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 \'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
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)
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</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>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</td>
</tr>
<tr>
<td>Synchrony</td>
<td>communication</td>
</tr>
<tr>
<td>Synchronous dataflow</td>
<td>Clock signals</td>
</tr>
<tr>
<td>Sensors</td>
<td>Buffers</td>
</tr>
<tr>
<td>Arbiters</td>
<td>Source processes</td>
</tr>
<tr>
<td></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
The Syntax of Tiny-SHIM

\[
e ::= L \quad \text{(literal)} \quad s ::= V = e \quad \text{(assignment)}
\]

\[
| V \quad \text{(variable)} \quad | \text{if } (e) \text{ s else s} \quad \text{(conditional)}
\]

\[
| op \ e \quad \text{(unary op)} \quad | \text{while } (e) \ s \quad \text{(loop)}
\]

\[
| e \ op \ e \quad \text{(binary op)} \quad | s \ ; \ s \quad \text{(sequencing)}
\]

\[
| (e) \quad \text{(paren)} \quad | \text{read}(C, V) \quad \text{(blocking read)}
\]

\[
| \{s\} \quad | \text{write}(C, e) \quad \text{(blocking write)}
\]

\[
| \quad | \quad \text{(grouping)}
\]
Example Processes

Local variables: d, e

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

Local variables: a, b, r, v

```c
int a = 0;
int b = 0;
while (1) {
    int r = 1;
    while (r) {
        read(c, r);
        if (r != 0) {
            read(c, v);
            a = a + v;
        }
    }
    b = b + 1;
}
```
Behavior of the Processes

```
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;
}
```
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
    Read pixel from memory
    Send pixel to display
  Read next 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

Robby Roto in SHIM: Block Diagram

Software

Blit

Video out

start-of-frame

dead-of-frame

command

pixels

frame

pixels

sync

Pixel Clock
Translating Tiny-SHIM to 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 scheduler their communication partner.
C Translation Example

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

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

```c
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;
```

Linked list of runnable processes
Function of process
Next runnable process

Channel datatype
Value being transferred
Blocked process, if any

Definition of channel C
State of each process
Process functions
(forward declarations)

Linked List
of runnable
processes
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

```c
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;
}
```
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

Assignment

Decision

Merge

Cycle Boundary

CFG Node

Control Fragment

Datapath Fragment
Translation Patterns

\[ \text{if}(\ e\ ) \: s_1 \: \text{else} \: s_2 \]

\[ \text{while}(\ e\ ) \: s \]

\[ \text{write}(\ c, \ e ) \]

\[ \text{read}(\ c, \ v ) \]
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

\( \sigma \) Process memory state \( p \) Process code

\( \langle \sigma, p \rangle \) Process \( p \) in state \( \sigma \) \( \langle \sigma \rangle \) Terminated in state \( \sigma \)

\( a \) Single-process rule \( \Rightarrow \) System rule

\( E(\sigma, e) \) Value of \( e \) in \( \sigma \)

\[
E(\sigma, e) = n
\]

\[
\langle \sigma, v = e \rangle \rightarrow \langle \sigma[v \leftarrow n] \rangle
\]

(assign)

\[
E(\sigma, e) \neq 0
\]

\[
\langle \sigma, if \ (e) \ p \ else \ q \rangle \rightarrow \langle \sigma, p \rangle
\]

(if-true)

\[
E(\sigma, e) = 0
\]

\[
\langle \sigma, if \ (e) \ p \ else \ q \rangle \rightarrow \langle \sigma, q \rangle
\]

(if-false)
Semantics of Looping & Sequencing

\[
\begin{align*}
E(\sigma, e) \neq 0 & \quad \Rightarrow \quad \langle \sigma, \text{while (e) } p \rangle \rightarrow \langle \sigma, p ; \text{while (e) } p \rangle \\
E(\sigma, e) = 0 & \quad \Rightarrow \quad \langle \sigma, \text{while (e) } p \rangle \rightarrow \langle \sigma \rangle \\
\langle \sigma, p \rangle \xrightarrow{a} \langle \sigma', p' \rangle & \quad \Rightarrow \quad \langle \sigma, p ; q \rangle \xrightarrow{a} \langle \sigma', p ; q \rangle \\
\langle \sigma, p \rangle \xrightarrow{a} \langle \sigma' \rangle & \quad \Rightarrow \quad \langle \sigma, p ; q \rangle \xrightarrow{a} \langle \sigma', q \rangle
\end{align*}
\]

(while-true)

(while-false)

(seq)

(seq-term)
Communication and Concurrency

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

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

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

\[
\langle \sigma, p \rangle \xrightarrow{c \text{ put } n} s \quad \langle \sigma', p' \rangle \xrightarrow{c \text{ get } n} s' \\
\{ \langle \sigma, p \rangle, \langle \sigma', p' \rangle \} \cup S \Rightarrow \{ s, s' \} \cup S 
\]

(\text{sync})
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
- Formal operational semantics of Tiny-SHIM
Ongoing Work

- Relaxation of block-on-single-channel rule
- Complete hardware/software design language
- Static analysis of deadlock
- Translation optimization for hardware and software