SHIM

A Deterministic Concurrent Language for Embedded Systems

Stephen A. Edwards

Columbia University

Joint work with Olivier Tardieu

Definition

shim \'shim\ *n*

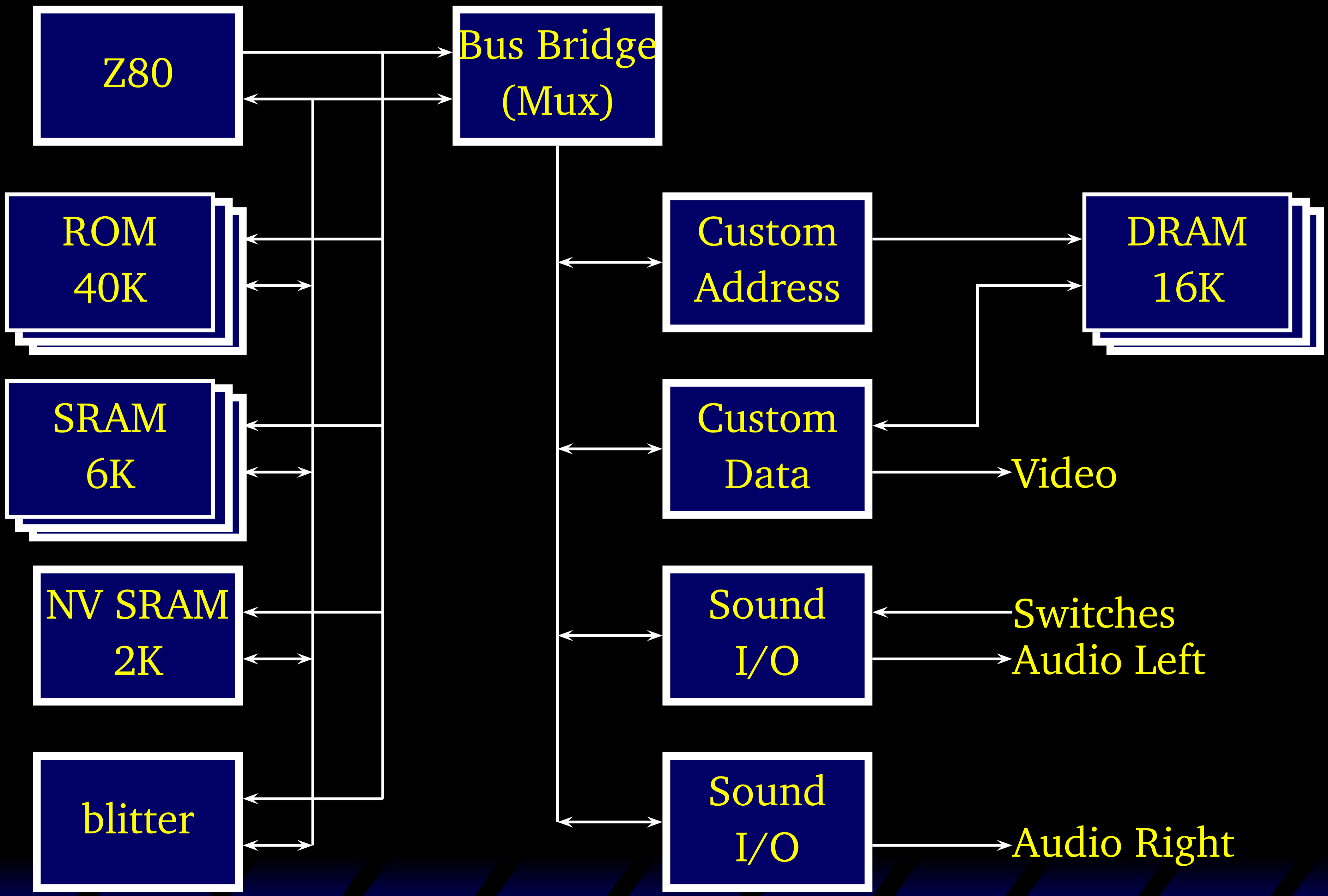
1 : a thin often tapered piece of material (as wood, metal, or stone) used to fill in space between things (as for support, leveling, or adjustment of fit).

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

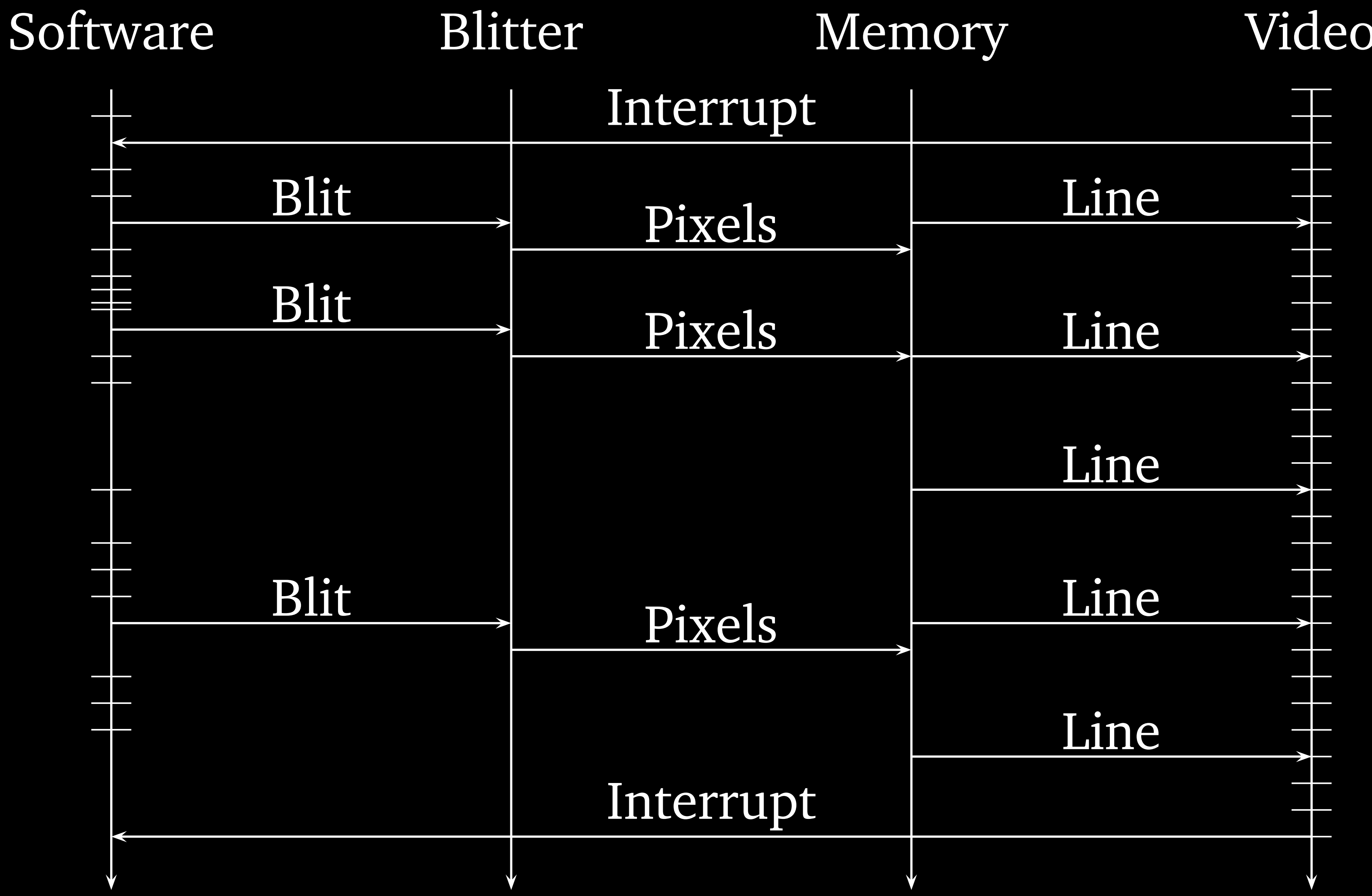
Robby Roto

(Bally/Midway 1981)

Robby Roto Block Diagram



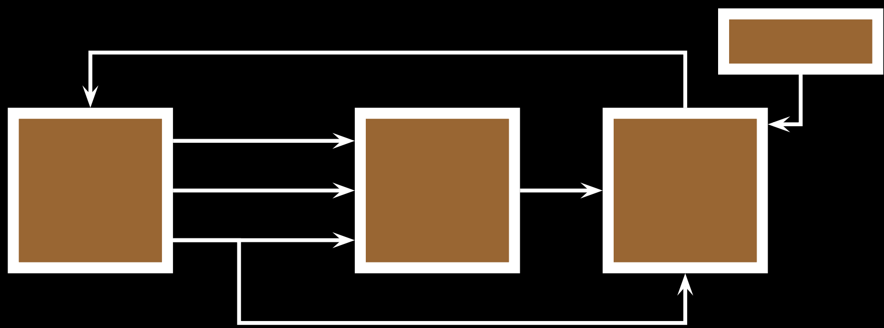
HW/SW Interaction



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

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

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

Four Additional Constructs

$stmt_1$ `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 & *par*

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

```
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

```
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 & *next*

"next a" reads or writes next value of variable *a*

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

```
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

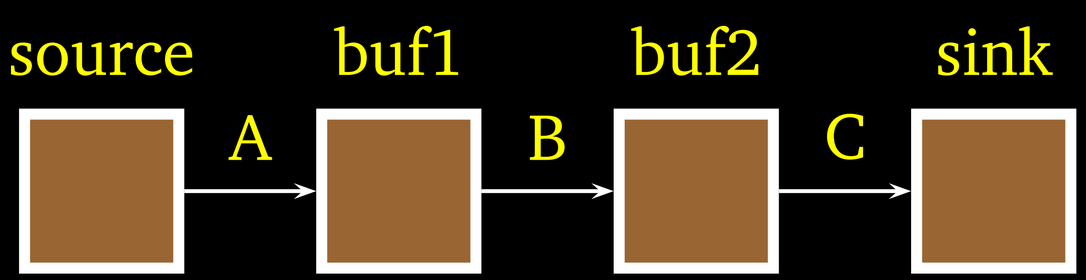
```
{ next a; next a; } par next a;  // OK
```

Philosophy: deadlocks easy to detect; races are too subtle

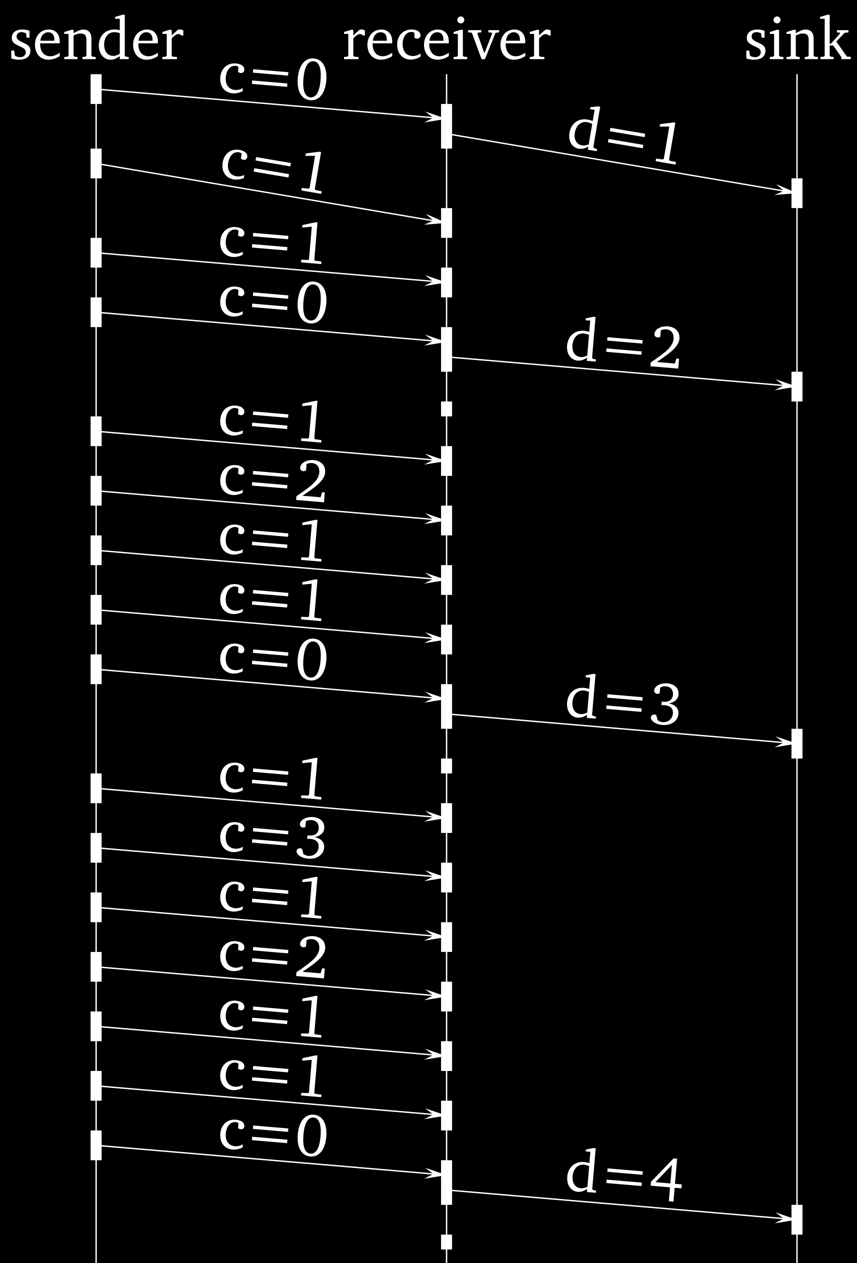
SHIM prefers deadlocks to races (always reproducible)

An Example

```
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

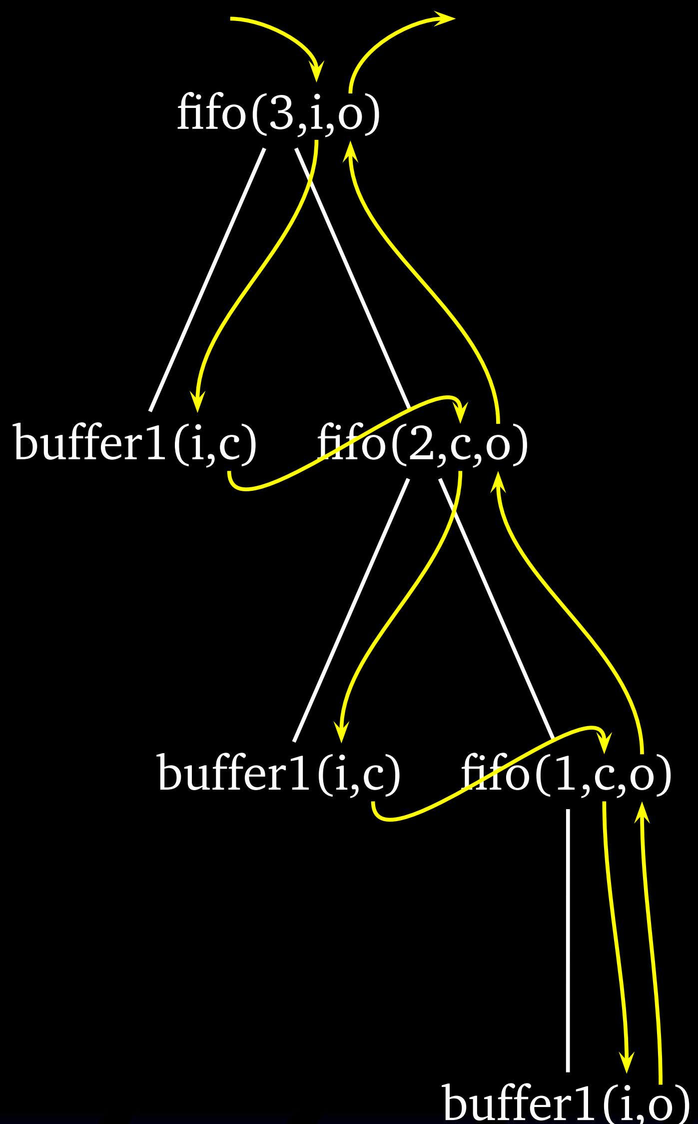



```
{
  d = 0;
  for (;;) {
    e = d;
    while (e > 0) {
      next c = 1;
      next c = e;
      e = e - 1;
    }
    next c = 0;
    next d = d + 1;
  }
} par {
  a = b = 0;
  for (;;) {
    do {
      if (next c != 0)
        a = a + next c;
    } while (c);
    b = b + 1;
  }
} par {
  for (;;) next d;
}
```

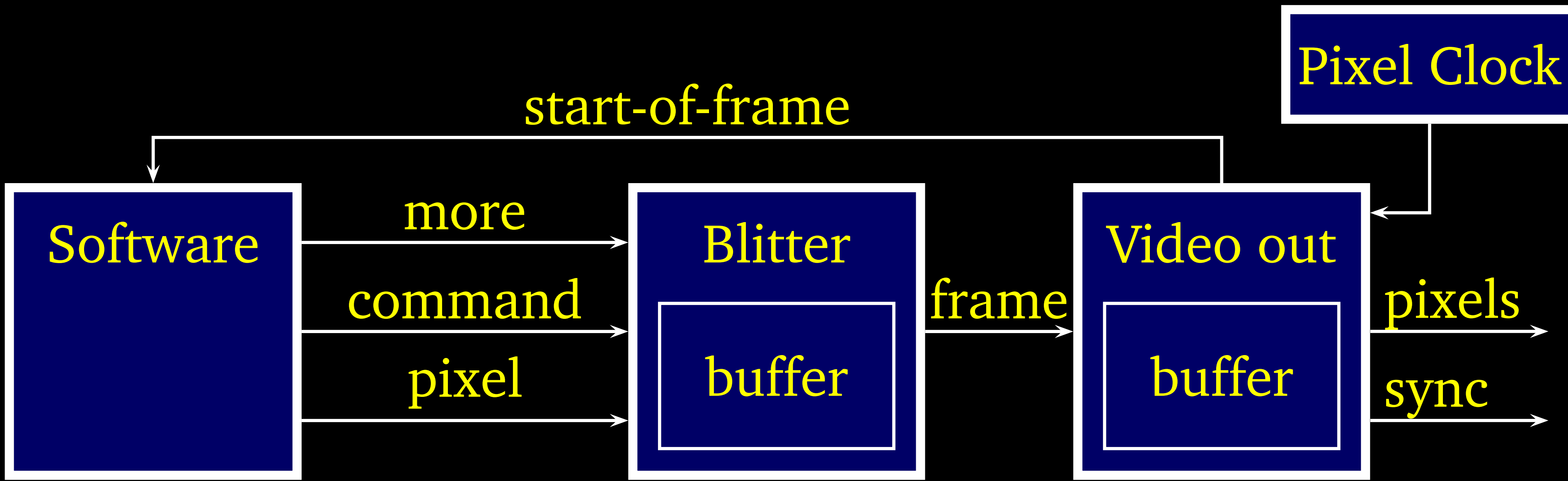
Recursion & Concurrency

A bounded FIFO: compiler will analyze & expand

```
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

```
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
}
```

Exceptions & Concurrency

```
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) {}
}
```

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”

Generating Software from SHIM

Faking Concurrency in C

One function

```
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

```
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

```
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

```
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

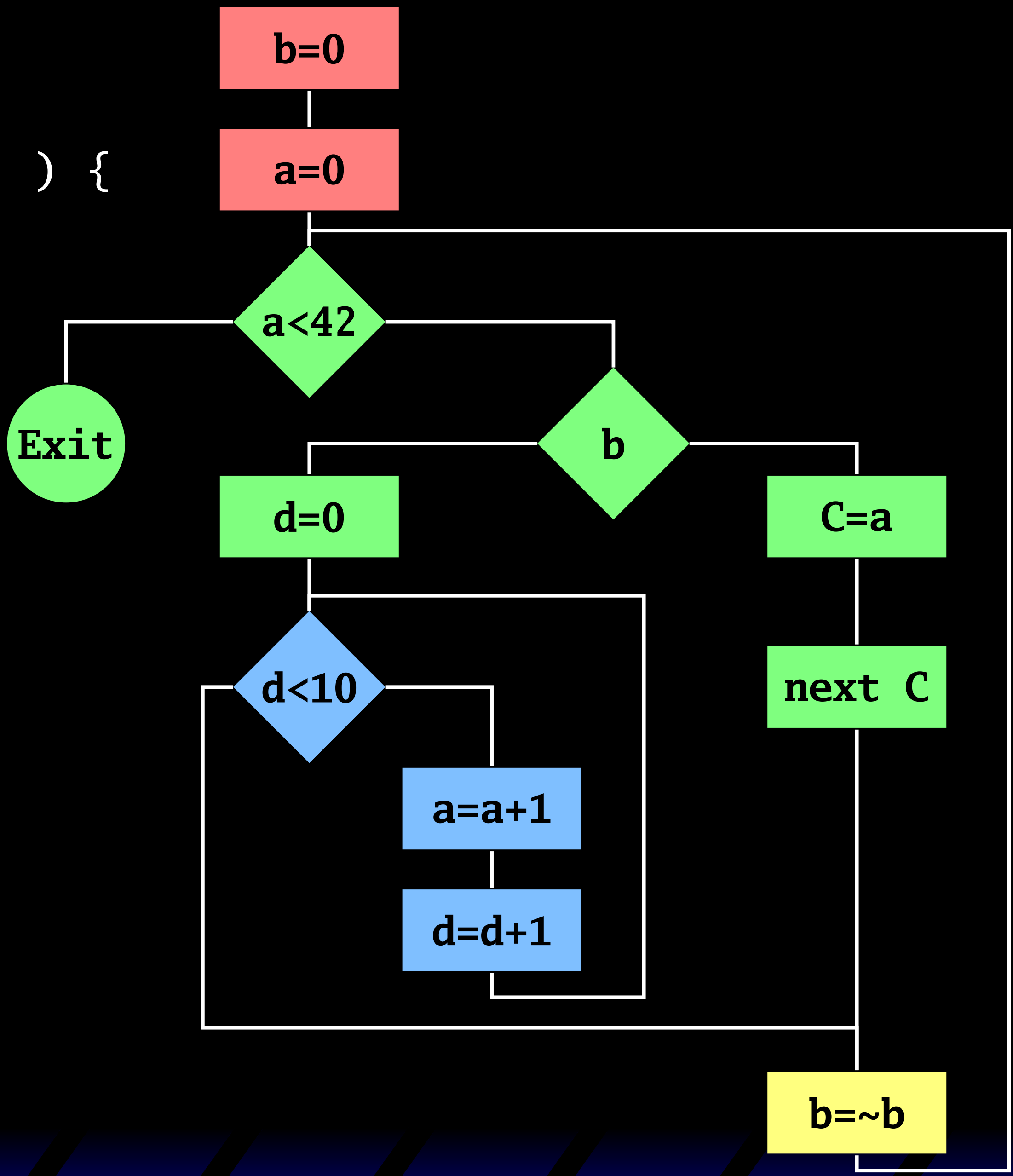
```
void run() {
  for (;;)
    run1(), run2();
}

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

```
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;
}
```

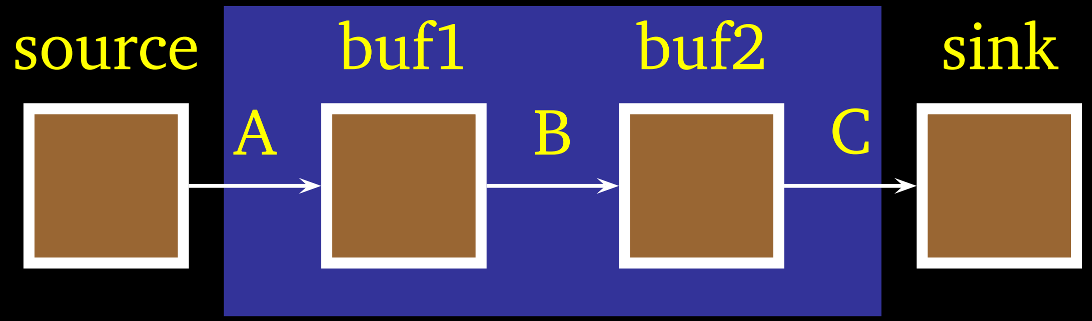
Dividing into Fragments

```
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

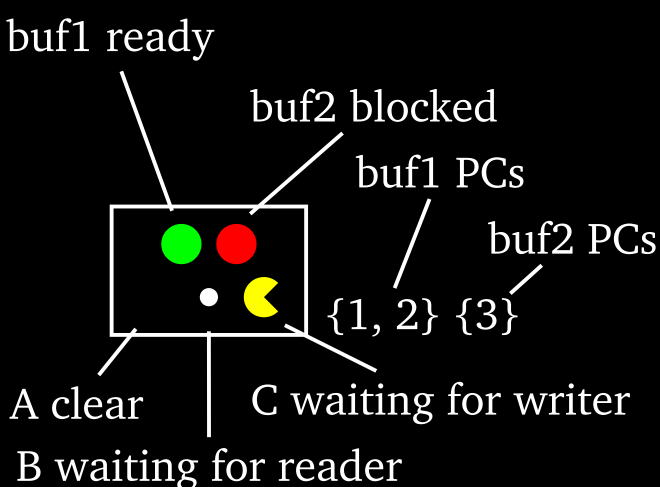
State signature:

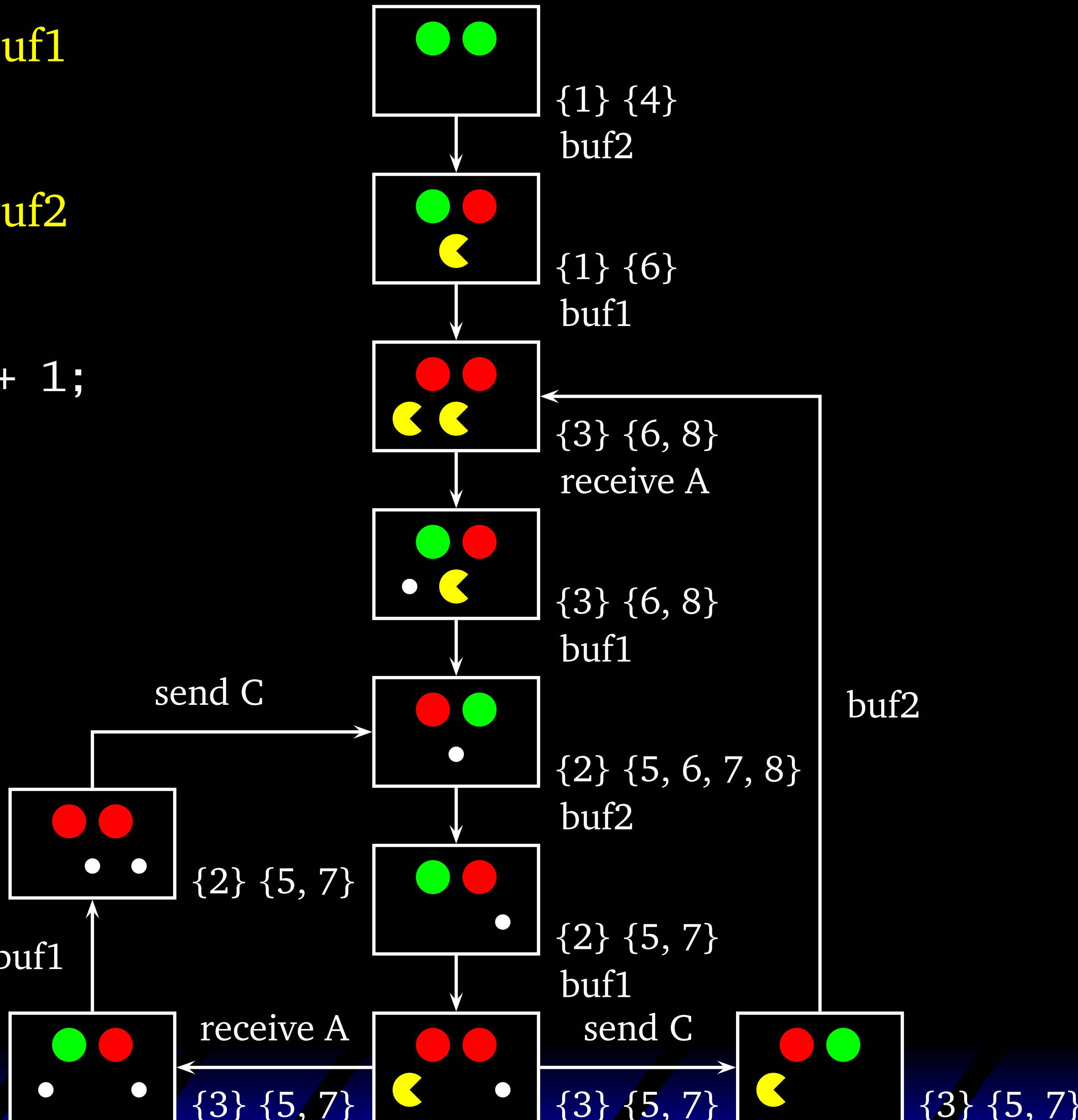
- Running/blocked status of each process
- Blocked on reading/writing status of each channel

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

Abstract Simulation

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





Benchmarks

Example	Lines	Processes
Berkeley	36	3
Buffer2	25	4
Buffer3	26	5
Buffer10	33	12
Esterel1	144	5
Esterel2	127	5
FIR5	78	19
FIR19	190	75

Executable Sizes

Example	Switch	Tail-Recursive	Static (partial) size	Static (partial) states	Static (full) size	Static (full) states
Berkeley	860	1299	1033	5	551	6
Buffer2	832	1345	1407	10	403	8
Buffer3	996	1579	1771	20	443	10
Buffer10	2128	3249	5823	174	687	24
Esterel1	3640	5971	8371	49	5611	56
Esterel2	4620	7303	6871	24	2539	18
FIR5	4420	6863	6819	229	1663	79
FIR19	17052	25967	67823	2819	7287	372

Speedups vs. Switch

Example	Tail-Recursive	Static (partial)	Static (full)
Berkeley	2.9×	2.6	7.8
Buffer2	2.0	2.4	11
Buffer3	2.1	2.6	10
Buffer10	1.7	4.8	12
Esterel1	1.9	2.9	5.9
Esterel2	2.0	2.5	5.2
FIR5	0.92	4.8	7
FIR19	0.90	5.9	7.1

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