SHIM: A Deterministic Model for Heterogeneous Embedded Systems

Stephen A. Edwards, *Member, IEEE,* and Olivier Tardieu

***Abstract*— Typical embedded hardware/software systems are implemented using a combination of C and an HDL such as Verilog. While each is well-behaved in isolation, combining the two gives a nondeterministic model of computation whose ultimate behavior must be validated through expensive (cycle-accurate) simulation.**

We propose an alternative for describing such systems. Our SHIM (software/hardware integration medium) model, effectively Kahn networks with rendezvous communication, provides deterministic concurrency. We present the Tiny-SHIM language for such systems and its semantics, demonstrate how to implement it in hardware and software, and discuss how it can be used to model a real-world system.

By providing a powerful, deterministic formalism for expressing systems, designing systems and verifying their correctness will become easier.

***Index Terms*— Hardware/software codesign, Deterministic model of computation, Software synthesis, Hardware synthesis**

I. Introduction

UNLIKE single-threaded software programs or synchronous digital logic circuits, real-world embedded systems contain many computational styles. Most are amalgams of hardware and software; the hardware is often implemented as one or more islands of synchronous logic, while the software may be single-threaded, concurrent, parallel, distributed, or event-driven.

We propose the SHIM (Software/Hardware Integration Medium) model, a concurrent, asynchronous, deterministic model for specifying, validating, and synthesizing such heterogeneous embedded systems, and discuss its simulation and synthesis. The need for concurrency is clear: at the minimum, hardware peripherals operate in parallel with software.

The need for an asynchronous model is more subtle: although embedded hardware-software systems are typically implemented using synchronous digital logic, software timing is difficult to predict. At the lowest level, the number of cycles required to execute each machine instruction on a modern processor is unpredictable because of complex interactions among instructions in the pipeline, the cache, superscalar instruction scheduling, and branch predictors. While such behavior can be modeled, it is costly to simulate. Similarly, at a higher level, the behavior of complex algorithms can be very difficult to predict. An asynchronous model of software allows such details to be safely ignored.

The third characteristic of SHIM—determinism—is more controversial. While nondeterminism has its place in models of unpredictable systems (e.g., lossy communication systems such as the Internet), we believe that it is wrong for specification languages because it makes the already-very-difficult question of functional verification that much harder.

Systems are invariably validated using simulation. Although simulation provides advantages such as scaling, its Achilles' heel is its need of appropriate stimulus. While the simulation of deterministic models suffer from this problem, nondeterministic models are worse because not only do they require the right stimulus, but since the simulator makes nondeterministic choices, even the right stimulus may not flush out bugs.

Nondeterministic models reduce the assurance a simulator provides from "the system *will* do that given this stimulus" to "the system *could* do that given this stimulus." Providing such weak assurance seems unacceptable. We believe it is no accident that the two most widely-used computational models—single-threaded software and synchronous digital logic—are deterministic.

Determinism also has advantages for formal verification. By reducing the number of possible behaviors the system can exhibit, determinism reduces the computational burden. For example, performing model checking on nondeterministic concurrent models is possible, but such algorithms devote substantial energy to dealing with nondeterminism.

Determinism also greatly simplifies debugging because it renders bugs reproducible. Provided an identical stimulus can be applied (which our model makes easy—the timing of stimulus does not need to be considered), the system behavior is guaranteed to be the same.

The environment of an embedded system is often nondeterministic, raising the question of whether determinism in the model is so important. We believe it is: the number of behaviors that need to be considered grows as the *product* of the number of behaviors of the environment and the number of behaviors of the system since the two run concurrently. When the system is deterministic, this number reduces to the number of behaviors of the environment.

In this paper we argue for the utility of SHIM. We describe the model, give a proof of its determinism (Section II), present a small language ("Tiny-SHIM"), and give its formal semantics (Section III). This enables us to discuss modeling a real-world hardware/software system in Section IV and some specific constructs in Section V.

We show how to compile Tiny-SHIM to software in Section VI and present a hardware translation in Section VII. Finally, we compare our model with others in Section VIII.

Edwards and Tardieu are with Columbia University

II. The SHIM Model

A system in the SHIM model consists of concurrently-running sequential processes that communicate exclusively with rendezvous through fixed, point-to-point communication channels. Thus, SHIM is a restriction of Kahn's networks [1] that uses a style of communication inspired by Hoare's CSP [2].

The processes in SHIM can be described in a classical imperative way (i.e., think of them as C functions or Java methods) that do not communicate through shared variables. All processes execute concurrently and their relative execution speeds are undefined, i.e., they execute asynchronously.

Inter-process communication is synchronous in the sense that both sending and receiving processes must agree when data is to be transferred. In general, either the sender or receiver may try to communicate first and will wait for the other. The topology of the system—the number and type of processes and communication channels—is fixed, and each communication channel connects a single sending process with a single receiving process. The communication structure of a system is therefore a directed graph whose nodes are processes and whose arcs are channels The graph may contain cycles.

Theorem 1: The sequence of symbols transmitted over each channel is deterministic.

Proof Follows from SHIM systems being a restriction of Kahn networks. First, interpret the system as a Kahn network (i.e., treat the communication channels as unbounded buffers and make the write operations non-blocking). Next, for each channel in the system, introduce a second "acknowledge" channel going in the opposite direction. After each receive operation, send on the acknowledge channel. Similarly, after each send operation, add a receive on the acknowledge channel. This receive forces the send to be blocking, just as in our model. Under this transformation, the processes compute continuous functions of their inputs and hence are deterministic since this augmented system fits Kahn's model. □

Corollary 1: The sequence of states visited by each process is deterministic.

Proof The states are determined by the structure of the machine and the data values transmitted on each channel, both of which are deterministic. □

A. Rationale

In the introduction, we discussed our rationale for wanting determinism: it greatly simplifies system validation whether with simulation or formal techniques. Thus, we felt that nondeterministic models such as CSP [2] or Petri nets [3] were unsatisfactory.

We rejected Kahn's unbounded buffers because they make the model Turing-complete even for simple processes. Buck [4] showed that simple multiplexer-like processes together with unbounded communication was enough to build a Turing machine.

The communication in our model is finite and does not introduce Turing-completeness. Specifically, our model is finite-state provided each process is finite-state. The advantages of this are legion: scheduling is much easier in our model because the synchronous communication restricts the number of choices. By definition, our systems can always be executed with finite memory; this question is undecidable for Kahn networks. Compare our simple scheduler, presented in Section VI, to the clever but costly one for Kahn networks by Parks [5], which dynamically detects buffer-overflow deadlock and increases buffer size in response.

$$
\begin{aligned}
e &::= L \mid V \mid op\ e \mid e\ op\ e \mid \texttt{(}\ e\ \texttt{)} \\
s &::= V = e \mid \texttt{if (}\ e\ \texttt{)}\ s\ \texttt{else}\ s \mid \texttt{while (}\ e\ \texttt{)}\ s \mid s\ \texttt{;}\ s \\
&\quad \mid \texttt{read(}\ C, V\ \texttt{)} \mid \texttt{write(}\ C, e\ \texttt{)} \mid \{\ \mathrm{s}\ \}
\end{aligned}
$$

Fig. 1. The syntax of the Tiny-SHIM language. Expressions and statements are classical except for the blocking `read` and `write` operations, which communicate values through channels. L is a literal, V represents a variable name, C a channel name, *op* represents the usual collection of operators (+, −, etc.), and braces indicate grouping.

Buffered communication can be implemented in SHIM by chaining multiple single-place-buffer processes. Such buffers are bounded, but adding more processes to increase a buffer's size is straightforward.

We could have chosen bounded buffers instead of rendezvous, but it would have complicated the model and probably necessitated an optimization step to simplify buffer management. There are already myriad ways to implement rendezvous communication and many opportunities for optimization.

We rejected the synchronous broadcast communication typical of register-transfer-level hardware languages such as VHDL, because we feel it is more error-prone. From observing students using this model, the most common mistake is a mismatch between when a signal is sent and when it is expected. The simulator cannot warn about such a situation because it is semantically valid, producing a difficult-to-diagnose failure.

SHIM does not preclude synchronous broadcast-style communication, but it must be requested explicitly, i.e., with processes triggered by a periodic clock that always receives every input. Section V presents a way to implement synchronous processes in SHIM.

III. Tiny-SHIM and its Semantics

Figure 1 shows the syntax for Tiny-SHIM, a language embodying our model. Each process is a statement (or group thereof) with its own set of variables, and each channel is read and written by exactly one process, although each such process may contain, for example, multiple read operations for a specific channel.

Tiny-SHIM is a simple language with no syntactic sugar. Meant as an easy-to-understand and analyze intermediate language, we plan to create the larger SHIM language that will include many more constructs. This will be dismantled into Tiny-SHIM.

We express the semantics of Tiny-SHIM in Plotkin's [6] structural operational style. The state of a process is represented as pair of the form $\langle \sigma, p \rangle$, where σ represents the state of the local store for each process, i.e., a mapping from a process's variables to values, and p is the statement the process

has become; or of the form $\langle\sigma\rangle$, which represents the process terminated in state σ.

The state of a system is a multiset of such process states: an unordered list of potentially repeated process states, since several processes may be in identical states.

Most rules (the "$\rightarrow$" rules) refer to the operation of a single process, which operates independently except when it communicates. The last two rules describe the operation of the system as a whole (the "$\Rightarrow$" rules) and either allow a single process or a pair of communicating processes to advance.

The rule for assignment statements is simplest. We use a helper function $\mathscr{E}$ that maps a store and expression to the value of the expression. The rule transforms a process consisting of an assignment to a variable v to a terminated process with the value of variable v replaced with the value of the expression. Expression evaluation is therefore side-effect free.

$$\frac{\mathscr{E}(\sigma,e)=n}{\langle\sigma, v = e\rangle \rightarrow \langle\sigma[v \leftarrow n]\rangle} \quad \text{(assign)}$$

The two rules for *if* statements are nearly as simple. Depending on whether the predicates evaluates to a non-zero value, either the *then* or *else* clause is scheduled to run.

$$\frac{\mathscr{E}(\sigma,e)\neq 0}{\langle\sigma,\texttt{if (}e\texttt{) }p\texttt{ else }q\rangle \rightarrow \langle\sigma,p\rangle} \quad \text{(if-true)}$$

$$\frac{\mathscr{E}(\sigma,e)= 0}{\langle\sigma,\texttt{if (}e\texttt{) }p\texttt{ else }q\rangle \rightarrow \langle\sigma,q\rangle} \quad \text{(if-false)}$$

The two rules for *while* use the residual style. The first unrolls the body of the *while* statement once if the predicate expression is true; the second terminates if the predicate is false.

$$\frac{\mathscr{E}(\sigma,e)\neq 0}{\langle\sigma,\texttt{while (}e\texttt{) }p\rangle \rightarrow \langle\sigma,p\texttt{ ; while (}e\texttt{) }p\rangle} \quad \text{(while-true)}$$

$$\frac{\mathscr{E}(\sigma,e)= 0}{\langle\sigma,\texttt{while (}e\texttt{) }p\rangle \rightarrow \langle\sigma\rangle} \quad \text{(while-false)}$$

The rules for *read* and *write* appear to be able to always execute, but the (sync) rule below only allows processes that contain them to execute in conjunction with each other.

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

$$\frac{\mathscr{E}(\sigma,e)=n}{\langle\sigma,\texttt{write(}c,e\texttt{)}\rangle \xrightarrow{c\ \text{put}\ n} \langle\sigma\rangle} \quad \text{(write)}$$

Sequencing requires two rules: one for when the first statement remains active, the other for when the first statement terminates. Here, the statement being executed may or may not require communication with another process depending on whether it is a *read* instruction ($a = c$ get n), a *write* instruction ($a = c$ put n), or another instruction (no transition label).

$$\frac{\langle\sigma,p\rangle \xrightarrow{a} \langle\sigma',p'\rangle}{\langle\sigma,p\texttt{ ; }q\rangle \xrightarrow{a} \langle\sigma',p'\texttt{ ; }q\rangle} \quad \text{(seq)}$$

$$\frac{\langle\sigma,p\rangle \xrightarrow{a} \langle\sigma'\rangle}{\langle\sigma,p\texttt{ ; }q\rangle \xrightarrow{a} \langle\sigma',q\rangle} \quad \text{(seq-term)}$$

The following rule expresses the fact that if a process can advance using one of the non-communicating rules, it can do so voluntarily without affecting any other processes. The $\uplus$ notation denotes the union of multisets.

$$\frac{\langle\sigma,p\rangle \rightarrow s}{\{\langle\sigma,p\rangle\}\uplus S \Rightarrow \{s\}\uplus S} \quad \text{(step)}$$

The final rule expresses synchronous communication: the only one to involve two processes and hence the only way two processes may influence each other. One process must be waiting to write on channel c; another must be waiting to read on c. Only when both are satisfied can both processes advance.

$$\frac{\langle\sigma,p\rangle \xrightarrow{c\ \text{put}\ n} s \qquad \langle\sigma',p'\rangle \xrightarrow{c\ \text{get}\ n} s'}{\{\langle\sigma,p\rangle,\langle\sigma',p'\rangle\}\uplus S \Rightarrow \{s,s'\}\uplus S} \quad \text{(sync)}$$

To guarantee determinism, we require each channel to have a unique reading process and a unique writing process, an easily-checked syntactic constraint. While such a restriction is stronger than necessary for determinism—that (sync) has no choice of which processes may communicate is enough—more liberal rules would require a more costly analysis.

IV. Motivating Example

Our choice of model comes from the observation of many embedded hardware/software systems. Here we describe one commercial embedded system that is representative of many microprocessor-based real-time systems and how to model it in SHIM.

In 1981, Bally/Midway produced the Robby Roto video arcade game.[1] Although primitive by today's standards, it is representative of many early arcade games and illustrates a realistic, commercial embedded system.

Robby is a bus-based microprocessor system with support for video, sound, and some simple input devices. Built around a Z80 running at about 1.8 MHz, it contains the usual RAMs (both static and dynamic), ROMs, and memory-mapped I/O devices, including a video controller with bit-mapped graphics, a hardware blitter, and a pair of sound synthesizers.

Robby employs the usual mechanisms for communicating between hardware and software: memory mapped I/O for software-initiated communication and interrupts for hardware-initiated. The video display is the only source of interrupts in the system. It can generate two types: a light pen interrupt that goes unused in the Robby game (an artifact of its home arcade system origins), and a scan-line interrupt that can be triggered at any scan line under program control.

During gameplay, Robby uses the scanline interrupt feature to invoke three separate routines at lines 50, 100, and 200. Each of these immediately schedules the next one in sequence. Together, they synchronize the software to the frame rate.

Overall, Robby is a synchronous system that operates in lockstep with the video display. Clocks include the 14 MHz pixel clock, the 1.8 MHz Z80 clock, the 31 kHz line clock, the 180 Hz software clock, and the 60 Hz frame clock. Not

[1] Robby is unique among commercial arcade games because Jamie Fenton, the author of its software, released it to the public domain in 1999. See http://www.fentonia.com/bio/

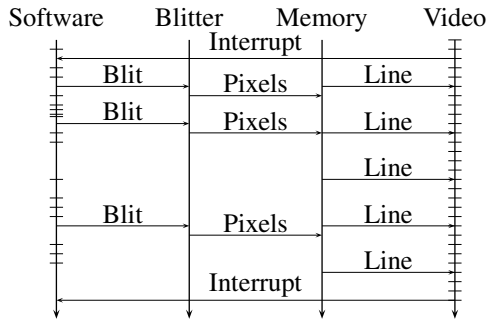


Fig. 2. A message sequence chart illustrating part of the hardware/software interaction of Robby. Time runs from top to bottom, downward arrows indicate concurrently-running processes, and horizontal arrows indicate communication. Tick marks suggest the hardware and software clocks.

unusual for such systems, the slowest clock is separated from the fastest by nearly six orders of magnitude, which would make it inefficient to simulate everything at the fastest clock frequency.

While technically the behavior of every part of Robby in each clock cycle is determined, the designers certainly did not conceive of it that way. Instead, each system (e.g., video, sound) marches to its own clock, or in the case of the software, is actually a collection of unscheduled (in the sense that the exact running time was not considered) assembly-language instructions. At some point, we presume the designers verified the software met its timing constraints, i.e., that each interrupt routine was able to complete its task before the next interrupt occurred.

We designed the SHIM model to capture this mix of multi-rate synchronous hardware and software that is scheduled both coarsely (e.g., the software) and finely (e.g., the video display).

A. Software and Video Interaction

Figure 2 illustrates the original interaction of the software with the video system, which raises a number of interesting issues. At a coarse level, the software runs synchronously with the video system (a periodic interrupt from the video system is the software clock), but at a finer level, the software is asynchronous, running a complex mix of instructions whose exact running time is difficult to compute. The software occasionally invokes a hardware blitter to draw objects on the screen, which writes directly into the video memory. Meanwhile, the video system is synchronous, reading data from memory and continuously sending a stream of pixels to the display.

In the existing game, there is a danger of nondeterministic behavior because the blitter and video display is apparently unsynchronized. Depending on when in the frame a particular blit operation is requested, the effects may become visible in the current frame, in the next frame, or a combination of the two. The designer may have manually scheduled the code to avoid this problem (e.g., by making sure important blit operations happen during vertical refresh), but this is not clear.

Double-buffering is one well-known solution to the problem. This uses two memory spaces for the frame buffer. At any time, one space is being displayed while the other is being modified, and their roles are swapped after each frame.

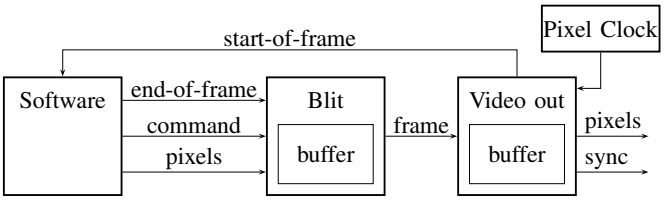


Fig. 3. The software, blit, and video processes implementing a double-buffered display. At the end of each frame, the software signals the blitter memory to transfer its contents to the video display. The video system signals the software at the start of each frame.

```
while 1 do
  while Read end-of-frame is not true do
    Read the blit command
    Write the pixels to memory
  Write the frame to the video process
```

Fig. 4. Pseudocode for the blitter process.

Figure 3 is our model of the game. We added an "end-of-frame" channel from the software to the blitter and an additional video buffer. The blitter process (Figure 4) repeatedly takes an end-of-frame message that indicates whether the software is done updating the current frame. When another object needs to be displayed (i.e., when end-of-frame is false), the blitter then received a command followed by stream of pixels to be displayed. When the frame is done, the blitter sends the frame to the video-out process. Since the size of each frame is fixed, the video-out process knows when to read the next frame from the blitter.

Although it appears the contents of the entire video frame is copied from the blitter to the video-out buffer, this is merely one way to interpret the model, not necessarily how it must be implemented. In general, communication may be implemented in a variety of ways, including through shared memory, which is the typical way to implement a video display. In fact, the transmission of the frame called for in the model would probably be implemented by simply exchanging the roles of two halves of a shared memory.

The software process is straightforward (Figure 5). It is a loop that periodically waits for (reads from) the start-of-frame signal. Within each cycle, in addition to executing the game logic (where the enemies move, what score the player has achieved, etc.), the software occasionally invokes the blitter to draw objects on the screen such as the player and the enemies.

Modeling the video process is also easy (Figure 6). It

```
while the player is alive do
  Wait for (read from) start-of-frame
  ...game logic...
  Write "false" to end-of-frame
  Write to the blitter
  ...game logic...
  Write "true" to end-of-frame
```

Fig. 5. Pseudocode for the software process.

while 1 **do**
  Write start-of-frame
  for each line **do**
    Emit line timing signals
    for each pixel **do**
      Read (wait for) the pixel clock
      Read the pixel from memory
      Send the pixel to the display
  Read the next frame from the blitter

Fig. 6. Pseudocode for the video process.

```
write(o, v); /* Optional: the initial value */
while (1) {
  read(i, v);
  write(o, v);
}
```

Fig. 7. A single-place buffer process in Tiny-SHIM with an initial value. The `i` channel is the input; `o` is the output. Stringing these in series gives arbitrary-sized buffers.

consists of nested loops that read from memory to generate a sequence of pixels to send to the display. An external pixel clock is used to synchronize this system, and writing to the start-of-frame channel synchronizes the software to the video system.

V. SHIM Design Patterns

A design pattern is an idiom that can be used to solve recurring problems in system design. While we do not have enough examples of SHIM systems to make a thorough study of such patterns, here we present a number of them we expect will be useful in practice.

A. Buffers

Although communication in SHIM is unbuffered, it is easy to create finite-size buffered communication channels. For a buffered channel of size n, introduce a chain of n single-place buffer processes (Figure 7) that repeatedly read a value from an input channel and immediately write to an output channel. To initialize the contents of a channel, begin one or more of the processes with a series of *write* statements.

We expect such buffers will be a very common design idiom. In the larger SHIM language, we plan to provide syntactic sugar for specifying buffers. Three parameters characterize a buffer: its size, the type of data it conveys, and any initial contents.

B. Interrupts

There are two issues with interrupts and SHIM. The first is the use of interrupts in the implementation of a SHIM model. This seems like a natural way to implement channels on which there is infrequent communication, or when integrating existing hardware with a SHIM system.

Interrupts are also sometimes used to wake up a sleeping (halted) processor, perhaps in response to a keystroke. This could be modeled in SHIM, say, by adding a "wake up" channel to a process representing a processor that would be implemented in this way. A second periodic process that waited for a keystroke could periodically poll and emit the "wake up" event when one was detected. How to inform the synthesis system that such an idiom should be implemented with such a mechanism is outside the scope of this paper.

```
/* Interrupting hardware process */
write(int, v);
/* ... */

/* Interrupt handler process */
while (1) {
  read(int, d); /* Acknowledge interrupt */
  /* process d */
  write(buf, d);
}

/* Software process */
while (1) {
  read(buf, q); /* Get data from the interrupt */
  /* do something with q */
}
```

Fig. 8. A synchronous interrupt handler: the hardware generates a communication that is buffered by the interrupt handler process before being sent to the software process.

The other issue is explicitly modeling interrupt-like behavior in a SHIM model, i.e., stopping a process at some point, making it handle incoming data, then resuming from the point at which it was interrupted. We cannot model traditional software interrupts in their full generality because they are nondeterministic.

Instead, we suggest the effects of well-behaved interrupts be represented with three processes: the hardware generating the interrupt, the interrupt handler, and the software program being interrupted. For continuous streams of data, such as from a network controller, Figure 8 illustrates a group of such processes. Very little is going on here: the interrupt handler is just a buffer.

Because our systems are deterministic, we do not support truly asynchronous interrupts (i.e., that can at any time affect the execution of a program). But such behavior is sometimes needed, e.g., in a network connection where an out-of-band signal can signal the end of a connection. To obtain this behavior, a designer must write in a polling style, i.e., explicitly indicate where an "interrupt" may occur and check for it. Figure 9 is an example illustrating this style.

In general, such a polling style works for any aperiodic event that may be interleaved with periodic events. A key challenge becomes balancing the relative execution rates of the two processes so that one is not constantly forced to wait for the other. The determinism of our model guarantees that the behavior of such systems is always correct and well-defined, but tuning to improve performance may still be necessary. As usual, adding buffering to a system makes it resilient to jitter, but not to mismatched rates.

C. Synchronous Processes

The synchronous model, such as that in the Esterel [7] and Lustre [8], has proven itself useful in modeling a large class of systems. In it, synchronous processes march to a common clock and communicate in a broadcast style. Representing such

```
/* Interrupting hardware process */
if (should_interrupt) {
  write(int, 1);
  write(intdata, v);
} else {
  write(int, 0);
}
/* ... */

/* Interrupt handler process */
while (1) {
  read(int, flag);
  if (flag) {
    read(intdata, q); /* acknowledge interrupt */
    /* ... perform simple computations on q ... */
    write(interrupted, 1);
    write(buf, q);
  } else {
    write(interrupted, 0);
  }
}

/* Software process */
while (1) {
  read(interrupted, f); /* check for interrupt */
  if (f) {
    read(buf, q); /* get interrupt data */
    /* handle the interrupt */
  } else {
    /* normal operation */
  }
}
```

Fig. 9. An asynchronous interrupt handler. Here, the hardware periodically emits a signal (int) that indicates whether an interrupting condition has occurred. Similarly, the software periodically checks this signal.

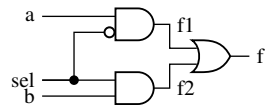
systems in SHIM can be useful, either as a mechanism for importing code from existing system, or as a design pattern (e.g., certain systems might be best modeled as synchronous islands communicating asynchronously).

In SHIM, synchronous systems can be expressed by introducing "redundant" communication. A synchronous process in our model must periodically communicate with all of its peers (i.e., every process with which it ever communicates). The one-to-many channels typical in synchronous models can be emulated with "fanout" processes that repeatedly read from an input channel and replicate the data on every output channel.

Berry and Sentovich [9] propose a low-level way to implement a synchronous formalism in an asynchronous model of computation. They translate a synchronous model into a network of logic gates and then assemble a network of processes with the same topology that simulate the gate network. Each process alternates between a calculation and reset phase, and together, all the processes march in lockstep to a single clock.

Figure 10 shows a gate-level model of a multiplexer illustrating a simplified construction style inspired by Berry and Sentovich. Each gate is a simple process that reads all its inputs and produces an output whose value is the function of the gate. In each cycle, a value must be presented on every input to produce all the outputs. This style cannot model cyclic combinational circuits; it would deadlock.

It is not necessary to model synchronous systems in a gate-level style, but its periodic communication is typical. Figure 11 shows a small Esterel [7] program and two TinySHIM processes that implement its behavior. The behavior of this Esterel program illustrates one of the subtleties of




```
/* fork */
while (1) {
  read(sel, v);
  write(sel1, v);
  write(sel2, v);
}

/* AND-I gate */
while (1) {
  read(a, x);
  read(sel1, y);
  write(f1, x & !y);
}
```

```
/* AND gate */
while (1) {
  read(b, x);
  read(sel2, y);
  write(f2, x & y);
}

/* OR gate */
while (1) {
  read(f1, x);
  read(f2, y);
  write(f, x || y);
}
```

Fig. 10. A gate-level model of a multiplexer consisting of four processes. The `sel` input selects whether the `a` or `b` input is passed to the output `f`. Note that although the value one of the inputs is always ignored, communication must take place on both inputs to produce an output.

```
module reqack:
input S, I;
output O;

signal R, A in
  every S do
    await I;
    weak abort
      sustain R
    when immediate A;
    emit O
  ||
    loop
      pause; pause;
      present R then
        emit A
      end
    end loop
  end every
end signal

end module
```

(a)

```
s = 0;
while ( s == 0 ) {
  read(S, s);
  read(R, r);
  write(A, 0);
}
while (1) {
  read(S, s);
  read(R, r);
  write(A, 0);
  if (s == 0) {
    read(S, s);
    read(R, r);
    if (s)
      write(A, 0);
    else
      write(A, r);
  }
}
```

(b)

```
state = 1;
while (1) {
  read(S, s);
  read(I, i);
  nop = 1;
  if (state == 1)
    state = 2;
  else if (state == 2) {
    if (s) state = 3;
  } else if (state == 3) {
    if (!s && i) {
      nop = 0;
      write(R, 1);
      read(A, a);
      if (a) {
        write(O, 1);
        state = 2;
      } else {
        write(O, 0);
        state = 4;
      }
    }
  } else { /* state == 4 */
    if (s) {
      state = 3;
    } else {
      nop = 0;
      write(R, 1);
      read(A, a);
      if (a) {
        write(O, 1);
        state = 2;
      } else {
        write(O, 0);
      }
    }
  }
  if (nop) {
    write(R, 0);
    read(A, a);
    write(O, 0);
  }
}
```

(c)

Fig. 11. (a) An Esterel program modeling a request/acknowledge handshake and a translation of it into two SHIM processes. Process (b) is written in an imperative style and models the second half of the Esterel program. Process (c) is written in a state-machine style and models the first half.

the Esterel language: the two concurrently-running threads (separated by the || operator) communicate bidirectionally within a cycle, i.e., the first process can send *R* to the second process, which can respond with an *A* and cause the first process to emit an *O* in response. Thus, the execution of the two threads must be interleaved within a cycle.

The process in Figure 11b, which corresponds to the second half of the Esterel program, is simpler because its control behavior is much cleaner and because its execution does not need to be interleaved. Each "cycle" starts by reading *S* and *R*. In alternate cycles, the value of *R* is sent on *A*. Note that the communication order is always *S*-*R*-*A*; making synchronously communicating processes follow such a simple discipline makes it easy to avoid deadlock.

The process in Figure 11c is much more complicated, partially because of the richer Esterel constructs (*sustain*, *weak abort*), but also because the reception of *A* must always follow the transmission of *R*. Not doing this would lead to a deadlock when this process was combined with the other since the other process generates *A* only in response to *R*. As a result, this process is constructed in such a way that it always communicates in the order *S*-*I*-*R*-*A*-*O* to avoid introducing a deadlock with the other process.

We translated these processes manually. Figure 11b was simple enough to do by inspection, but the control behavior of Figure 11c was complex enough that we first generated an automata for the process (Berry and Gonthier [7] describe how to generate automata from Esterel), then coded it in Tiny-SHIM. The procedure described by Zeng et al. [10] for translating Esterel to software might give a more efficient implementation.

D. Synchronous Dataflow

Lee and Messerschmitt's Synchronous Dataflow [11]—SDF—can be implemented in SHIM once buffer sizes are known. Each SDF actor becomes a process connected through finite-size buffers. Inconsistent-rate SDF systems will eventually deadlock since our model does not allow unbounded accumulation of data on buffers.

Interestingly, because SHIM communication is bounded, the rate-computation step in SDF scheduling is unnecessary for SHIM. Systems that can run forever in a specific set of buffer sizes will run forever with any fair scheduler; the finite capacity of the buffers effectively enforces relative process execution rates.

Much of the SDF scheduling machinery remains useful for SHIM. Choosing buffer sizes is an important step and SDF scheduling algorithms can choose them appropriately. One way to do this would be to compute the relative execution rates, find some schedule, and simply use the maximum buffer sizes required by that schedule. This is sufficient, but may require larger buffers than necessary or produce less efficient behavior.

Going the other way, an obvious improvement would be to perform more static scheduling of SHIM systems. Fully static scheduling may not be practical. Adding data-dependent choice is tricky, as Buck [4] showed, but the techniques of Lin and Zhu [12], [13] as well as Cortadella et al. [14] suggest there are many possibilities.

Along these lines, it might be useful to identify SDF-like subsystems in SHIM systems and apply some of the very sophisticated SDF scheduling techniques [15] to them. The result would be a system with a mix of static and dynamic scheduling, which seems appropriate for certain classes of applications.

E. Timing

In SHIM, communication serves the double purpose of synchronization and data transfer. Ensuring precise timing, therefore, can be done through synchronization to a periodic clock, and while it might appear that SHIM would demand that a clock wait for a slow process, our vision is to employ a form of static timing analysis to determine that the process will always be faster than its clock. While well-known for hardware, static timing analysis is more difficult for software because of its unpredictable nature.

An optimization procedure would start by trying to derive a quasi-static schedule for the communication actions in a SHIM system. For example, analyzing the process in Figure 11c would give the simple periodic communication order *S*-*I*-*R*-*A*-*O*. Comparing this with the process in Figure 11c, which has the order *S*-*R*-*A*, the algorithm would note that the second process runs "in the middle" of the first. If these two were implemented in parallel, the algorithm might observe that the second process always tries to read *R* before the first process can write it, meaning that this communication action in the first process would never block and any machinery (either in hardware or software) to check to see whether the second process is ready could be discarded.

F. Sensors

We can model sensors—unsynchronized time-varying environmental signals—as processes with a single output through which the sensor value is constantly available to be read. While the timing of such values would be difficult to control without an additional clock signal, the system will respond only to the sequence of values it receives from the sensors.

G. Arbitration

Although SHIM prohibits nondeterministic access to shared resources, it can describe deterministic arbiters. Choosing an appropriate arbitration algorithm is the responsibility of the designer. Round-robin, hold until release, or some more complicated mechanisms are possible; the choice will vary with the application.

VI. Software Implementations

We describe two techniques for implementing SHIM systems in single-threaded software. By design, these are not the only possible implementations, and are certainly not the most efficient, but they illustrates how simple SHIM systems are to execute and points the way to more efficient techniques. For example, Lin and Zhu [12], [13] describe a more efficient

```
Mark all processes as ready
while there is some ready process do
  Fairly select a ready process p
  if no instruction is left in p then
    Mark p as terminated
  else if p reached read(c,...) or write(c,...) then
    if another process p' is blocked on c then
      Synchronize p and p' and mark p' as ready
    else
      Mark p as blocked on c
  else
    Execute one step of p
```

Fig. 12. The basic software scheduling algorithm.

quasi-static technique for a similar model that unfortunately may produce exponentially-large code.

The most basic algorithm, Figure 12, consists of a preemptive scheduler that orchestrates the execution of the processes. Repeatedly, the scheduler chooses a runnable process and passes control to it. This process executes a single step (e.g., an assignment, a test) independently from other processes, or synchronizes with another process, or fails to do so and blocks, in any case passing the control back to the scheduler.

A. Fairness and Preemption

The algorithm in Figure 12 performs preemptive, fair scheduling to ensure that every system executes as much as possible, but such a pedantic approach is often unnecessary. Many systems can be executed with an unfair, non-preemptive scheduler (i.e., one that only regains control from a process when the process reaches a *read* or *write* statement or terminates), which is often more efficient; such scheduling policies are permitted by the structure of communication in most well-behaved systems.

First of all, systems that terminate or deadlock (i.e., reach a point where every process has either terminated or is waiting for communication on a channel and no two processes are waiting on the same channel) do not need preemptive or fair schedulers. It follows from the determinism of our systems that any correct scheduling procedure (i.e., is always running some ready process) will ultimately reach this point. However, many interesting embedded systems are non-terminating, so we will consider them in more detail.

Two subclasses of systems are interesting: cooperative systems, which can be executed indefinitely with a non-preemptive scheduler; and dynamically connected systems, a type of cooperative system whose communication behavior makes it impossible for an unfair scheduling policy to cause process starvation. A cooperative system is one in which no process diverges, i.e., fails to either terminate or initiate communication beyond a point. Informally, a system is cooperative if its processes never enter infinite loops that do not contain a communication action. This is a dynamic property of the whole system since a process may make a data-dependent choice to enter such a loop.

By design, a cooperative system can be scheduled with a non-preemptive scheduler because any process will eventually relinquish control to the scheduler. However, a cooperative system may still require a fair scheduling policy. Consider a system consisting of two pairs of mutually-communicating processes that do not otherwise communicate. An unfair scheduler may choose to execute only one of the two pairs of processes, which is undesirable because the system will not approach its correct behavior in the limit.

Dynamically connected systems are an interesting subclass of cooperative systems whose communication behavior ensures fair execution even without a fair scheduling policy. The processes in a dynamically connected system may not terminate, and the graph of communication channels over which an infinite number of communication take place must be connected, i.e., there cannot be two or more islands of processes with non-infinite communication between them.

To see why a dynamically connected system can be executed with an unfair, non-preemptive scheduler, consider an unfair scheduler that tries to starve a particular process p. By definition, p must try to communicate infinitely often through at least one of its channels. If the scheduler starves p, it will eventually block the other endpoint of this channel, which will eventually block that process, and by induction all other processes in the system since the graph of infinitely-communicating processes is connected. The system will reach the point where every other process is blocked and the scheduler will be compelled to execute p, thus breaking the logjam.

We expect most interesting embedded systems will be dynamically connected since most systems do not deliberately shut parts of themselves down forever. This is good since unfair, non-preemptive schedulers are usually more efficient than their fair, preemptive counterparts. The one possible exception would be systems whose execution starts with an initialization phase, which could include some terminating processes. However, if these had to communicate with the infinitely-running remainder of the system, an unfair scheduler would still work.

B. An Efficient C Implementation

Here, we present an implementation of SHIM systems in C. The scheduler is neither fair nor preemptive, but it is simple to implement and correctly simulates dynamically connected systems as discussed above.

Our Tiny-SHIM compiler, which is about 4000 lines of OCAML, translates each process into a C function. The execution of these functions is coordinated by a simple scheduler. Figure 13 is a complete example of a generated program for a pair of simple processes. Shown in boxes, process *p1* writes 42 on channel C, which the second process, *p2*, receives. This is the simplest example that illustrates communication.

The scheduler, the *main()* function at the bottom of Figure 13, repeatedly removes and invokes the first process on a linked list of runnable processes. By design, this scheduler is extremely simple; all the action happens at the communication events in the processes, where context switching and scheduling of other processes occurs.

For each process, the compiler generates three things: an integer state variable (in Figure 13, *p1_state* and *p2_state*); a

```
typedef struct process_struct { /* Linked list cell*/
  void (*process)(void);       /* process function */
  struct process_struct *next; /*     next in list */
} process_t;

typedef struct {                          /* Channel */
  int value;            /* value being transferred */
  process_t *waiting;   /* blocked process, if any */
} channel_t;

channel_t C = { 0, 0 }; /* definition of channel C */

int p1_state = 0;                 /* process state */
int p2_state = 0;
void p1_function(void);    /* forward declarations */
void p2_function(void);

/* Linked list of runnable processes */
process_t p1 = { p1_function, 0 };
process_t p2 = { p2_function, &p1 };
process_t *head_process = &p2;

void p1_function() {
  switch (p1_state) {   /* resume at current state */
  case 1: goto L1;
  case 0: goto L0;
  }

 L0:
  /* write(C, 42) */
  C.value = 42;
  if (C.waiting) {
    (C.waiting)->next = head_process;  /* schedule */
    head_process = C.waiting;   /* reading process */
  }
  C.waiting = &p1; p1_state = 1;
  return;                                 /* suspend  */
 L1:
  ;
}

void p2_function() {
  static int v;
  switch (p2_state) {   /* resume at current state */
  case 0: goto L0;
  case 1: goto L1;
  }

 L0:
  /* read(C, v) */
  if (!C.waiting) {
    C.waiting = &p2; p2_state = 1;
    return;                                /* suspend */
  }
 L1:
  v = C.value;
  (C.waiting)->next = head_process;    /* schedule */
  head_process = C.waiting;     /* writing process */
  C.waiting = 0;
}

int main()                                /* Scheduler */
{
  process_t *running_process;
  while (head_process) {
    running_process = head_process; /* remove head */
    head_process = running_process->next;
    (*(running_process->process))();     /* run it */
  }
  return 0; /* everything terminated or deadlocked */
}
```

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

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

Fig. 13. A complete example of synthesized Tiny-SHIM code that includes two processes (in the boxes) that communicate and the `main()` function that schedules them.

function that, when called, runs the process until it reaches a *read* or *write* statement; and a cell for a linked list that points to the process function. Each cell is an object of type *process_t* that also holds a pointer to the next cell in the linked list. Each cell is given the name of its process.

At any time, a process may be in one of four states: running, runnable, blocked on a channel, or terminated. These states are distinguished largely by the location of the cell for each process. When a process is runnable, its cell is linked into the list of all runnable processes. When a process is blocked on a channel, the data structure for the channel holds the process's cell. When a process is running, its cell is effectively held by the code for the process, which will place it in a channel structure when the process blocks on a channel. When a process terminates, the cell for the process is effectively forgotten.

When the system starts, every process is runnable by definition, so the cells for all processes are linked together. Note that the determinism property tells us that the order in which processes appear in the list does not matter; we chose the order in Figure 13 because it was easy to generate.

Processes communicate through the *channel_t* data type. Each holds two things: the value being communicated (even though the communication behaves synchronously, a sequential software implementation requires this single-place buffer), and a pointer to the cell for process, if any, that is blocked waiting to communicate on this channel. Note that the *next* field of a cell being held by a channel is unused and immediately overwritten when the cell is added to the runnable list. Since the channels in our systems are fixed, all *channel_t* objects are allocated statically. In Figure 13, there is a single channel, C, that is initialized near the top of the program.

The code for each process is translated into a function—*p1_function* and *p2_function* in Figure 13—that begins with a *switch* statement that uses the process's state variable to send control to just after where the process last suspended itself.

That a process must be able to suspend and resume itself means that the SHIM model cannot be implemented as a pure C library without insisting the programmer do something like insert a *switch* statement like the one we use. The semantics require some sort of concurrent semantics, such as coroutine-like suspend/resume behavior. The SystemC simulation library provides such behavior (Liao et al. [16] describe the SystemC kernel under its old name, Scenic), but relies on a threads package implemented in part in assembly language. While Tiny-SHIM could take a similar approach, the compilation approach we propose makes the generated code more efficient, portable, and opens the door to optimizations. The other disadvantage of a library-based approach is that it cannot provide any guarantee of correctness, e.g., a library user could use C constructs to circumvent the channel-based communication model.

Each process starts with an initial label, *L0*, that corresponds to the beginning of the process. After that, each *read* and *write* statement generates another label with a distinct state. The state variables for each process, *p1_state* and *p2_state*, are encoded with sequential integers. A separate terminated state for each process is not needed because once control falls off the end of a process, the system effectively forgets about it

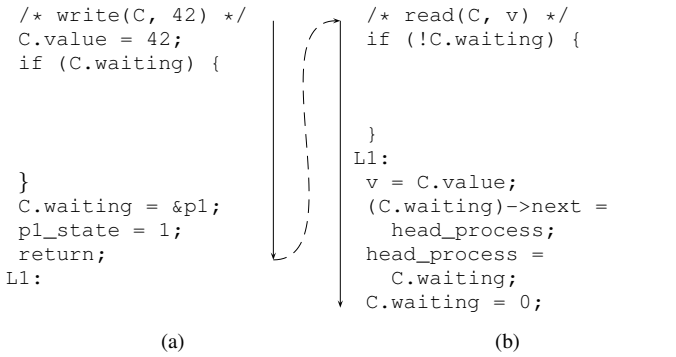


Fig. 14. When (a) write runs before (b) read, the writing process stores the value, indicates that it is waiting on the channel, and suspends itself. Later (indicated by a dashed line), the reading process captures the value, schedules the writing process, and clears the channel.

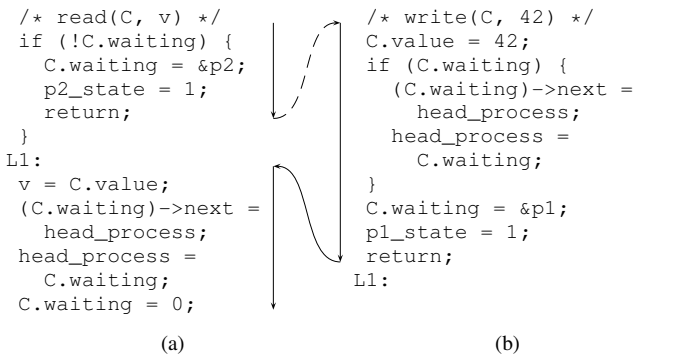


Fig. 15. When (a) read runs before (b) write, the reading process indicates that it is waiting on the channel and suspends. Later (indicated by a dashed line), the writing process stores its value, schedules the reading process, and suspends, which immediately passes control to the reading process. Finally, the reading process reads the value, schedules the writing process, and clears the channel.

and will never again place its cell on the runnable list.

The generated code for the statements in each process is straightforward (i.e., one-to-one) except for the code for *read* and *write* statements. The code for these statements is clever and relies heavily on both the point-to-point nature of communication and the behavior of the scheduler. There are two cases. Figure 14 depicts the behavior of the simpler case: when a write operation runs before the corresponding read. Here, the writer saves its value and blocks. Later, when the reading process reaches a read on the channel (this may not happen; the writer may be blocked forever), it reads the value and schedules the waiting writer process.

Figure 15 shows the more complicated case, when a read is reached before a corresponding write. Here, the reader immediately suspends itself. If a corresponding write is reached, it writes the data and *makes the reader execute immediately* by scheduling it. Because the scheduler always runs the process at the front of the list, the *return* statement in the write process (Figure 15b) effectively passes control directly to the read process, which captures the value, schedules the writer to resume later, and clears the channel.

The writer immediately transfers control to the reader in this manner to avoid the possibility of overwriting the data in the channel. If two writes to the same channel occurred without an intervening read, the data would be overwritten.

The SHIM communication semantics suggest a visualization of system behavior: Figure 16 is a message-sequence-chart-like diagram that was generated from the execution of the processes in Figure 11. Arrows indicate communication and dark line segments indicate which process is running at any given time. The length of these segments does not denote actual execution time, although the diagram could be modified so they do.

The behavior near the top of Figure 16 illustrates an interesting effect that can arise when implementing synchronous specifications in SHIM: although everything is synchronous, the scheduler discovered an opportunity for pipelining. Specifically, the printer process, which repeatedly reads from the O channel, does not run until after the testbench process has generated the stimulus (specifically, the S signal) for the next cycle. It is only the attempt by the sfork process to transmit to the requester process that forces the printer process to run.

After this point, the schedule settles down into predictable, periodic behavior. For efficiency, it would have been nice for the scheduler to have behaved this way from the beginning, but the determinism of the SHIM model ensures this is only an optimization and that it does not change the sequence of events communicated on each channel.

VII. A Hardware Implementation

To complement the software implementation presented in the previous section, in this section we present a syntax-directed translation of Tiny-SHIM into synchronous digital hardware. As in the software case, the SHIM semantics admit many other translations as well as optimizations of this one. Thus, this particular translation is meant to illustrate the issues in a hardware implementation rather than be an ultimate solution.

Like Berry's translation of Esterel [17], our technique uses a template for each type of statement and produces a circuit whose structure follows the control-flow graph of the program. A true value on a wire in a cycle indicates control passes through the corresponding part of the program in that cycle.

Our templates are simpler than Berry's because our language does not include the preemption constructs of Esterel, but our translation deals with dataflow using static single-assignment analysis. We employ the algorithm of Cytron et al. [18] and construct a circuit using a technique like that of Edwards et al. [19].

Our synthesis procedure translates each process into a control-flow graph with four node types: assignments, decisions, merges, and cycle boundaries. Static single-assignment analysis then identifies the data pathways, and finally the control-flow graph and datapath information is mechanically translated into gates.

Figure 17 shows the four types of blocks in the control-flow graph and how they are translated into circuitry. Each block is translated into a control circuit fragment, which implements the control-flow of the imperative code, and a datapath fragment, which implements operations on variables.

An action block, which assigns the value of a (side-effect-free) expression to a variable, has a trivial control fragment:

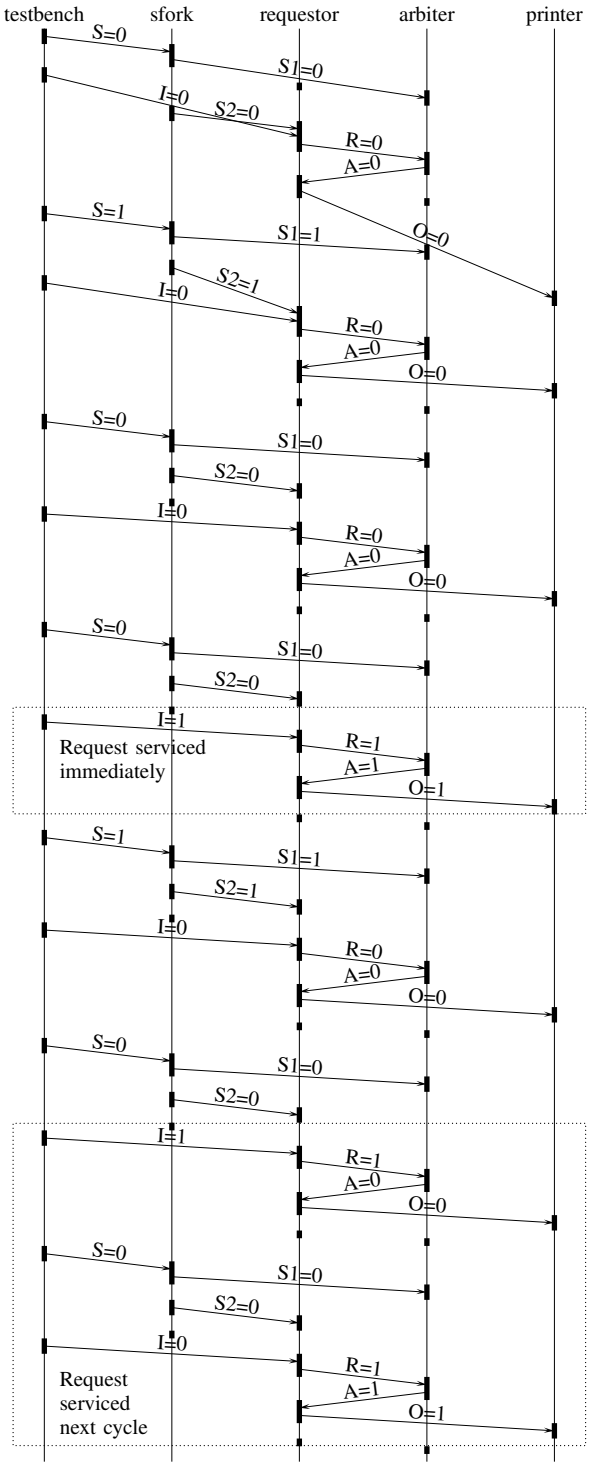


Fig. 16. A message sequence chart generated from the synchronous processes in Figure 11. Time runs from top to bottom. Dark line segments indicate when each process is running; their length does not denote actual running time. Three additional processes were added: a testbench process that generates a pattern of S and I input signals, a fork process that fans out the S signal onto two channels, and a printer process that acts as a sink.

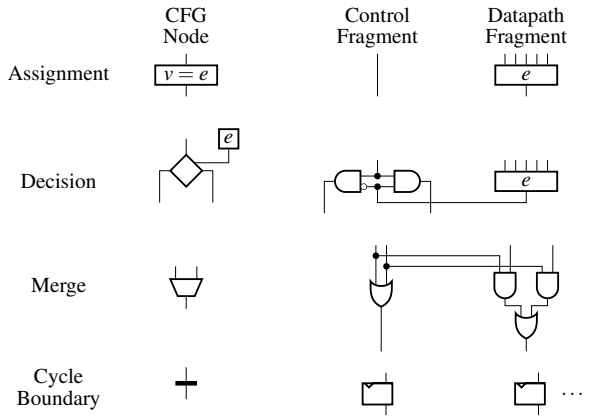


Fig. 17. The four types of control-flow blocks and their hardware equivalents. The signal flow in the hardware schematic fragments follows the structure of the control-flow graph.

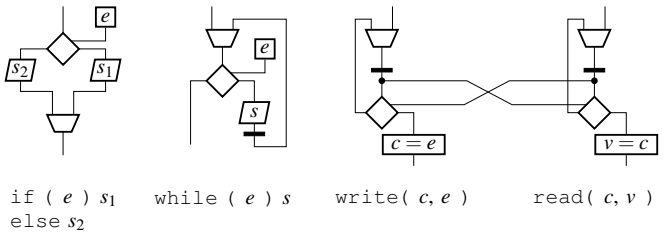


Fig. 18. The translation of Tiny-SHIM statements into control-flow graph fragments.

just a wire that passes control to the next statement in order. The complexity naturally comes in the datapath, which calculates the value of the expression.

A decision block evaluates its expression and passes a Boolean value back to a control circuit, which passes control to either its *then* or *else* branch.

A merge block forms the logical OR of its two (mutually exclusive) control inputs; the datapath implements a multiplexer that selects between variables coming from its two incoming branches. These are the concrete realization of the ϕ functions in static single-assignment form.

Finally, a cycle boundary turns into a collection of registers: one for the control path, and one for each bit of each live variable crossing the cycle boundary.

Figure 18 shows how we translate each statement in Tiny-SHIM into a control-flow graph fragment. The *if-else* statement is straightforward; notice that it executes in a single cycle if its bodies do. The *while* statement is mostly a decision in a loop, but a cycle boundary after the body ensures that no combinational cycles are produced. In many cases, this extra cycle is not necessary; eliminating these is an obvious optimization that we will consider in the future.

The template for communication is the richest: a pair of post-test loops each containing a cycle boundary. Figure 18 shows the simple case: a single *read* matching a single *write*. Figure 19 is a multiple read/write case, which uses a pair of OR gates to collect the various read and write requests before passing them to the other process.

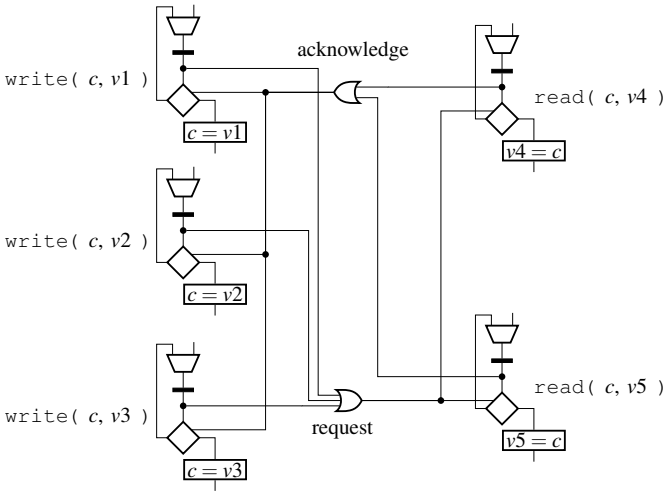


Fig. 19. An example of translating multiple communication actions on the same channel. This assumes that control is at most one read and one write in any cycle, i.e., that at most one communication takes place per channel per cycle. The placement of cycle boundaries in our construction ensures this.

```
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;
  write(D, b);
}
```

Fig. 20. A pair of processes to illustrate the hardware synthesis process. The receiving process on the right reads a value from the channel and uses it to decide whether to immediately read a normal value on the channel or to treat it as an end-of-block marker. The process on the left produces a series of such blocks consisting of descending sequences of numbers.

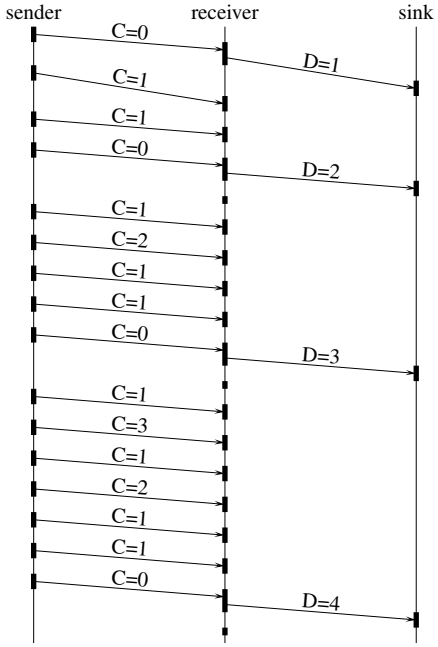


Fig. 21. A message sequence chart illustrating the behavior of the processes in Figure 20. An additional *sink* process was added to receive the values on the *D* channel.

The cycle boundaries in the *read* and *write* transactions force each communication action to take place at least one cycle after it is requested, thus ensuring that at most one communication takes place per channel per cycle. This seemingly wasteful choice greatly simplifies the logic: while it would be possible to construct circuitry that perform multiple communications through a single channel in the same cycle, such circuitry is very complicated in general because communication can be data-dependent. For example, imagine a pair of processes that contain four *read* statements and three *write* statements. If these statements could all execute in a single cycle and each were conditional, the circuitry would have to handle the case where the first *read* matched up with the first *write* or the second *write*, the second read matched up with the first *write* or the second *write* and so forth. The OR gates in Figure 19 would certainly not be enough. We plan to eventually consider such a rich translation, but it will require substantial static analysis.

The OR gates for *read* and *write* collect the "request" signals from their communication counterparts. Our language requires that all *read* states for a particular channel reside in a unique process, and that the corresponding *write* statements for the channel reside in another, different process. By construction, the inputs to each OR gate are exclusive because control can only be at a single point within each process.

To illustrate our translation procedure, consider the pair of processes in Figure 20. Although fairly simple, they illustrate an idiom for (deterministic) arbitration for a shared resource. Each consists of two nested loops; the innermost loops are data-dependent. Furthermore, the communication behavior is also data-dependent, although this example is simple because it uses only a single channel. Figure 21 illustrates the behavior of these two processes plus a *sink* process that receives *D* (not shown);

Figure 22 shows how the code of Figure 20 is translated into a control-flow graph using the templates from Figure 18, which can be translated into hardware using the templates of Figure 17. The two OR gates in the center of Figure 22 determine when the processes attempt to communicate. The circuitry for *write(D, b)* has been omitted for simplicity.

The circuit implied by Figure 22 has a lot of redundancy and presents many opportunities for optimization. In addition to the usual Boolean simplifications, the most interesting aspect of such circuits is their communication pattern. The current translation of *read-write* pairs is relatively complicated because it must cope with all cases, e.g., *read* executed before *write*, *write* runs before *read*, etc. However, as is often the case, the communication pattern in this example is regular and such regularity could be used to greatly simplify the circuitry used for communication. Lin [12] performs exhaustive analysis to determine communication patterns in a model much like ours, although he uses the result for software synthesis.

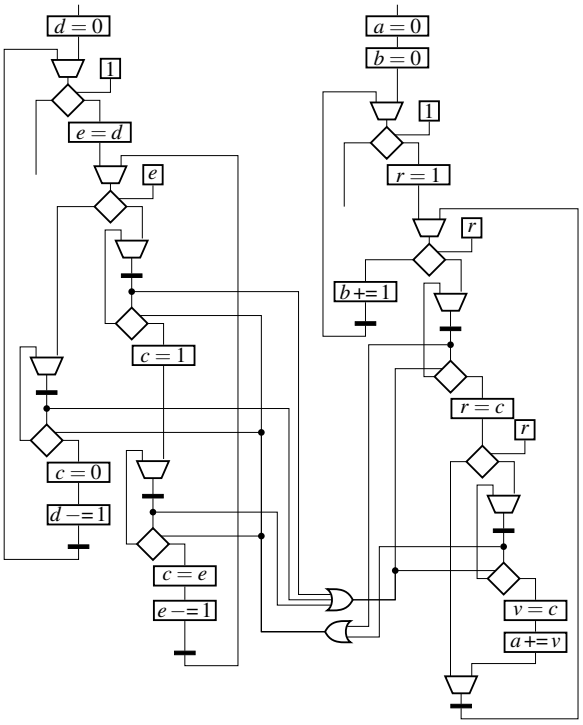


Fig. 22. The translation of the processes in Figure 20. The circuitry for the communication on *D* has been omitted for simplicity.

VIII. Relationship to Other Models

The SHIM model is similar to many existing concurrent models of computation: a restriction of some, a generalization of others. We strove to find the most liberal model that somehow remains tractable.

A. CSP

SHIM differs from Hoare's CSP [2] primarily in its focus on determinism. Like Hoare, we use a rendezvous model of communication in which two communicating processes can only advance when they synchronize, which has the advantage of simple semantics yet can easily model more flexible (and complicated) buffered communication. Hoare's processes also block when waiting for communication, but our insistence that a process may only block on a single channel guarantees determinism.

B. Kahn Networks

SHIM systems are deterministic for much the same reason as Kahn's [1], but are more restrictive. Kahn's processes communicate through unbounded buffers, which can be both an advantage (our systems are subject to deadlock from buffers filling up; Kahn's are not) and a liability. Adding unbounded buffers makes Kahn networks Turing-complete and difficult to schedule since it is desirable to use bounded buffer memory wherever possible. Parks [5] scheduling algorithm does this, but it can be difficult to implement and (understandably) provides no a priori bounds on buffer sizes, a real liability for resource-constrained embedded systems.

Once buffer sizes are fixed, a Kahn network is easily translated into SHIM. Determining these sizes can be difficult in practice, but at least the deterministic property of our model can help to answer the question of whether a particular system will deadlock because of insufficient buffer space.

Like ours, other formalisms are restrictions of Kahn's networks. Karp and Miller's [20] and Lee and Messerschmitt's [11], [21] systems both restrict the behavior of processes in a Kahn-like model to make their relative execution rates predictable. Again, because SHIM can be used for Kahn systems with fixed-sized buffers, these other models can be translated into SHIM with no loss of behavior.

C. Asynchronous Hardware Models

SHIM was inspired in part by van Berkel's asynchronous handshake circuits [22], which show among other things the practicality of implementing a traditional imperative language consisting of assignments, conditionals, and loops using nothing but rendezvous communication. Handshake circuits and the Balsa/Tangram/Haste language, however, are aimed at the challenges of implementing asynchronous digital circuits and as such contain many low-level directives that would not make sense, say, for software.

Another troubling aspect of handshake circuits is their inclusion of arbiters, which break the determinism of the model. While certainly adding to the expressiveness of the model, arbiters suddenly makes simulation more difficult. Janin, Bardsley, and Edwards [23] describe a simulator that takes snapshots of the system at every nondeterministic choice to allow the simulation to be restarted from these points.

Related to CSP, Josephs's Receptive Processes [24] are lower-level than SHIM. Aimed at modeling the gate-level behavior of asynchronous circuits, they do not explicitly represent data, assuming instead that it is encoded in interaction order. Josephs also proposed a deterministic variant [25] that somehow the same as Kahn's processes[26], but they remain at a very low-level of abstraction that is inappropriate for software. For example, they do not speak of numbers, only single bits.

The Polis project [27] had aims similar to ours. They proposed a unifying model of computation that could support both hardware and software implementations (CFSMs [28]) and constructed simulators and hardware and software synthesizers around it. Their model, however, is nondeterministic and its specification of processes rather abstract, making it difficult to synthesize large pieces of software.

D. Synchronous Models

Synchronous models [29] are also concurrent and deterministic. While attractive, these models place a bigger scheduling burden on a designer and thus tend to be better-suited for lower-level models. Our motivation for using an asynchronous model came in part from trying to model something like the video game described in Section IV in a purely synchronous model. That system is most naturally described as multi-rate, with clocks ranging from pixel-speed to frame-speed.

The synchronous languages Lustre [8], and Signal [30] both handle multi-rate dataflow, but only the Esterel [7] language supports an imperative style of coding—natural for

software—and unfortunately its support for multi-rate behavior is currently lacking, despite a number of attempts. Berry and Sentovich's construction [9] show that the Esterel semantics can be implemented in an asynchronous model.

E. Heterogeneous Models

Projects such as Lee's Ptolemy [31] take a different approach to modeling hardware/software systems. Ptolemy is primarily a flexible simulation environment in which different models of computation can be supplied in the form of "domains." Communication between domains, however, has largely been ad hoc, and the main focus of Ptolemy has been simulation rather than synthesis. SHIM could easily be implemented as a domain in Ptolemy.

The Metropolis project [32], a follow-on to the Polis project [27], tries to provide a more structured environment for multiple models of computation. They provide a meta-modeling language in which different models of computation, such as SHIM, can be specified. Like Ptolemy, Metropolis could be used to implement SHIM.

F. FairThreads

Like SHIM, FairThreads [33] also provide deterministic concurrency. Our approach in SHIM is that of Kahn networks: restricted communication patterns ensure determinism for any fair scheduling policy. By contrast, FairThreads achieves determinism through a fixed scheduling policy: a cooperative round-robin algorithm. The scheduler cannot interrupt the running thread. Each process must explicitly pass control to the scheduler, which is compelled to run the next thread in a predefined sequence. While this approach makes it possible to implement more complex communication schemes, it is only relevant for software running on a single processor, and is thus not applicable for the wide range of hardware/software embedded systems we want to model with SHIM.

IX. Conclusions and Future Work

We propose SHIM, a deterministic, concurrent model for embedded hardware/software systems that amounts to Kahn networks with rendezvous-style communication. We presented Tiny-SHIM, a simple language for realizing such systems and its formal semantics, a motivating example illustrating how to model a real-world hardware/software system, techniques for modeling familiar constructs, and software and hardware implementations.

We are currently creating a hardware/software codesign environment around this model and plan to demonstrate a real-world system implemented with it. We envision at least four major components of this system: an extended SHIM language with, for example, a richer type system; a simulation environment that allows complete systems to be debugged before any hardware or target system is complete; a software synthesis system that takes hints about how to implement certain processes (e.g., "make this an interrupt service routine") and generates C code for various real-time operating system environments; and a hardware synthesis system that can generate register-transfer level VHDL or Verilog. For both hardware and software synthesis, part of the challenge is ensuring that the native communication mechanisms (e.g., shared memory or inter-process communication in software; wires in hardware) correctly implement the SHIM communication model.

By design, timing is conspicuously absent from the SHIM model. Our philosophy is that functional verification should be separate from timing verification. The determinism of SHIM makes it possible to do this, just as for synchronous digital logic or sequential software programs, but in this paper we have only addressed functional aspects of our model.

A long-term goal is to add Giotto-like [34] timing constraints to SHIM and perform similar scheduling analysis. The much richer control behavior of SHIM systems makes this much more challenging than the equivalent in Giotto, however. We expect the analysis will be symbolic and fairly abstract to be practical; we plan to address this problem in the future.

In addition to mechanisms for optimizing performance (speeding simulation, generating faster hardware circuits), a SHIM-based development system will need mechanisms for static timing analysis. For hardware, the problem is fairly well-understood and it should be possible to adapt many existing techniques for use in our environment. Timing analysis of software is much less mature. A long-term goal of SHIM is to bring some of the discipline of concurrent hardware design to software; migrating static timing analysis will be an important part of this effort.

Another idea, suggested by a reviewer, is to develop algorithms for determining buffer sizes. Doing this in general (e.g., asking whether the system will deadlock and whether introducing additional buffering could prevent it) is probably too costly (it is at least as hard as state-space exploration). Instead, we suspect that the problem is tractable and interesting for certain classes of SHIM systems, such as feed-forward networks or those in the synchronous dataflow model, so we plan to pursue this question.

Like Kahn, we make two assumptions about communication and processes in our model: that all channels are one-to-one and that a process may only block on a single channel at a time. While simple and convenient for implementation, these restrictions are stronger than necessary to guarantee determinism. Non-compositionality is one unfortunate consequence of these assumptions: the behavior of a group of parallel process cannot be expressed as a process because of the single-blocking-channel rule. In a forthcoming publication, we will relax these requirements and provide a more expressive model that remains deterministic.

We envision SHIM becoming the standard for developing both the software and hardware in wide class of embedded systems. We believe the discipline and simplicity of the underlying model will be a key enabler for raising the level of abstraction available to designers.

Acknowledgements

Edwards and his group are supported by an NSF CAREER award, a grant from Intel corporation, an award from the SRC, and from New York State's NYSTAR program.

REFERENCES

[1] G. Kahn, "The semantics of a simple language for parallel programming," in *Information Processing 74: Proceedings of IFIP Congress 74*. Stockholm, Sweden: North-Holland, Aug. 1974, pp. 471–475.

[2] C. A. R. Hoare, *Communicating Sequential Processes*. Upper Saddle River, New Jersey: Prentice Hall, 1985.

[3] T. Murata, "Petri nets: Properties, analysis, and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, Apr. 1989.

[4] J. T. Buck, "Scheduling dynamic dataflow graphs with bounded memory using the token flow model," Ph.D. dissertation, University of California, Berkeley, 1993, available as UCB/ERL M93/69.

[5] T. M. Parks, "Bounded scheduling of process networks," Ph.D. dissertation, University of California, Berkeley, 1995, available as UCB/ERL M95/105. [Online]. Available: http://ptolemy.eecs.berkeley.edu/

[6] G. D. Plotkin, "A structural approach to operational semantics," Aarhus University, Åarhus, Denmark, Tech. Rep. DAIMI FN-19, 1981. [Online]. Available: http://www.dcs.ed.ac.uk/home/gdp/

[7] G. Berry and G. Gonthier, "The Esterel synchronous programming language: Design, semantics, implementation," *Science of Computer Programming*, vol. 19, no. 2, pp. 87–152, Nov. 1992.

[8] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice, "LUSTRE: A declarative language for programming synchronous systems," in *ACM Symposium on Principles of Programming Languages (POPL)*. Munich: Association for Computing Machinery, Jan. 1987.

[9] G. Berry and E. Sentovich, "An implementation of constructive synchronous programs in POLIS," *Formal Methods in System Design*, vol. 17, no. 2, pp. 165–191, Oct. 2000.

[10] J. Zeng, C. Soviani, and S. A. Edwards, "Generating fast code from concurrent program dependence graphs," in *Proceedings of Languages, Compilers, and Tools for Embedded Systems (LCTES)*, Washington, DC, June 2004, pp. 175–181.

[11] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, Sept. 1987.

[12] B. Lin, "Efficient compilation of process-based concurrent programs without run-time scheduling," in *Proceedings of Design, Automation, and Test in Europe (DATE)*, Paris, France, Feb. 1998, pp. 211–217.

[13] X. Zhu and B. Lin, "Compositional software synthesis of communicating processes," in *Proceedings of the IEEE International Conference on Computer Design (ICCD)*, Austin, Texas, Oct. 1999, pp. 646–651. [Online]. Available: http://www.computer.org/proceedings/iccd/0406/04060646abs.htm

[14] J. Cortadella, A. Kondratyev, L. Lavagno, M. Massot, S. Moral, C. Passerone, Y. Watanabe, and A. Sangiovanni-Vincentelli, "Task generation and compile-time scheduling for mixed data-control embedded software," in *Proceedings of the 37th Design Automation Conference*. Los Angeles, California: Association for Computing Machinery, 2000, pp. 489–494.

[15] S. S. Bhattacharyya, R. Leupers, and P. Marwedel, "Software synthesis and code generation for signal processing systems," *IEEE Transactions on Circuits and Systems—II: Analog and Digital Signal Processing*, vol. 47, no. 9, pp. 849–875, Sept. 2000.

[16] S. Liao, S. Tjiang, and R. Gupta, "An efficient implementation of reactivity for modeling hardware in the Scenic design environment," in *Proceedings of the 34th Design Automation Conference*, Anaheim, California, June 1997.

[17] G. Berry, "Esterel on hardware," *Philosophical Transactions of the Royal Society of London. Series A*, vol. 339, pp. 87–103, Apr. 1992, issue 1652, Mechanized Reasoning and Hardware Design.

[18] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 4, pp. 451–490, Oct. 1991.

[19] S. A. Edwards, T. Ma, and R. Damiano, "Using a hardware model checker to verify software," in *Proceedings of the 4th International Conference on ASIC (ASICON)*, Shanghai, China, Oct. 2001.

[20] R. M. Karp and R. E. Miller, "Properties of a model for parallel computations: Determinacy, termination, and queueing," *SIAM Journal on Applied Mathematics*, vol. 14, no. 6, pp. 1390–1411, Nov. 1966.

[21] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Transactions on Computers*, vol. C-36, no. 1, pp. 24–35, Jan. 1987.

[22] K. van Berkel, *Handshake Circuits: An Asynchronous Architecture for VLSI Programming*. Cambridge University Press, 1993.

[23] L. Janin, A. Bardsley, and D. A. Edwards, "Simulation and analysis of synthesised asynchronous circuits," *International Journal of Simulation Systems, Science & Technology*, vol. 4, no. 3–4, pp. 31–43, 2003.

[24] M. B. Josephs, "Receptive process theory," *Acta Informatica*, vol. 29, no. 1, pp. 17–31, Feb. 1992.

[25] ——, "An analysis of determinacy using a trace-theoretic model of asynchronous circuits," in *Proceedings of the Ninth International Symposium on Asynchronous Circuits and Systems (ASYNC)*, Vancouver, BC, Canada, May 2003, pp. 121–130.

[26] S. A. Edwards and O. Tardieu, "Deterministic receptive processes are Kahn processes," in *Proceedings of the 3rd International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, Verona, Italy, July 2005.

[27] F. Balarin, P. Giusto, A. Jurecska, C. Passerone, E. Sentovich, B. Tabbara, M. Chiodo, H. Hsieh, L. Lavagno, A. Sangiovanni-Vincentelli, and K. Suzuki, *Hardware-Software Co-Design of Embedded Systems: The POLIS Approach*. Boston, Massachusetts: Kluwer, 1997.

[28] M. Chiodo, P. Giusto, A. Jurecska, L. Lavagno, H. Hsieh, and A. Sangiovanni-Vincentelli, "A formal specification model for hardware/software codesign," in *Proceeding of the International Workshop on Hardware-Software Codesign*, Cambridge, Massachusetts, Oct. 1993. [Online]. Available: http://www-cad.eecs.berkeley.edu/Respep/Research/hsc/abstract.html

[29] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone, "The synchronous languages 12 years later," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 64–83, Jan. 2003, invited.

[30] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire, "Programming real-time applications with SIGNAL," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1321–1336, Sept. 1991.

[31] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A framework for simulating and prototyping heterogeneous systems," *International Journal of Computer Simulation*, vol. 4, pp. 155–182, Apr. 1994. [Online]. Available: http://ptolemy.eecs.berkeley.edu/papers/JEurSim/index.html

[32] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli, "Metropolis: An integrated electronic system design environment," *IEEE Computer*, vol. 36, no. 4, pp. 45–52, Apr. 2003.

[33] F. Boussinot, "FairThreads: mixing cooperative and preemptive threads in C," INRIA, RR 5039, 2003.

[34] T. A. Henzinger, B. Horowitz, and C. M. Kirsch, "Giotto: a time-triggered language for embedded programming," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 84–99, Jan. 2003.

Stephen A. Edwards (S'93-M'97) received the B.S. degree in Electrical Engineering from the California Institute of Technology in 1992, and the M.S. and Ph.D degrees, also in Electrical Engineering, from the University of California, Berkeley in 1994 and 1997 respectively. He is currently an assistant professor in the Computer Science Department of Columbia University in New York, which he joined in 2001 after a three-year stint with Synopsys, Inc., in Mountain View, California. His research interests include embedded system design, domain-specific languages, and compilers.

Olivier Tardieu graduated from *École Polytechnique*, Paris in 1998 and *École des Mines*, Paris in 2001. He received a Ph.D in Computer Science from *INRIA*, Sophia Antipolis and *École des Mines*, Paris in 2004. He is currently a postdoctoral research scientist in the Computer Science Department of Columbia University in New York, which he joined in 2005. His research interests include programming language design, compilers, software safety, concurrency theory, and hardware synthesis.