SHIM: A Deterministic Model for Heterogeneous Embedded Systems

Stephen A. Edwards and Olivier Tardieu
Department of Computer Science,
Columbia University

www.cs.columbia.edu/~sedwards
{sedwards,tardieu}@cs.columbia.edu
Definition

**shim** \ˈʃim\  *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
Conclusions

SHIM is an effective model of computation for embedded hardware/software systems

Formal semantics guarantee determinism & boundedness

Easy to synthesize into hardware and software

Applicable to large, important class of systems, but not all

Embedded systems should be designed on the SHIM model of computation
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
HW/SW Interaction

Software

Blitter

Memory

Video

Interrupt

Blit

Pixels

Line

Blit

Pixels

Line

Blit

Pixels

Line

Interrupt
SHIM Wishlist

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

- *Delay-insensitive (Deterministic)*
  Want simulated behavior to reflect reality
  Verify functionality and performance separately

- *Only requires bounded resources*
  Hardware resources fundamentally bounded

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

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”
# SHIM vs. Other Models

<table>
<thead>
<tr>
<th>Feature</th>
<th>SHIM</th>
<th>CSP</th>
<th>Kahn</th>
<th>SDF</th>
<th>Haste</th>
<th>Sync</th>
<th>Petri</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deterministic</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Blocking Communication</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Bounded Buffers</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Multi-Rate</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Data-Dependent Rates</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Easy-To-Schedule</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Static Scheduling</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>
## Modeling in SHIM

<table>
<thead>
<tr>
<th>To model</th>
<th>introduce</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffers</td>
<td>Buffer processes</td>
</tr>
<tr>
<td>Interrupts</td>
<td>Polling and periodic communication</td>
</tr>
<tr>
<td>Synchrony</td>
<td>Clock signals</td>
</tr>
<tr>
<td>Synchronous dataflow</td>
<td>Buffers</td>
</tr>
<tr>
<td>Sensors</td>
<td>Source processes</td>
</tr>
<tr>
<td>Arbiters</td>
<td>A deterministic algorithm</td>
</tr>
</tbody>
</table>
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
Example Processes

process sender(uint32 &C) {
    for (int d = 0; d < 4; ++d) {
        for (int e = d; e > 0; --e) {
            C = 1, C = e;
        }
    }
    C = 0;
}

process sink(uint32 &D) {
    for (;;) { int v = D; }
}

process receiver(uint32 C, uint32 &D) {
    int b = 0;
    for (;;) {
        int a = 0;
        int r = 1;
        for (;;) {
            r = C;
            if (r == 0) break;
            int v = C;
            a = a + v;
        }
        ++b;
        D = b, D = a;
    }
}
while the player is alive do
Wait for start-of-frame
...game logic...
Write “false” to end-of-frame
Write to the blitter
...game logic...
Write “true” to end-of-frame

while 1 do
while not end-of-frame do
Read blit command
Write pixels to memory
Write frame

for each line do
Emit line timing signals
Write frame

while 1 do
Write start-of-frame
for each pixel do
Wait for pixel clock
Read pixel from memory
Send pixel to display
Read next frame
The Syntax of Tiny-SHIM

\[
e ::= L \quad \text{(literal)} \quad s ::= V = e \quad \text{(assignment)}
\]
\[
| \quad V \quad \text{(variable)} \quad | \quad \text{if} (e) s \text{ else } s \quad \text{(conditional)}
\]
\[
| \quad \text{op } e \quad \text{(unary op)} \quad | \quad \text{while} (e) s \quad \text{(loop)}
\]
\[
| \quad e \text{ op } e \quad \text{(binary op)} \quad | \quad s ; s \quad \text{(sequencing)}
\]
\[
| \quad (e) \quad \text{(paren)} \quad | \quad \text{read} (C, V) \quad \text{(blocking read)}
\]
\[
| \quad \text{write} (C, e) \quad \text{(blocking write)}
\]
\[
| \quad \{s\} \quad \text{(grouping)}
\]
Implementing Tiny-SHIM in Software

Each process becomes a C function with static variable that can resume itself.

Main scheduler takes runnable process from head of list and calls its function.

Processes mark themselves blocked, can schedule their communication partner.
C Translation Example

process p1 {
    output C;
    write(C, 42);
}

process p2 {
    input C;
    int v;
    read(C, v);
}
C Declarations

typedef struct process_struct {
    void (*process)(void);
    struct process_struct *next;
} process_t;

typedef struct {
    int value;
    process_t *waiting;
} channel_t;

channel_t C = { 0, 0 };  

int p1_state = 0;
int p2_state = 0;
void p1_function(void);
void p2_function(void);

process_t p1 = { p1_function, 0 };
process_t p2 = { p2_function, &p1 };
process_t *head_process = &p2;
int main()
{
    process_t *running_process;
    while (head_process) {
        running_process = head_process;
        head_process = running_process->next;
        (*(running_process->process))();
    }
    return 0;
}
The Writing Process

```c
void p1_function() {
    switch (p1_state) {
    case 1: goto L1;
    case 0: goto L0;
    }

L0:
    C.value = 42;
    if (C.waiting) {
        (C.waiting)->next = head_process;
        head_process = C.waiting;
    }
    C.waiting = &p1;
    p1_state = 1;
    return;
L1:
    ;
}
```

**Resume at current state**

**write(C, 42)**

**Schedule**

**reading process**

**Suspend**
The Reading Process

void p2_function() {
    static int v;
    switch (p2_state) {
        case 0: goto L0;
        case 1: goto L1;
    }

    L0:
        if (!C.waiting) {
            C.waiting = &p2;
            p2_state = 1;
            return;
        }
    L1:
        v = C.value;
        (C.waiting)->next = head_process;
        head_process = C.waiting;
        C.waiting = 0;
    }

    read(C, v); // Suspend
    process p2 {
        input C;
        int v;
        read(C, v);
        Schedule
    }
    writing process
    resume at current state
Write before Read

/* write(C, 42) */
C.value = 42;
if (C.waiting) {
    (C.waiting)->next = head_process;
    head_process = C.waiting;
}
C.waiting = &p1;
p1_state = 1;
return;
L1:

/* read(C, v) */
if (!C.waiting) {
    C.waiting = &p2;
p2_state = 1;
    return;
}
L1:
v = C.value;
(C.waiting)->next = head_process;
head_process = C.waiting;
C.waiting = 0;
Read before Write

/* write(C, 42) */
C.value = 42;
if (C.waiting) {
    (C.waiting)->next = head_process;
    head_process = C.waiting;
}
C.waiting = &p1;
p1_state = 1;
return;
L1:
/* read(C, v) */
if (!C.waiting) {
    C.waiting = &p2;
p2_state = 1;
    return;
}
L1:
v = C.value;
(C.waiting)->next = head_process;
head_process = C.waiting;
C.waiting = 0;
Translating Tiny-SHIM to Hardware

 CFG Node
 Assignment
 Decision
 Merge
 Cycle Boundary

 Control Fragment

 Datapath Fragment
Translation Patterns

\[ if \left( e \right) s_1 \ else \ s_2 \]

\[ while \left( e \right) s \]

\[ write \left( c, e \right) \]

\[ read \left( c, v \right) \]
Hardware Translation Example

d = 0;
while (1) {
    e = d;
    while (e > 0) {
        write(c, 1);
        write(c, e);
        e = e - 1;
    }
    write(c, 0);
    d = d + 1;
}

a = 0;
b = 0;
while (1) {
    r = 1;
    while (r) {
        read(c, r);
        if (r != 0) {
            read(c, v);
            a = a + v;
        }
    }
    b = b + 1;
}

__________
The SOS Semantics of Tiny-SHIM

\[ \sigma \quad \text{Process memory state} \quad p \quad \text{Process code} \]

\[ \langle \sigma, p \rangle \quad \text{Process } p \text{ in state } \sigma \quad \langle \sigma \rangle \quad \text{Terminated in state } \sigma \]

\[ \xrightarrow{a} \quad \text{Single-process rule} \quad \Rightarrow \quad \text{System rule} \]

\[ \mathcal{E}(\sigma, e) \quad \text{Value of } e \text{ in } \sigma \]

\[
\begin{align*}
\mathcal{E}(\sigma, e) &= n \\
\langle \sigma, v = e \rangle &\rightarrow \langle \sigma[v \leftarrow n] \rangle
\end{align*}
\]

(assign)

\[
\begin{align*}
\mathcal{E}(\sigma, e) &\neq 0 \\
\langle \sigma, \text{if } (e) \ p \ \text{else } q \rangle &\rightarrow \langle \sigma, p \rangle
\end{align*}
\]

(if-true)

\[
\begin{align*}
\mathcal{E}(\sigma, e) &= 0 \\
\langle \sigma, \text{if } (e) \ p \ \text{else } q \rangle &\rightarrow \langle \sigma, q \rangle
\end{align*}
\]

(if-false)
Semantics of Looping & Sequencing

\[ \mathcal{E}(\sigma, e) \neq 0 \]
\[ \langle \sigma, \text{while} (e) \ p \rangle \rightarrow \langle \sigma, p ; \text{while} (e) \ p \rangle \]  
(while-true)

\[ \mathcal{E}(\sigma, e) = 0 \]
\[ \langle \sigma, \text{while} (e) \ p \rangle \rightarrow \langle \sigma \rangle \]  
(while-false)

\[ \langle \sigma, p \rangle \xrightarrow{a} \langle \sigma', p' \rangle \]
\[ \langle \sigma, p ; q \rangle \xrightarrow{a} \langle \sigma', p' ; q \rangle \]  
(seq)

\[ \langle \sigma, p \rangle \xrightarrow{a} \langle \sigma' \rangle \]
\[ \langle \sigma, p ; q \rangle \xrightarrow{a} \langle \sigma', q \rangle \]  
(seq-term)
Communication and Concurrency

\[
\langle \sigma, \text{read}(c, v) \rangle \xrightarrow{c \text{ get } n} \langle \sigma[v \leftarrow n] \rangle \quad \text{(read)}
\]

\[
E(\sigma, e) = n \\
\langle \sigma, \text{write}(c, e) \rangle \xrightarrow{c \text{ put } n} \langle \sigma \rangle \quad \text{(write)}
\]

\[
\langle \sigma, p \rangle \rightarrow s \\
\{\langle \sigma, p \rangle\} \cup S \Rightarrow \{s\} \cup S \quad \text{(step)}
\]

\[
\langle \sigma, p \rangle \xrightarrow{c \text{ put } n} s \\
\langle \sigma', p' \rangle \xrightarrow{c \text{ get } n} s' \quad \text{(sync)}
\]

\[
\{\langle \sigma, p \rangle, \langle \sigma', p' \rangle\} \cup S \Rightarrow \{s, s'\} \cup S
\]
Summary

- SHIM: A delay-insensitive (deterministic) model of computation that supports synchrony and asynchrony
- Tiny-SHIM: A little language that embodies the model
- A procedure for translating Tiny-SHIM into software
- A procedure for translating Tiny-SHIM into hardware
- Operational semantics of Tiny-SHIM
Ongoing Work

- Complete hardware/software design language
  One novel aspect: exact integer arithmetic. Ranges inferred at compile time.
- Relaxation of block-on-single-channel rule
  There is a way to do this without breaking determinism.
- Static analysis of deadlock
  Decidable: our language is not Turing-complete
- Translation optimization for hardware and software