Definition

**shim**  
\'shim\  \textit{n}\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
The SHIM Wishlist

- **Concurrent**
  Hardware always concurrent

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

- **Only requires bounded resources**
  Hardware always bounded; simplifies verification

- **Formal semantics**
  Clarifies verification problem; want to avoid arguments

- **Scheduling-independent**
  Renders simulated behavior informative and definitive
  Solves some verification problems “by theorem”
  Verify functionality and performance separately
The SHIM Model

- Sequential processes
- Unbuffered one-to-many communication channels
- Exchange data tokens
- Dynamic topology with an easily-defined static subset
- Asynchronous process execution rates
- Blocking synchronous rendezvous communication
- Scheduling-independent: sequence of data through any channel independent of scheduling policy (the Kahn principle)
- “Kahn networks with rendezvous communication”
The SHIM Language

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 {
    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
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 rules avoid 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

Blocking: wait for all processes connected to \( a \)

```c
void f(chan int a) {
    // a is a copy of c
    a = 3; // change local copy
    recv a; // receive (wait for g)
    // a now 5
}

void g(chan int &b) {
    // b is an alias of c
    next b = 5; // sets c and send (wait for f)
    // b now 5
}

void main() {
    chan int c = 0;
    f(c); par g(c);
}
```
Synchronization, Deadlocks

Blocking communication makes for potential deadlock

\{ \textit{next a; next b; } \} \texttt{par} \{ \textit{next b; next a; } \} \ // \texttt{deadlocks}

Only threads responsible for a variable must synchronize

\{ \textit{next a; next b; } \} \texttt{par} \textit{next b; par} \textit{next a; } \ // \texttt{OK}

When a thread terminates, it is no longer responsible

\{ \textit{next a; next a; } \} \texttt{par} \textit{next a; } \ // \texttt{OK}

Philosophy: deadlocks easy to detect; races are too subtle

SHIM prefers deadlocks to races (always reproducible)
void main() {
    chan uint8 A, B, C;
    {{
        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 (;;)
            recv C;
    }
}
Recursion & Concurrency

A bounded FIFO: compiler analyzes & expands

```c
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);
    }
}
```

Verification Challenges in the SHIM Concurrent Language – p. 12/20
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() {
    chan int i = 0, j = 0;
    try {
        while (i < 5)
            next i = i + 1;
        throw T;
    } par {
        for (;;) {
            next j =
                next i + 1;
        }
    } par {
        for (;;) {
            recv 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”
SHIM Verification Challenges

- Can a particular program deadlock? General answer is data-dependent. Many systems exhibit regular patterns.
SHIM Verification Challenges

- *Can a particular program deadlock?*
  General answer is data-dependent
  Many systems exhibit regular patterns

- *Can a program achieve a particular performance?*
  Related to worst-case execution time analysis
  Complicated by communication behaviors
  Precise answer depends on particular implementation
SHIM Verification Challenges

• *Can a particular program deadlock?* General answer is data-dependent Many systems exhibit regular patterns

• *Can a program achieve a particular performance?* Related to worst-case execution time analysis Complicated by communication behaviors Precise answer depends on particular implementation

• *Does a translation faithfully implement SHIM semantics?* Pthreads implementation nondeterministic Many opportunities for inadvertant races
A Partial Evaluation Approach

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
  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
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
Abstract Simulation

{ // buf1
  for (; ;)
    next B = next A;
} par {
  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
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

Verification Challenges in the SHIM Concurrent Language – p. 17/20
Abstract Simulation

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

Verification Challenges in the SHIM Concurrent Language – p. 17/20
Abstract Simulation

\[
\begin{align*}
&\{ \quad \text{// buf1} \\
&\quad \text{for (;;)} \\
&\quad \text{next B = next A;} \\
&\} \quad \text{par} \quad \{ \quad \text{// buf2} \\
&\quad \text{for (;;)} \{ \\
&\quad \text{next C = next B;} \\
&\quad \text{next C = next B + 1;} \\
&\} \\
&\}
\end{align*}
\]
for (;;) next B = next A;

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
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

A clear
C waiting for writer
B waiting for reader

{1, 2} {3}
{1} {4}
buf2
{3} {6}
receive A
{2} {6}
buf1

{2} {5}
buf2

{3} {5}
buf1

{3} {5}
receive A
```

Verification Challenges in the SHIM Concurrent Language – p. 17/20
Abstract Simulation

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

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

Verification Challenges in the SHIM Concurrent Language – p. 17/20
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
B waiting for reader
C waiting for writer

Verification Challenges in the SHIM Concurrent Language – p. 17/20
Abstract Simulation

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

Verification Challenges in the SHIM Concurrent Language – p. 17/20
Abstract Simulation

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

Verification Challenges in the SHIM Concurrent Language – p. 17/20
Abstract Simulation

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

Verification Challenges in the SHIM Concurrent Language – p. 17/20
Abstract Simulation

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

Verification Challenges in the SHIM Concurrent Language – p. 17/20
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

Verification Challenges in the SHIM Concurrent Language – p. 17/20
Abstract Simulation

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

Verification Challenges in the SHIM Concurrent Language – p. 17/20
Abstract Simulation

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

Verification Challenges in the SHIM Concurrent Language – p. 17/20
Abstract Simulation

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

Verification Challenges in the SHIM Concurrent Language – p. 17/20
Abstract Simulation

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

```c
{ // buf1
  int x = 0;
  for (;;) {
    next B = x;
    next A = x + 1;
  }
} par {
  // buf2
  for (;;) {
    next C = next B + 1;
    next C = next B;
  }
}
```

Verification Challenges in the SHIM Concurrent Language – p. 17/20
Abstract Simulation

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

Verification Challenges in the SHIM Concurrent Language – p. 17/20
Abstract Simulation

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

Verification Challenges in the SHIM Concurrent Language – p. 17/20
### Experiments

<table>
<thead>
<tr>
<th>Example</th>
<th>Lines</th>
<th>Processes</th>
<th>States</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Partial</td>
<td>Full</td>
<td></td>
</tr>
<tr>
<td>Berkeley</td>
<td>36</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>Buffer2</td>
<td>25</td>
<td>4</td>
<td>10</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Buffer3</td>
<td>26</td>
<td>5</td>
<td>20</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>Buffer10</td>
<td>33</td>
<td>12</td>
<td>174</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>Esterel1</td>
<td>144</td>
<td>5</td>
<td>49</td>
<td>56</td>
<td></td>
</tr>
<tr>
<td>Esterel2</td>
<td>127</td>
<td>5</td>
<td>24</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>FIR5</td>
<td>78</td>
<td>19</td>
<td>229</td>
<td>79</td>
<td></td>
</tr>
<tr>
<td>FIR19</td>
<td>190</td>
<td>75</td>
<td>2819</td>
<td>372</td>
<td></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
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

• Richer data structures
  Shared arrays, Trees, etc.
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
- Richer data structures
  Shared arrays, Trees, etc.
- Convince world: scheduling-independent concurrency is good