Concurrency and Communication: Lessons from the SHIM Project

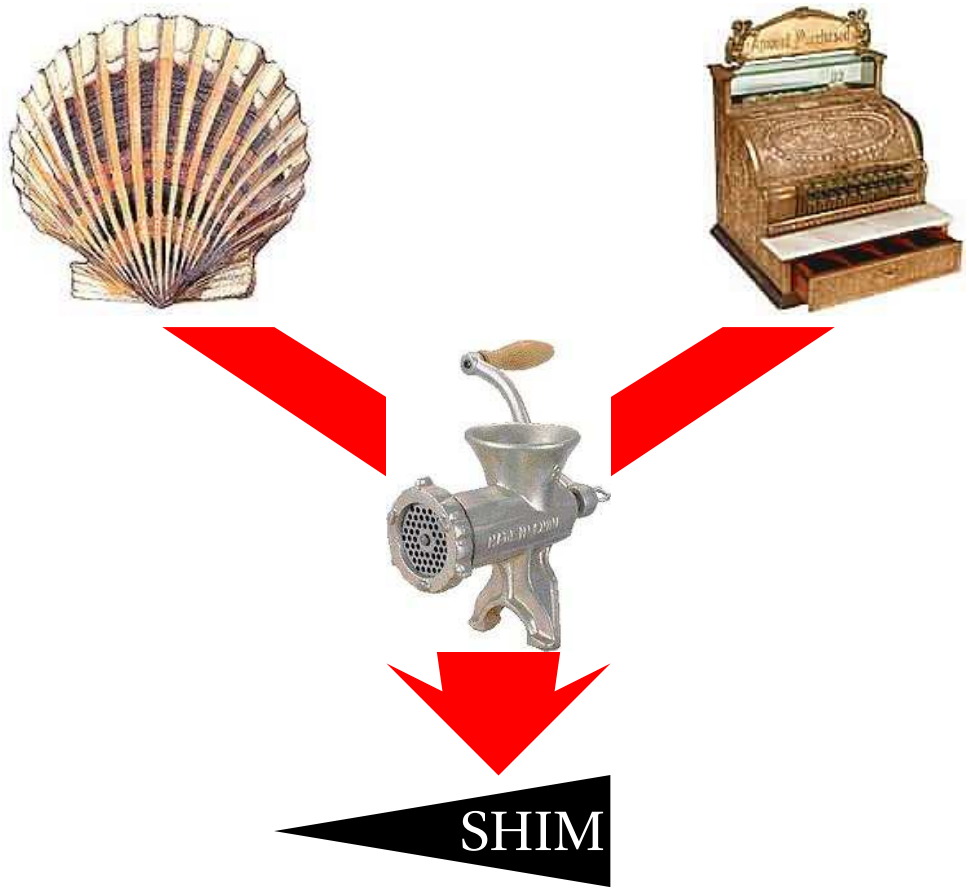
Stephen A. Edwards

Columbia University

SHIM

Motivation, Version 1.0

[Edwards, Dagstuhl 2004]



Combine two well-known semantic models in a single language:

Single-threaded software

Synchronous RTL hardware

The SHIM Language, Version 1.0

[Edwards, Dagstuhl 2004]

```
module timer {
  shared uint:32 counter; // Visible to HW and SW

  hw void count() { // Hardware process
    counter = counter + 1;
  }

  out void reset_timer() { // Software function
    counter = 0;
  }

  out uint get_time() { // Software function
    return counter;
  }
}
```

The problem: Nondeterminism

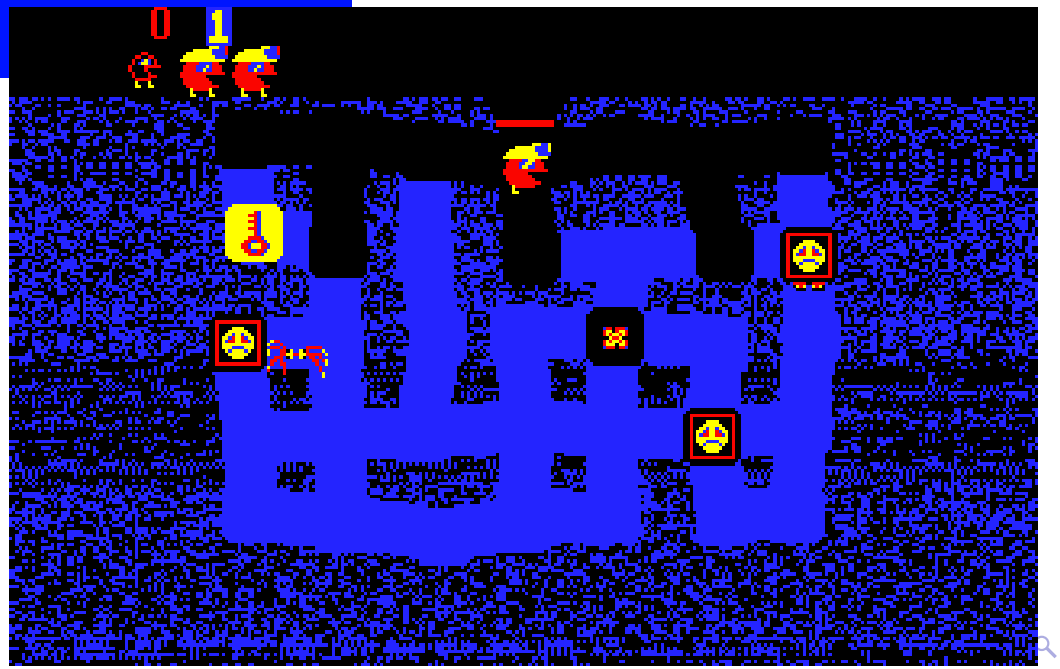
[Edwards, Dagstuhl 2004]

$$\psi_{\text{kitty}} = \frac{1}{\sqrt{2}}\psi_{\text{alive}} + \frac{1}{\sqrt{2}}\psi_{\text{dead}}$$

http://thenerdiestshirts.com/

Robby Roto

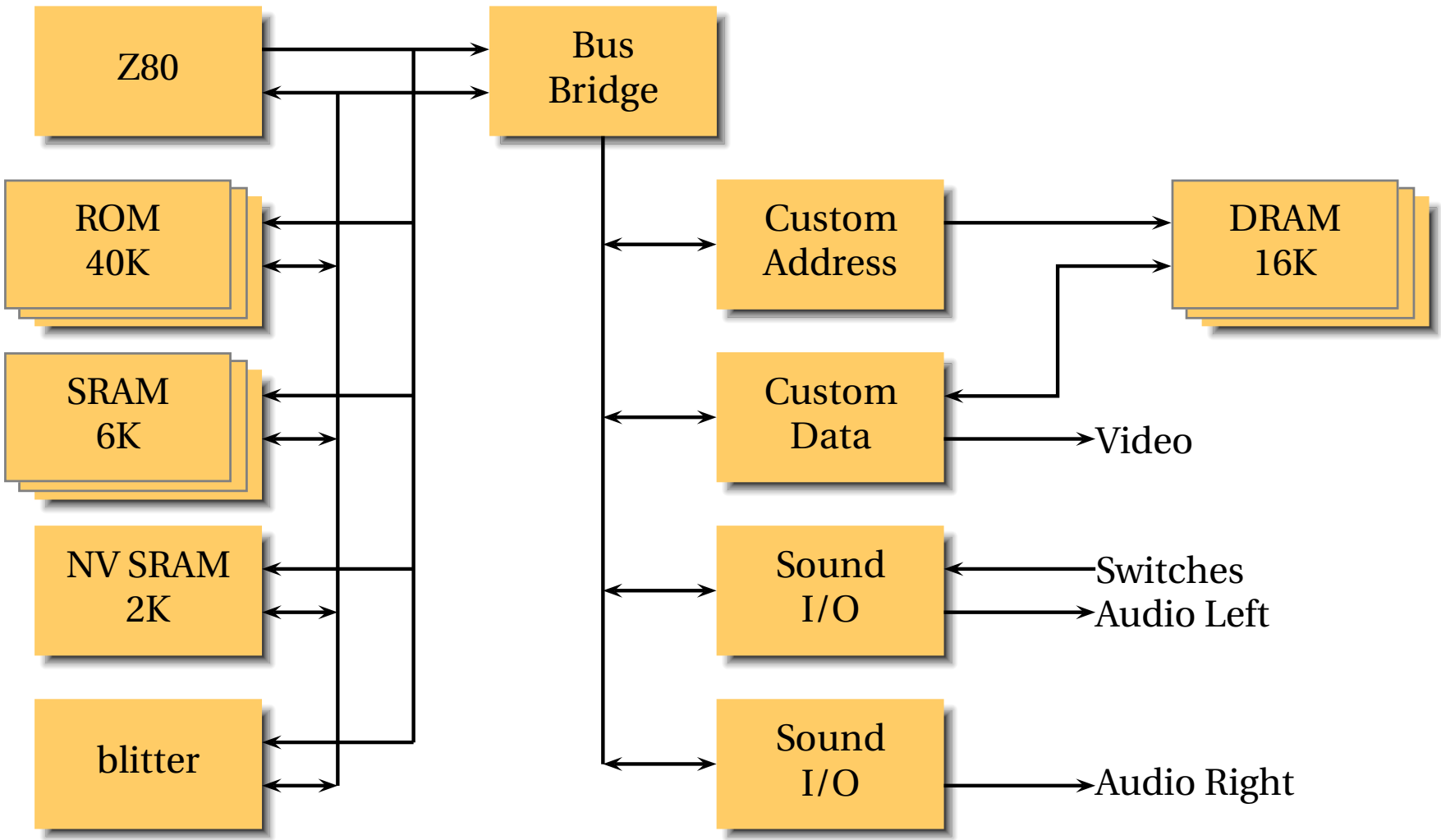
[Edwards and Tardieu, Emsoft 2005]



(Bally/Midway 1981)

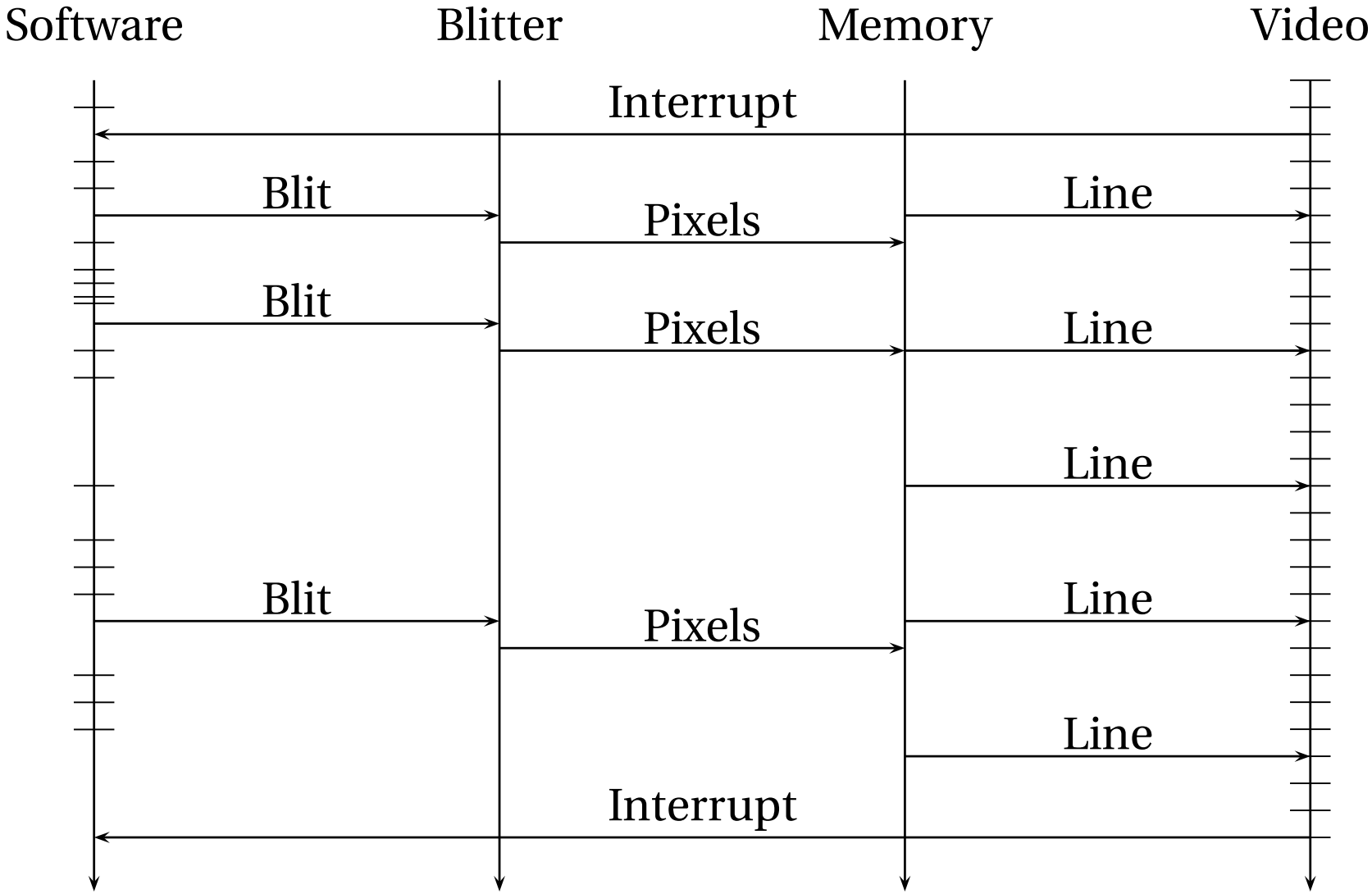
Robby Roto Block Diagram

[Edwards and Tardieu, Emsoft 2005]



HW/SW Interaction

[Edwards and Tardieu, Emsoft 2005]



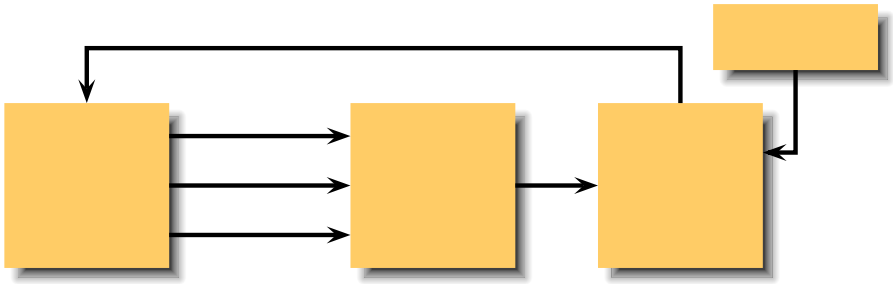
SHIM Wishlist

- *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

[Edwards and Tardieu, Emsoft 2005]

The SHIM Model

[Edwards and Tardieu, Emsoft 2005]



Sequential processes

Unbuffered point-to-point communication channels exchange data tokens

Fixed 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”

Code Generation


```
process
sink(int32 B) {
  for (;;) B;
}

process
buffer(int32 &B,
       int32 A) {
  for (;;) B = A;
}

process
source(int32 &A) {
  A = 17; A = 42;
  A = 157; A = 8;
}

network main() {
  sink();
  buffer();
  source();
}
```

Code Generation

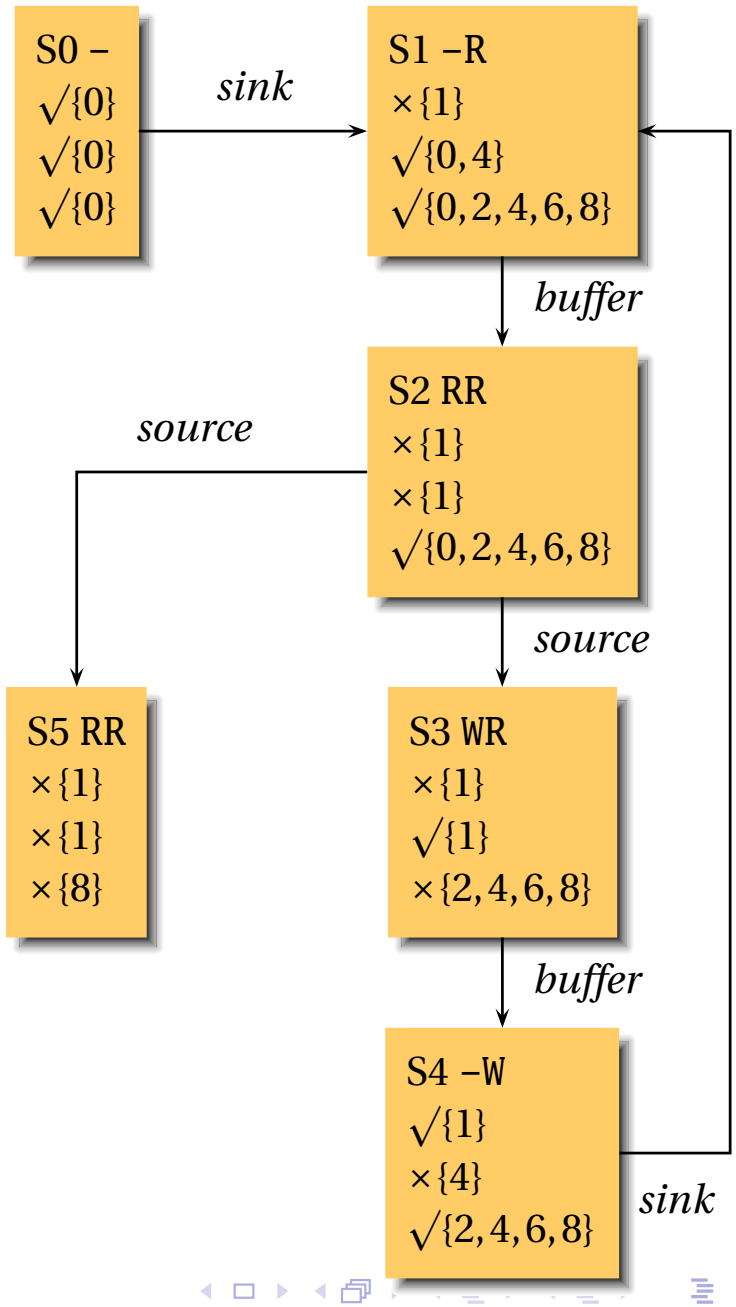

```
process                          sink
sink(int32 B) {                  0 PreRead 1
  for (;;) B;                    1 PostRead 1 tmp3
}                                2 goto 0

process                          buffer
buffer(int32 &B,                 0 PreRead 0
       int32 A) {                1 PostRead 0 tmp2
  for (;;) B = A;                2 tmp1 := tmp2
}                                3 Write 1 tmp1
                                 4 goto 0
process
source(int32 &A) {               source
  A = 17; A = 42;                0 tmp4 := 17
  A = 157; A = 8;                1 Write 0 tmp4
}                                2 tmp5 := 42
                                 3 Write 0 tmp5
network main() {                 4 tmp6 := 157
  sink();                        5 Write 0 tmp7
  buffer();                      6 tmp8 := 8
  source();                      7 Write 0 tmp8
}                                8 Exit
```

Code Generation


```
process                          sink
sink(int32 B) {                  0 PreRead 1
  for (;;) B;                    1 PostRead 1 tmp3
}                                2 goto 0

process                          buffer
buffer(int32 &B,                 0 PreRead 0
       int32 A) {                1 PostRead 0 tmp2
  for (;;) B = A;                2 tmp1 := tmp2
}                                3 Write 1 tmp1
                                 4 goto 0
process
source(int32 &A) {               source
  A = 17; A = 42;                0 tmp4 := 17
  A = 157; A = 8;                1 Write 0 tmp4
}                                2 tmp5 := 42
                                 3 Write 0 tmp5
network main() {                 4 tmp6 := 157
  sink();                        5 Write 0 tmp7
  buffer();                      6 tmp8 := 8
  source();                      7 Write 0 tmp8
}                                8 Exit
```



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
};
```

Three Additional Constructs

$stmt_1$ `par` $stmt_2$	Run $stmt_1$ and $stmt_2$ concurrently
`send` *var*	Communicate on channel *var*
`recv` *var*	
`next` *var*	
`try {`	Define the scope of an exception
⋮	
`throw` *exc*	Raise an exception
⋮	
`} catch(` *exc* `)` *stmt*	

Concurrency & *par*

[Tardieu & Edwards, Memocode 2006]

Par statements run concurrently and asynchronously

Terminate when all terminate

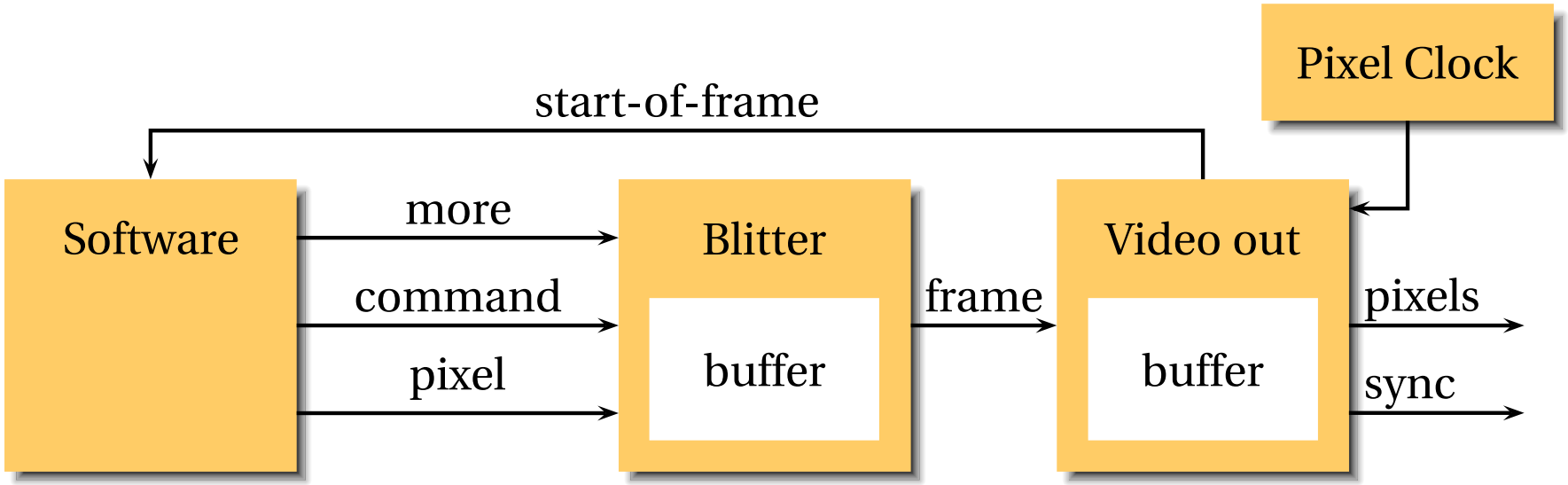
Each thread gets private copies of variables; no sharing

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
}
```

Robby Roto in SHIM

[Tardieu & Edwards, Memocode 2006]




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

```
for (;;) {
  while (next more) {
    recv command;
    Write to buffer
  }
  next frame = buffer;
}
```

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

Exceptions

Five functions that call each other and communicate through channel *A*

```
void main() {
  try {
    chan int A;
    f(A); par g(A);
  } catch (Done) {}
}
```

```
void f(chan int &A) throws Done {
  h(A); par j(A);
}
```

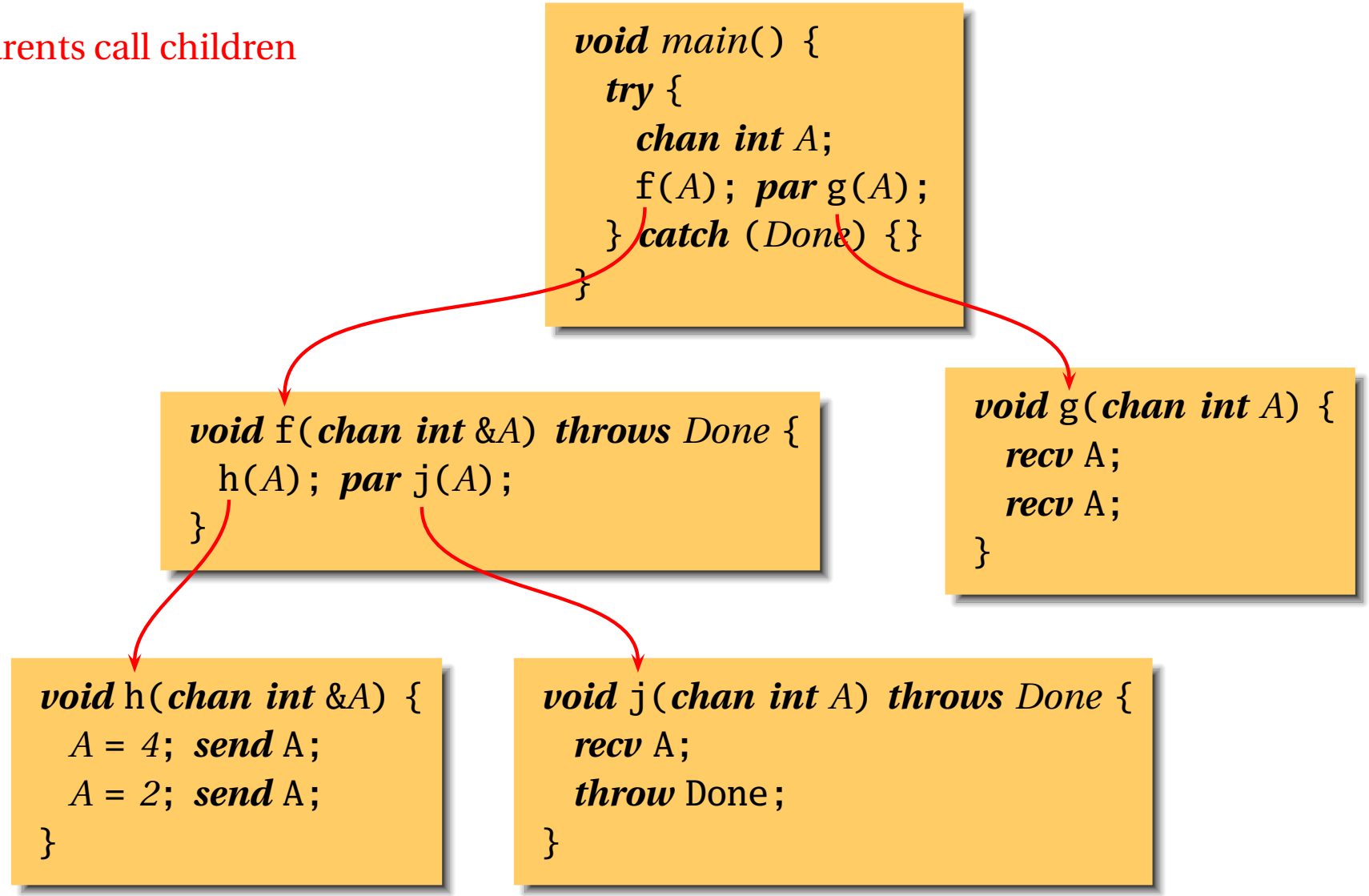
```
void g(chan int A) {
  recv A;
  recv A;
}
```

```
void h(chan int &A) {
  A = 4; send A;
  A = 2; send A;
}
```

```
void j(chan int A) throws Done {
  recv A;
  throw Done;
}
```

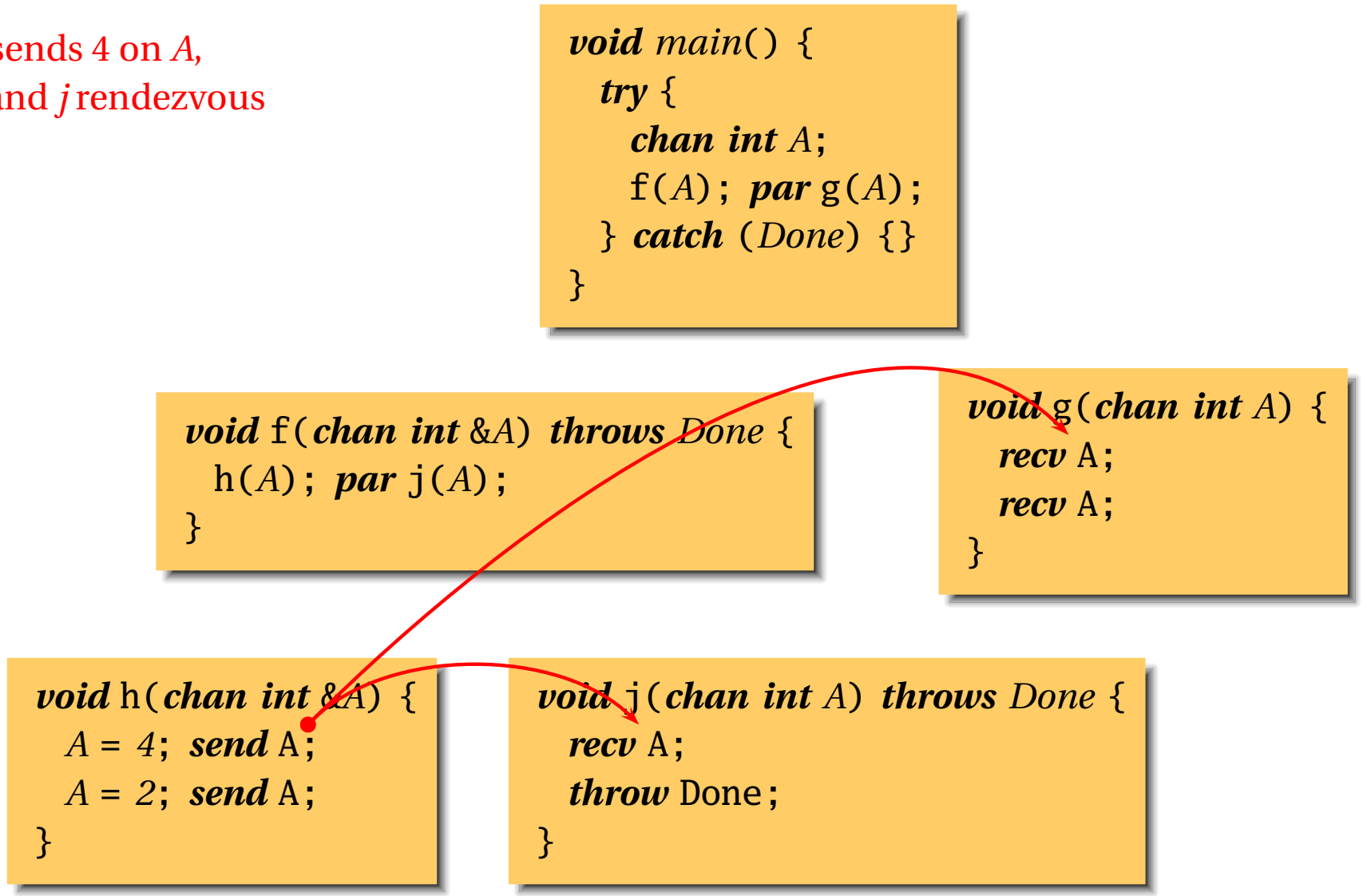
Exceptions

Parents call children



Exceptions

h sends 4 on *A*,
g and *j* rendezvous



Exceptions

j throws an exception. *g* and *h* poisoned by attempting communication

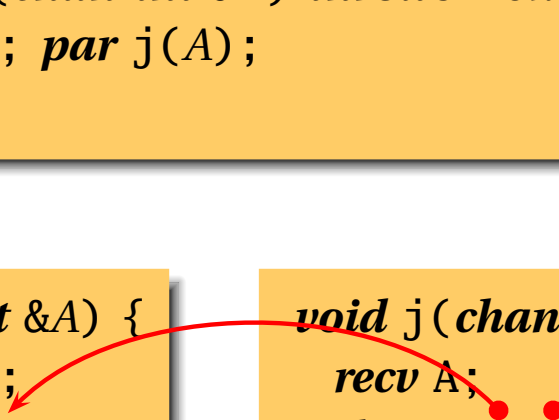
```
void main() {
  try {
    chan int A;
    f(A); par g(A);
  } catch (Done) {}
}
```

```
void f(chan int &A) throws Done {
  h(A); par j(A);
}
```

```
void g(chan int A) {
  recv A;
  recv A;
}
```

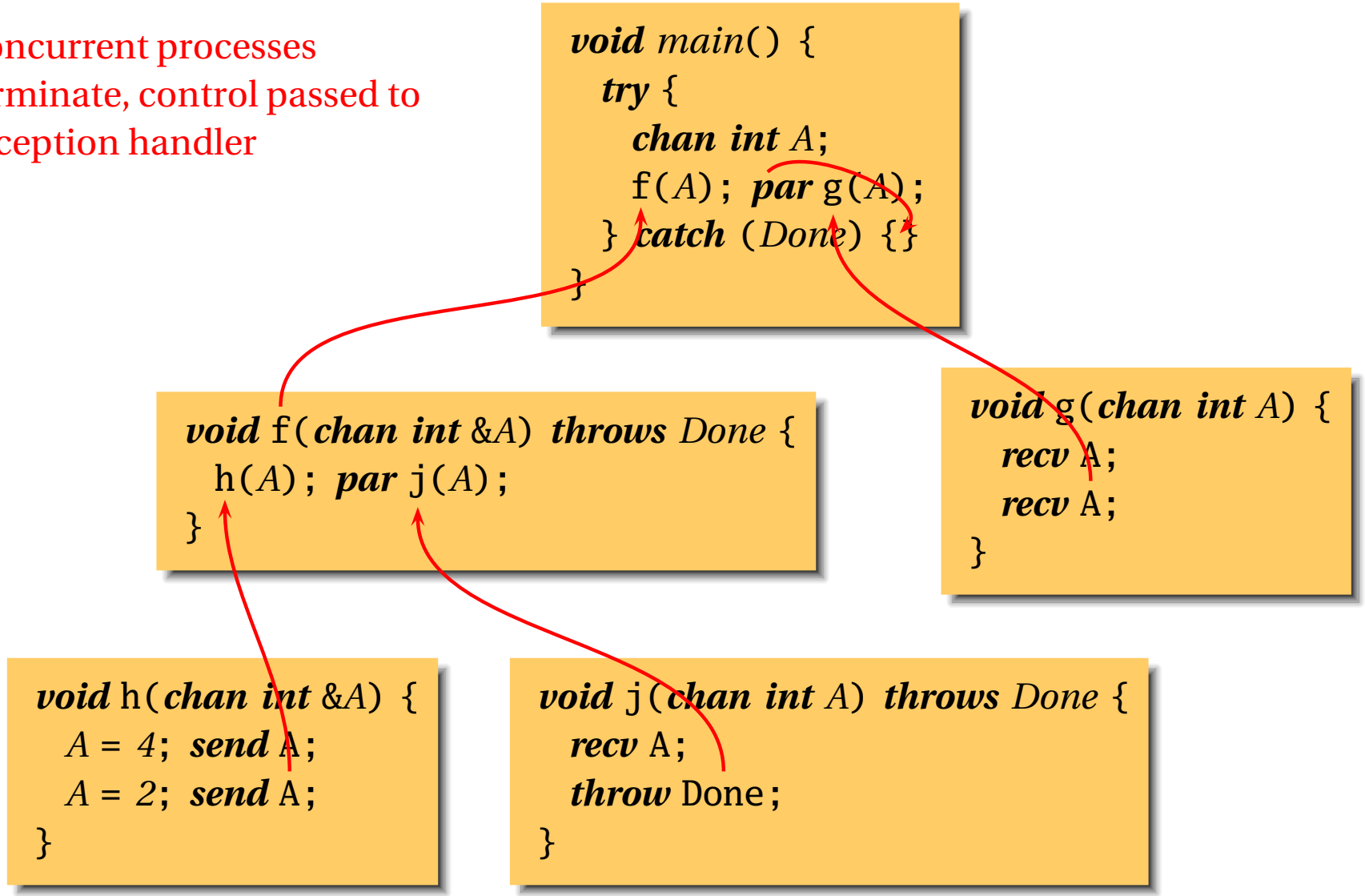
```
void h(chan int &A) {
  A = 4; send A;
  A = 2; send A;
}
```

```
void j(chan int A) throws Done {
  recv A;
  throw Done;
}
```



Exceptions

Concurrent processes terminate, control passed to exception handler

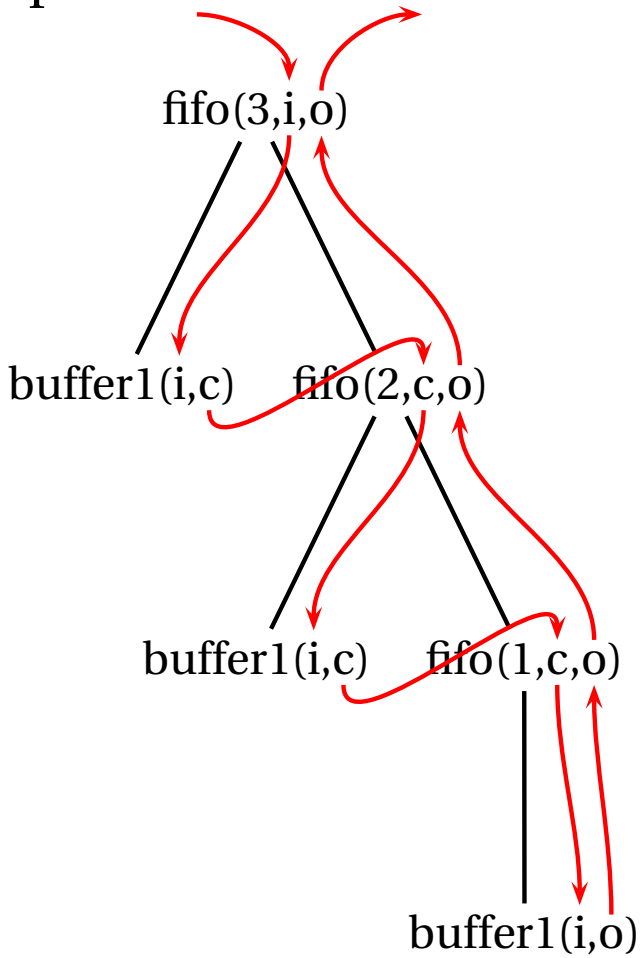


Bounded Recursion

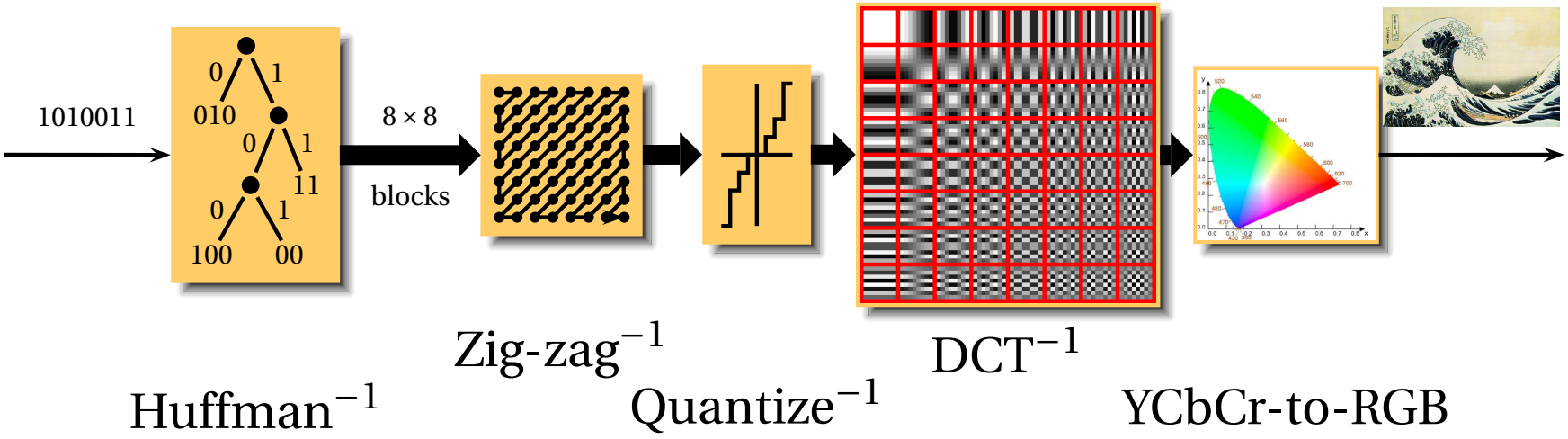
A bounded FIFO: compiler will analyze & expand

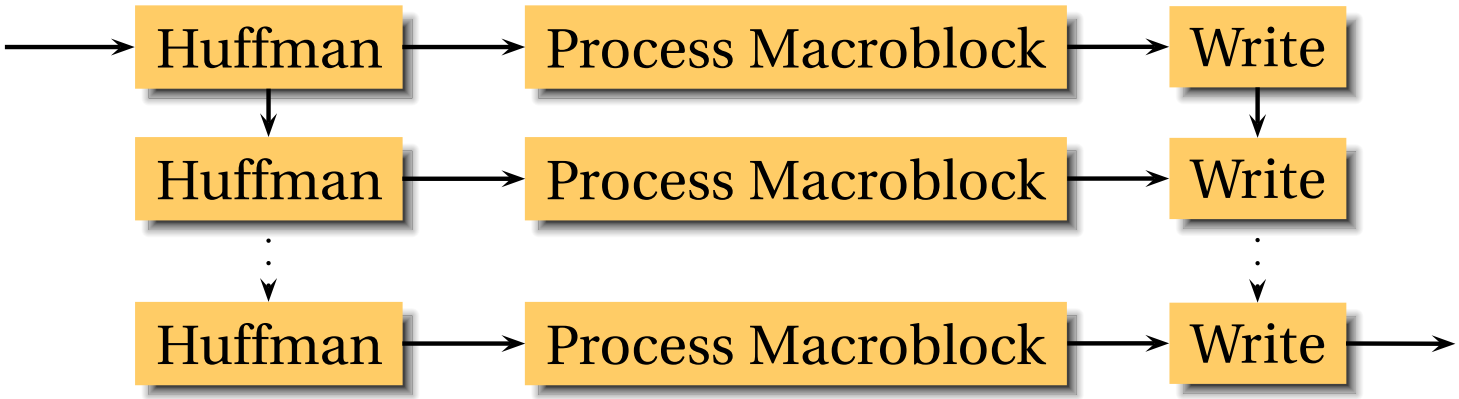
```
void buffer1(chan int in, chan int &out) {
  for (;;) next out = next in;
}

void fifo(int n, chan int in, chan int &out) {
  if (n == 1)
    buffer1(in, out);
  else {
    chan int channel;
      buffer1(in, channel);
    par
      fifo(n-1, channel, out);
  }
}
```



JPEG Decoding

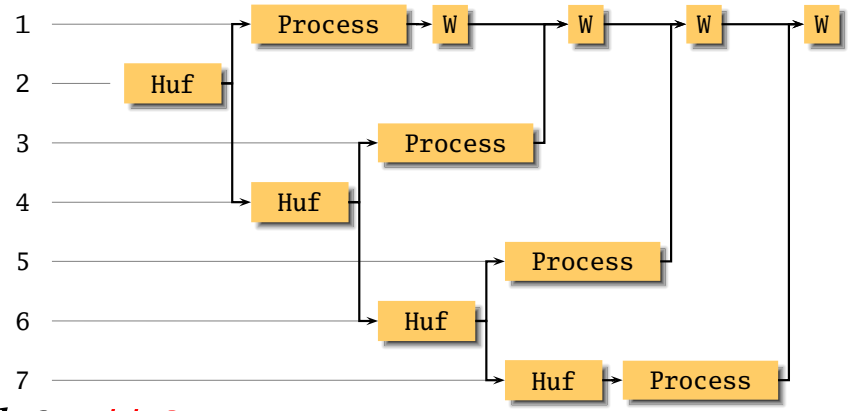




[Edwards, Vasudevan, and Tardieu, DATE 2008]

SHIM for the Seven-task Schedule


```
unpack(ustate, stripe1); // 2
{
  process(stripe1, pixels1); write(wstate, pixels1); // 1
  recv pixels2; write(wstate, pixels2);
  recv pixels3; write(wstate, pixels3);
  recv pixels4; write(wstate, pixels4);
} par {
  unpack(ustate, stripe2); // 4
  {
    process(stripe2, pixels2); send pixels2; // 3
  } par {
    unpack(ustate, stripe3); // 6
    {
      process(stripe3, pixels3); send pixels3; // 5
    } par {
      unpack(ustate, stripe4); // 7
      process(stripe4, pixels4); send pixels4;
} } }
```

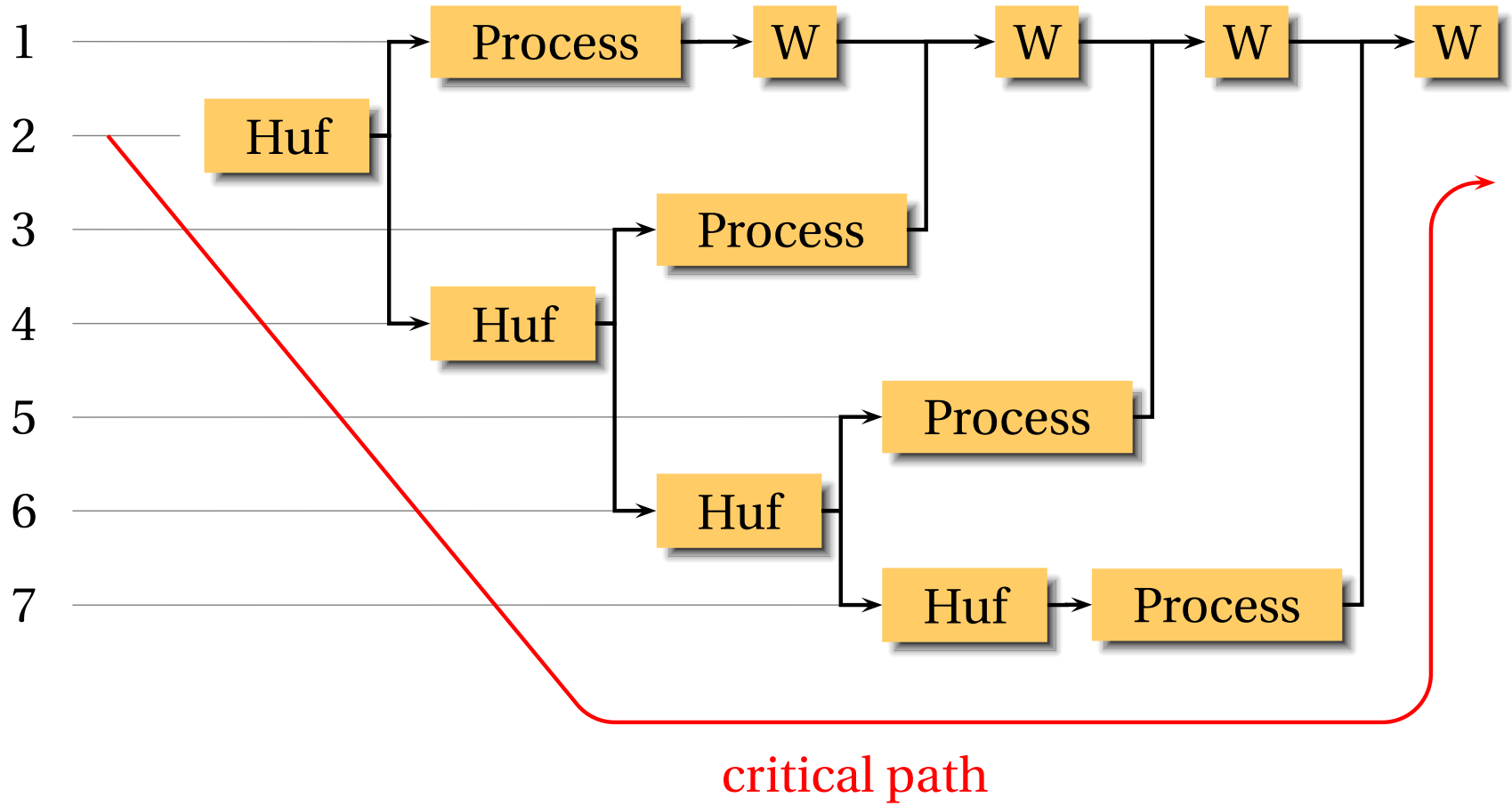


SHIM Enforces Dependencies


```
unpack(ustate, stripe1);
{
  process(stripe1, pixels1); write(wstate, pixels1);
  recv pixels2; write(wstate, pixels2);
  recv pixels3; write(wstate, pixels3);
  recv pixels4; write(wstate, pixels4);
} par {
  unpack(ustate, stripe2);
  {
    process(stripe2, pixels2); send pixels2;
  } par {
    unpack(ustate, stripe3);
    {
      process(stripe3, pixels3); send pixels3;
    } par {
      unpack(ustate, stripe4);
      process(stripe4, pixels4); send pixels4;
} } }
```

- Writer state local to one process
- Unpacker state can only be passed by reference once
- Trying to run *unpack* or *write* in parallel gives compiler error

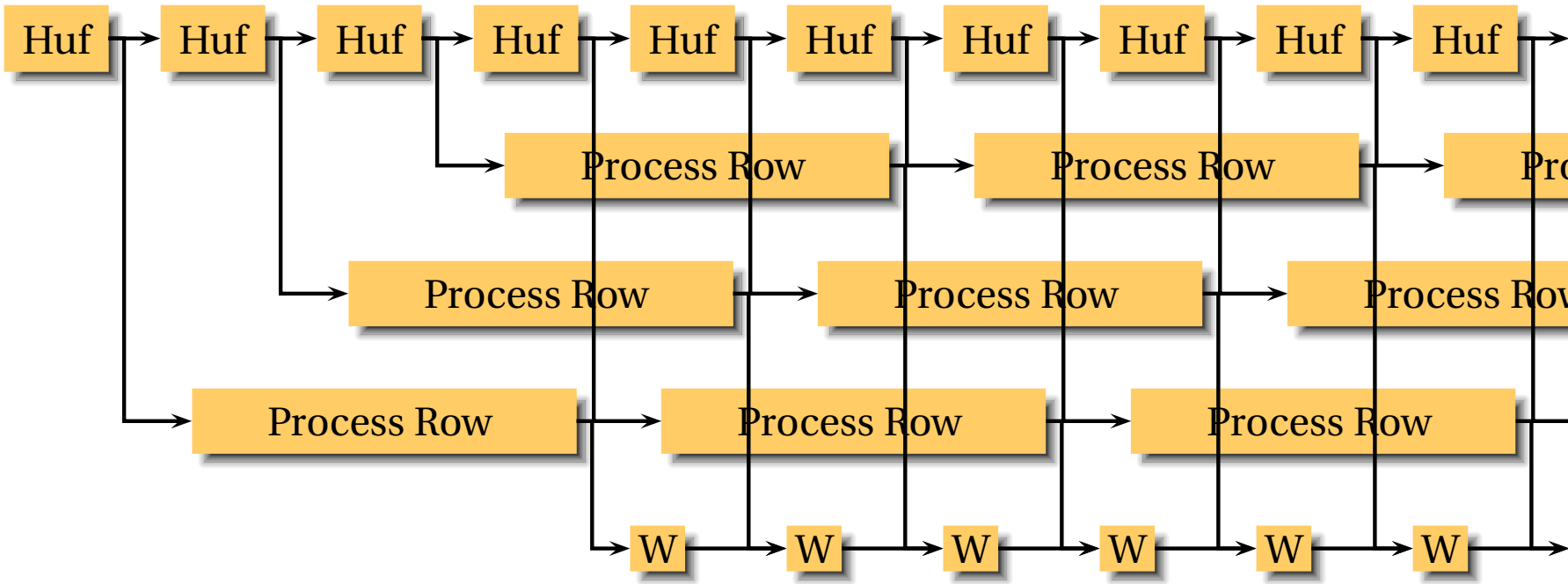
Oops



Only achieved a 1.8× speedup

Pipelined JPEG

[Edwards, Vasudevan, and Tardieu, DATE 2008]

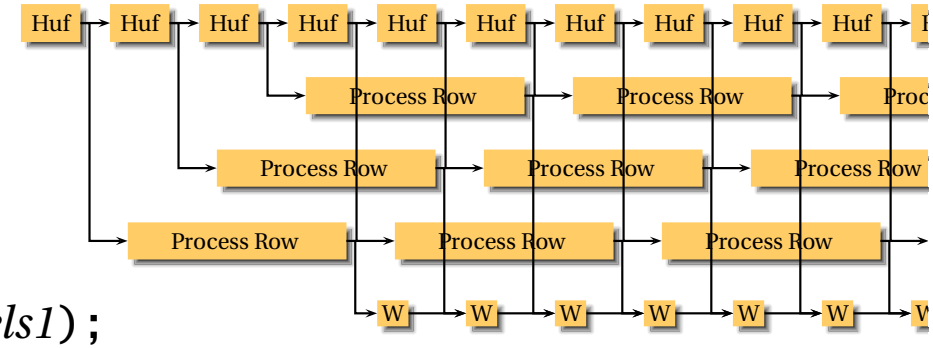


Process a row of blocks at a time (e.g., 64).

Reduce communication; accelerate start-up and termination.

SHIM for Pipelined JPEG


```
try {
  {
    for (;;) {
      unpack(ustate, row1); send row1; if (--rows == 0) break;
      unpack(ustate, row2); send row2; if (--rows == 0) break;
      unpack(ustate, row3); send row3; if (--rows == 0) break;
    } throw Done;
  } par
    process(row1, pixels1); par
    process(row2, pixels2); par
    process(row3, pixels3); par
  {
    for (;;) {
      recv pixels1; write(wstate, pixels1);
      recv pixels2; write(wstate, pixels2);
      recv pixels3; write(wstate, pixels3);
  } }
} catch (Done) {}
```



JPEG Results

[Edwards, Vasudevan, and Tardieu, DATE 2008]

Cores	Tasks	Time	Total	Total/Time	Speedup
1	1	25s	20s	0.8	1.0×(def)
1	1+3+1	24	24	1.0	1.04
2	1+3+1	13	24	1.8	1.9
3	1+3+1	11	24	2.2	2.3
4	1+3+1	8.7	25	2.9	2.9
4	1+1+1	16	24	1.5	1.6
4	1+2+1	9.3	25	2.7	2.7
4	1+3+1	8.7	25	2.9	2.9
4	1+4+1	8.2	25	3.05	3.05
4	1+5+1	8.6	25	2.9	2.9

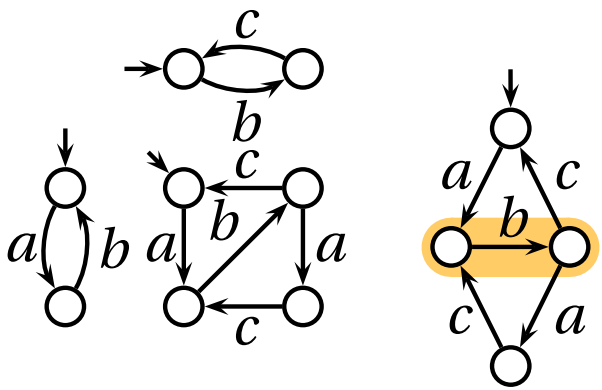
Compositional Deadlock Detection

[Shao, Vasudevan, and Edwards, Emsoft 2009]

```
void main()
{
  chan int a, b, c, d;
      for(;;) {
    recv a; b = a + 1; send b;
  } par for(;;) {
    recv b; c = b + 1; send c;
  } par for(;;) {
    recv c; d = c + 1; send d;
  } par for(;;) {
    recv d; a = d + 1; send a;
  }
}
```

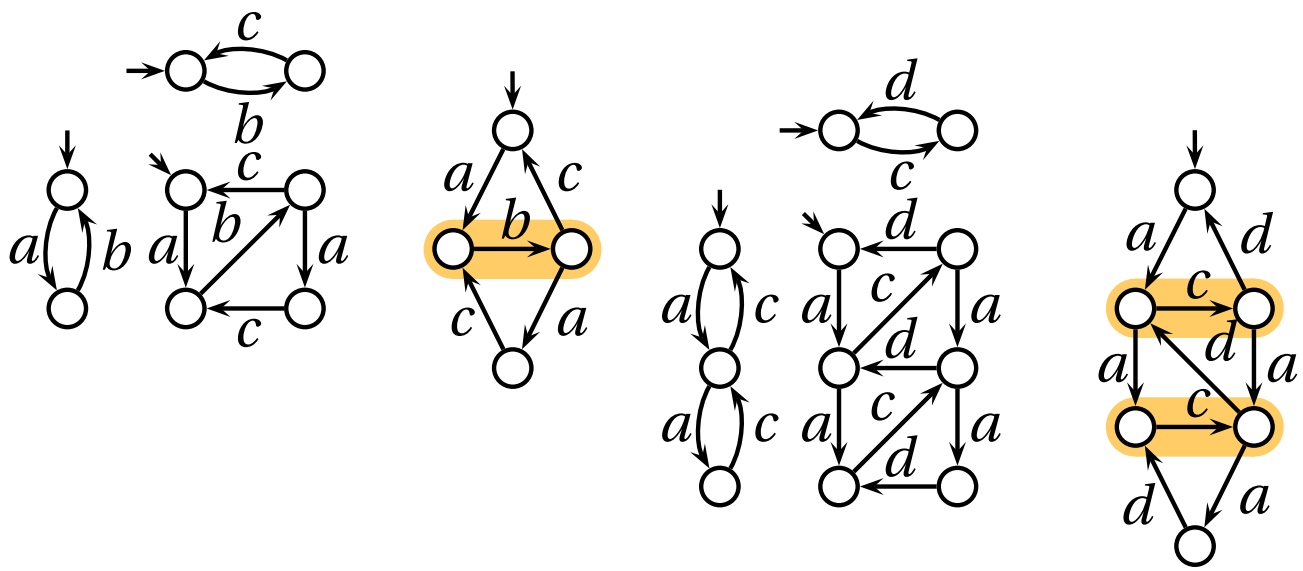
Compositional Deadlock Detection

[Shao, Vasudevan, and Edwards, Emsoft 2009]



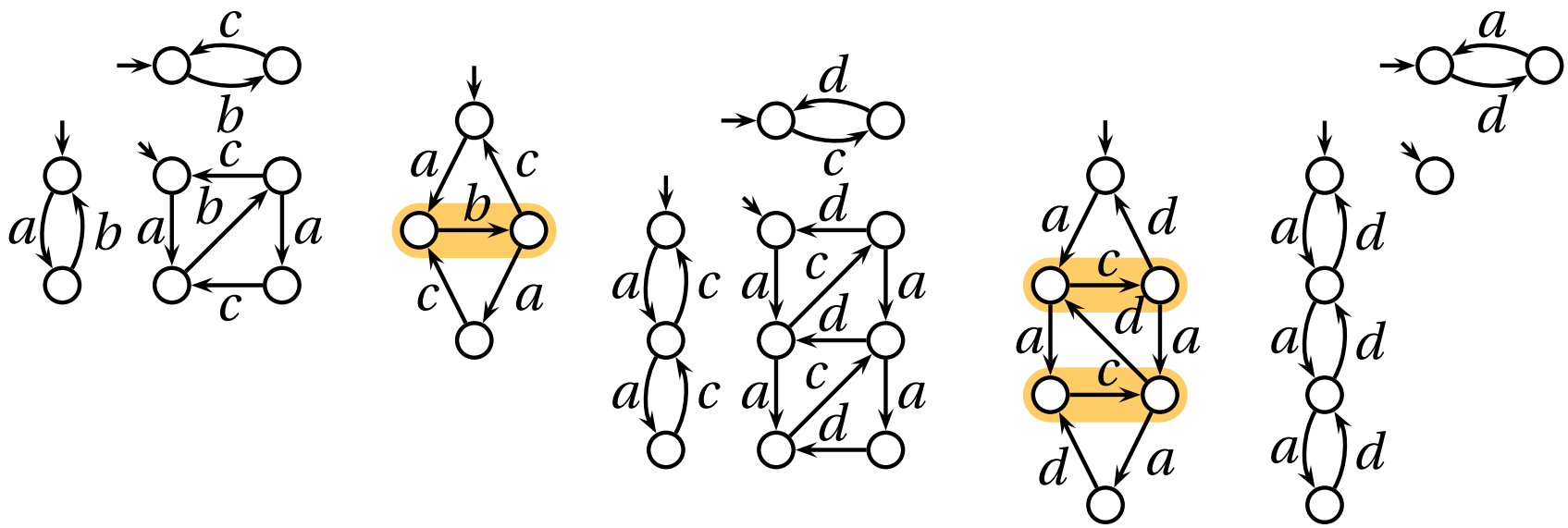
Compositional Deadlock Detection

[Shao, Vasudevan, and Edwards, Emsoft 2009]



Compositional Deadlock Detection

[Shao, Vasudevan, and Edwards, Emsoft 2009]

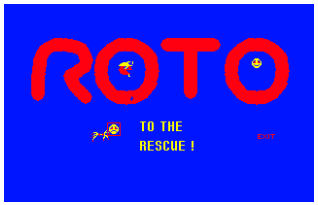


Conclusions

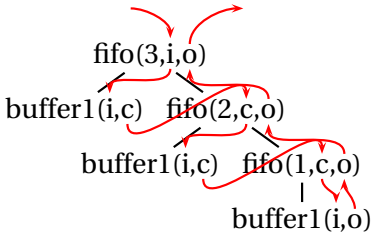
[Edwards, SEUS 2009]

SHIM

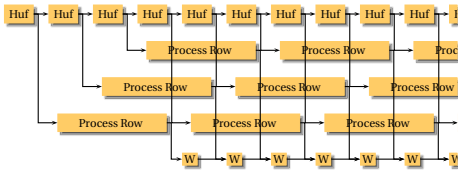
Scheduling-independent message passing



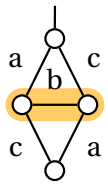
Designed for hardware/software systems



Imperative language with bounded recursion



Exploring schedules interesting, safe



Enables compositional deadlock detection