6}
buf1

{3} {6}
receive A

{3} {6}
buf1

{2} {6}
Abstract Simulation

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

buf1 ready
buf2 blocked
buf1 PCs
buf2 PCs
{1, 2} {3}
A clear
C waiting for writer
B waiting for reader

buf1

{1} {4}
buf2

{1} {6}
buf1

{3} {6}
receive A

{3} {6}
buf1

{2} {6}
buf1

{2} {5}
buf2
Abstract Simulation

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

buf1 ready
buf2 blocked
buf1 PCs
buf2 PCs

A clear
C waiting for writer
B waiting for reader

{1, 2} {3}
{1, 2} {3}
{2} {6}
{2} {5}
{3} {5}
{1} {6}
{1} {6}
{3} {6}
{3} {6}

receive A
buf2
buf1
buf1
buf2
buf1
buf1
}
```
Abstract Simulation

```
{ // buf1
  for (;;)
    next B = next A;
} par { // buf2
  for (;;)
    next C = next B;
    next C = next B + 1;
}
```
Abstract Simulation

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

SHIM:A Deterministic Concurrent Language for Embedded Systems – p. 28/38
Abstract Simulation

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

```plaintext
{ // buf1
  1 for (;;) {
    2 next B = 3 next A;
  }
  par { // buf2
    4 for (;;) {
      5 next C = 6 next B;
      7 next C = 8 next B + 1;
    }
  }
}
```

SHIM: A Deterministic Concurrent Language for Embedded Systems – p. 28/38
Abstract Simulation

```plaintext
\{ // buf1
  for (;;)
    next B = next A;
\} par // buf2
  for (;;) {
    next C = next B;
    next C = next B + 1;
  }
```

buf1 ready

buf2 blocked

buf1 PCs

buf2 PCs

{1, 2} {3}

A clear

C waiting for writer

B waiting for reader

{3} {5, 7}

receive A

send C

{2} {5, 6}

{2} {5, 6}

buf2

{2} {5, 7}

buf1

{2} {5}

buf1 PCs

buf2 PCs

{1} {3}

{1} {6}

receive A

{3} {6}

buf1

{3} {6}

buf1

{1, 2} {3}

SHIM:A Deterministic Concurrent Language for Embedded Systems – p. 28/38
Abstract Simulation

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

- buf1 ready
- buf2 blocked
- buf1 PCs
- buf2 PCs
- A clear
- C waiting for writer
- B waiting for reader

SHIM: A Deterministic Concurrent Language for Embedded Systems – p. 28/38
Abstract Simulation

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

buf1 ready

buf2 blocked

buf1 PCs

buf2 PCs

{1, 2} {3}

A clear

C waiting for writer

B waiting for reader

send C

{2} {5, 7}

receive A

{3} {5, 7}

receive A

{3} {5, 7}

SHIM:A Deterministic Concurrent Language for Embedded Systems – p. 28/38
Abstract Simulation

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

SHIM:A Deterministic Concurrent Language for Embedded Systems – p. 28/38
Abstract Simulation

```c
{ for (;;) {
    1 next B = 2 next A;
} par {
    // buf1
    4 for (;;) {
        5 next C = 6 next B;
        7 next C = 8 next B + 1;
    }
}
}
```

buf1 ready
buf2 blocked

buf1 PCs
buf2 PCs

A clear
C waiting for writer
B waiting for reader

receive A
send C
receive A

SHIM: A Deterministic Concurrent Language for Embedded Systems – p. 28/38
Abstract Simulation

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

SHIM: A Deterministic Concurrent Language for Embedded Systems – p. 28/38
Abstract Simulation

```plaintext
{1}
for (;;)
  next B = next A;
} par {
  {4}
for (;;)
  next C = next B;
  next C = next B + 1;
}
```

SHIM: A Deterministic Concurrent Language for Embedded Systems – p. 28/38
Abstract Simulation

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

SHIM: A Deterministic Concurrent Language for Embedded Systems – p. 28/38
Abstract Simulation

```plaintext
{ // buf1
  for (;;)
    next B = next A;
} par {
  // buf2
  for (;;)
    next C = next B;
    next C = next B + 1;
}
```
Abstract Simulation

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

buf1 ready
buf2 blocked
buf1 PCs
buf2 PCs

A clear
C waiting for writer
B waiting for reader

{1, 2} {3}
{2} {5, 7}
{3} {5, 7}

receive A
send C

{1} {6}
{3} {6, 8}
{2} {5, 6, 7, 8}
{3} {5, 7}

receive A
send C

{1} {4}
{2} {5, 7}
{3} {5, 7}

{2} {5, 7}
{3} {5, 7}

{3} {5, 7}
Abstract Simulation

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

buf1 ready

buf2 blocked

buf1 PCs

buf2 PCs

A clear

C waiting for writer

B waiting for reader

receive A

send C

receive A

send C

buf1 PCs

buf2 PCs

{1, 2} {3}

{1} {4}

buf2

{1} {6}

buf1

{3} {6, 8}

receive A

{3} {6, 8}

buf1

{2} {5, 6, 7, 8}

buf2

{2} {5, 7}

{3} {5, 7}

buf1

{3} {5, 7}

{3} {5, 7}

{3} {5, 7}
Abstract Simulation

```c
{ // buf1
1 for (;;)
2 next B = 3 next A;
} par { // buf2
4 for (;;)
5 next C = 6 next B;
7 next C = 8 next B + 1;
}
```

SHIM: A Deterministic Concurrent Language for Embedded Systems – p. 28/38
## 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 Recursive</th>
<th>Tail Recursive</th>
<th>Static (partial) size</th>
<th>Static (partial) states</th>
<th>Static (full) size</th>
<th>Static (full) states</th>
</tr>
</thead>
<tbody>
<tr>
<td>Berkeley</td>
<td>860</td>
<td>1299</td>
<td>1033</td>
<td>5</td>
<td>551</td>
<td>6</td>
</tr>
<tr>
<td>Buffer2</td>
<td>832</td>
<td>1345</td>
<td>1407</td>
<td>10</td>
<td>403</td>
<td>8</td>
</tr>
<tr>
<td>Buffer3</td>
<td>996</td>
<td>1579</td>
<td>1771</td>
<td>20</td>
<td>443</td>
<td>10</td>
</tr>
<tr>
<td>Buffer10</td>
<td>2128</td>
<td>3249</td>
<td>5823</td>
<td>174</td>
<td>687</td>
<td>24</td>
</tr>
<tr>
<td>Esterel1</td>
<td>3640</td>
<td>5971</td>
<td>8371</td>
<td>49</td>
<td>5611</td>
<td>56</td>
</tr>
<tr>
<td>Esterel2</td>
<td>4620</td>
<td>7303</td>
<td>6871</td>
<td>24</td>
<td>2539</td>
<td>18</td>
</tr>
<tr>
<td>FIR5</td>
<td>4420</td>
<td>6863</td>
<td>6819</td>
<td>229</td>
<td>1663</td>
<td>79</td>
</tr>
<tr>
<td>FIR19</td>
<td>17052</td>
<td>25967</td>
<td>67823</td>
<td>2819</td>
<td>7287</td>
<td>372</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>
Generating Hardware from SHIM
Hardware IR

Assignment

CFG Node

Decision

Datapath Fragment

Cycle Boundary

Merge
Translation Patterns

if (e) $s_1$ else $s_2$

while (e) $s$

next $c = e$

$v = \text{next } c$
Hardware Translation

```
{ 
  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;
}
```
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 with abstract simulation

- Hardware generation
  - Hardware IR: actions, decisions, merges, and registers
  - Syntax-directed translation from IR
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