SHIM: a deterministic model for heterogeneous embedded systems — Source link

Stephen A. Edwards, Olivier Tardieu

Institutions: Columbia University

Published on: 01 Aug 2006 - IEEE Transactions on Very Large Scale Integration Systems (IEEE)

Topics: Embedded software, Software system, Systems design, Hardware description language and Verilog

SHIM: A Deterministic Model for Heterogeneous Embedded Systems

Stephen A. Edwards[*]
Department of Computer Science
Columbia University, New York
sedwards@cs.columbia.edu

Olivier Tardieu
Department of Computer Science
Columbia University, New York
tardieu@cs.columbia.edu

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 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.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages

General Terms

Languages, Theory

Keywords

Hardware/software codesign, Deterministic model of computation, Software synthesis, Hardware synthesis

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

1. 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. The cycles required to execute each machine instruction on a modern processor is virtually 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. A truly 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." Such a weak guarantee 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.

A reviewer correctly noted that these systems' environments are often nondeterministic and raised the question whether determinism in the model was as important as we make it out to be. 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. For a deterministic system, 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 and give a proof of its determinism (Section 2) and present a small language ("Tiny-SHIM") for processes and give its formal semantics (Section 3). This enables us to discuss modeling a real-world hardware/software system in Section 4. Hardware and software synthesis is discussed in Sections 5 and 6. We compare SHIM to other models in Section 7, and discuss some specifi c modeling techniques in Section 8.

2. THE SHIM MODEL

A system in the SHIM model consists of concurrently-running sequential processes that communicate exclusively with rendezvous through fi xed, point-to-point communication channels. This is a restriction of Kahn's networks [18] that uses a style of communication inspired by Hoare's CSP [14].

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

Inter-process communication is synchronous in the sense that both sending and receiving processes must mutually agree on when data is to be transferred. In general, either the sender or receiver may try to communicate fi rst and will wait for the other. The topology of communication channels (and the number and types of processes) is fi xed, and each communication channel connects a sending process with a 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 fi ts 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. □

2.1 Rationale

Above, we discussed our rationale for wanting determinism: it greatly simplifi es system validation using either simulation or formal techniques. Thus, we felt that nondeterministic models such as CSP [14] or Petri nets [24] were unsatisfactory.

$e ::= L \mid V \mid op\ e \mid e\ op\ e \mid$ `(` e `)`

$s ::= V$ `=` $e \mid$ `if (` e `)` s `else` $s \mid$ `while (` e `)` $s \mid s$ `;` s
$\mid$ `read(` C, V `)` $\mid$ `write(` C, e `)` $\mid$ `{ s }`

Figure 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.

We rejected Kahn's unbounded buffers because they make the model Turing-complete even for simple processes. Buck [9] showed that simple multiplexer-like processes in Kahn's model could be assembled to build a Turing machine.

The communication in our model is fi nite and does not introduce Turing-completeness. Specifi cally, our model is fi nite-state provided each process is fi nite-state. The advantages of this are legion: scheduling is much easier in our model because the synchronous communication restricts the number of choices. By defi nition, our models can always be simulated in fi nite memory; this question is undecidable for Kahn networks. Compare our simple scheduler, presented in Section 5, to the clever but costly one for Kahn networks by Parks [25], which dynamically detects buffer-overflow deadlock and increases buffer size in response.

Note that SHIM does not preclude buffered communication: it is easy to construct buffers by chaining multiple single-place-buffer processes. Such buffers are bounded, but modifying a design 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 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 typically used in 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 diffi cult-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.

3. 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 specifi c 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 a 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, i.e., an unordered list of potentially repeated process states, since several processes may be in identical states.

Most rules (i.e., the "→" rules) describe the operation of a single process, which operates independently except for communication. The last two rules describe the operation of the system as a whole (the "⇒" 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 $\mathcal{E}$ 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.

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

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

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

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

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.

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

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

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.

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

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

Sequencing requires two rules: one for when the first statement remains active, the other when the first statement terminates. Here, the statement being executed may or may not require a 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 ⊎ notation denotes the union of multisets.

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

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.

4. MOTIVATING EXAMPLE

Our choice of model comes from the observation of many embedded hardware/software systems. Here we describe one commercial embedded system and how to model it.

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 make the software operate synchronously with 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 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, its 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. Later, they verified that the software met its timing constraints (i.e., that each interrupt routine was able to complete its task before the next interrupt).

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).

[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/

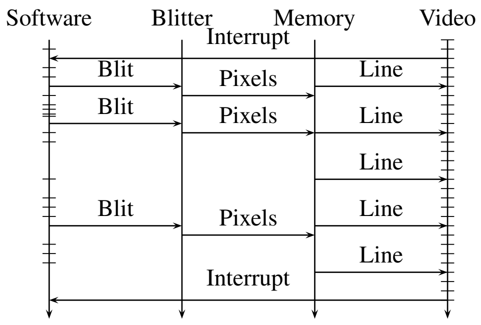

Figure 2: A message sequence chart illustrating the hardware/software interaction of Robby. Time runs from top to bottom, downward arrows indicate concurrently-running processes, and horizontal arrows indicate communication. Tick marks suggest the hardware and software clocks.

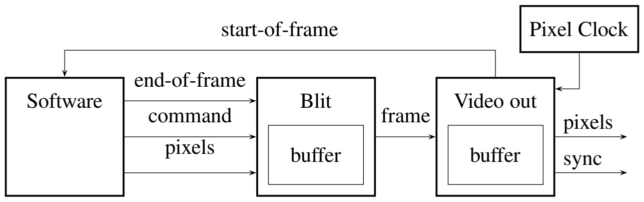

Figure 3: The software, blit, and video processes implementing a double-buffered display. At the end of each frame, the software signals the blitter memory to transfer its contents to the video display. The video system signals the software at the start of each frame.

4.1 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. To draw objects on the screen, the software occasionally invokes a hardware blitter that 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 current implementation of Robby, 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.

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.

```
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
```

Figure 4: Pseudocode for the blitter process.

```
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
```

Figure 5: Pseudocode for the software process.

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 achieve, 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 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.

5. A SOFTWARE IMPLEMENTATION

We describe a technique for implementing SHIM systems with single-threaded software. By design, this is not the only possible implementation, and it is certainly not the most efficient, but it illustrates how simple our models are to execute and points the way to more efficient techniques. For example, Lin and Zhu [23, 27] describe a more efficient quasi-static technique for our model that may produce exponentially-large code.

```
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
```

Figure 6: Pseudocode for the video process.

```
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 $p$
```

Figure 7: The software scheduling algorithm.

Our algorithm, Figure 7, 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, synchronizes with another process, or fails to do so and blocks, in any case passing the control back to the scheduler.

5.1 Fairness and Preemption

The algorithm in Figure 7 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 are often more effi cient; these 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 wish to consider them in more detail.

Two subclasses of systems are interesting: cooperative systems, which can be executed indefi nitely 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 infi nite 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 infi nite behavior.

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 infi nite number of communication take place must be connected, i.e., there cannot be two or more islands of processes with non-infi nite 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 defi nition, p must try to communicate infi nitely 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 infi nitely-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 a good thing since unfair, non-preemptive schedulers are usually more effi cient 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 infi nitely-running remainder of the system, an unfair scheduler would still work.

6. A HARDWARE IMPLEMENTATION

Here, we present a syntax-directed translation of Tiny-SHIM into synchronous digital hardware. 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 [4], 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 omits the preemption constructs of Esterel, but our translation deals with dataflow using static single-assignment analysis. We employ the algorithm of Cytron et al. [12] and construct a circuit using a technique like that of Edwards [13].

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 identifi es the data pathways, and fi nally the control-flow graph and datapath information is mechanically translated into gates.

Figure 8 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: a wire that passes control to the next statement in order. The complexity 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 type of multiplexer that selects between variables coming from its two incoming branches. Static single-assignment analysis determines which variables must be so chosen.

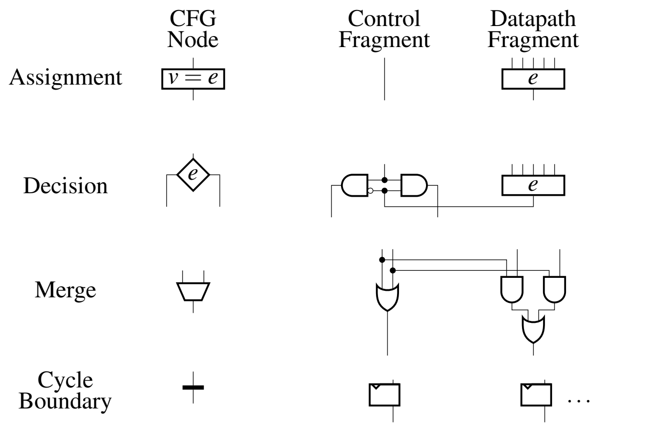

Figure 8: The four types of control-flow blocks and their hardware equivalents. The signal flow in the hardware schematic fragments follows the structure of the control-flow graph.

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 9 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 branches 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 unnecessary; in the future, we plan to devise an optimization that will eliminate these.

The template for communication is the richest: a pair of post-test loops each containing a cycle boundary. These boundaries force each communication action to take place at least one cycle after it is requested, thus ensuring that at most one communication takes place on each channel per cycle. This seemingly wasteful choice greatly simplifi es the logic: while it would be possible to construct circuitry that performs 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 fi rst *read* matched up with the fi rst *write* or the second *write*, the second read matched up with the fi rst *write* or the second *write* and so forth. We plan to consider such a translation in the future, 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 a another, different process. By construction, then, 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 10. 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 11 shows how the code of Figure 10 is translated into a control-flow graph using the templates from Figure 9, which can be

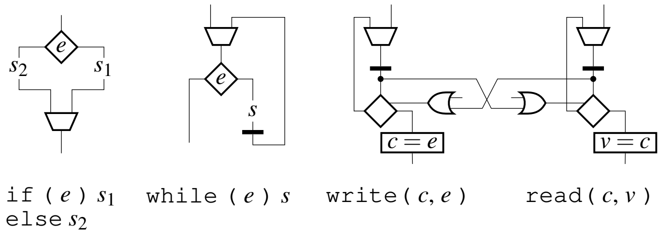

Figure 9: The translation of statements in our language.

```
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;
}
```

Figure 10: 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.

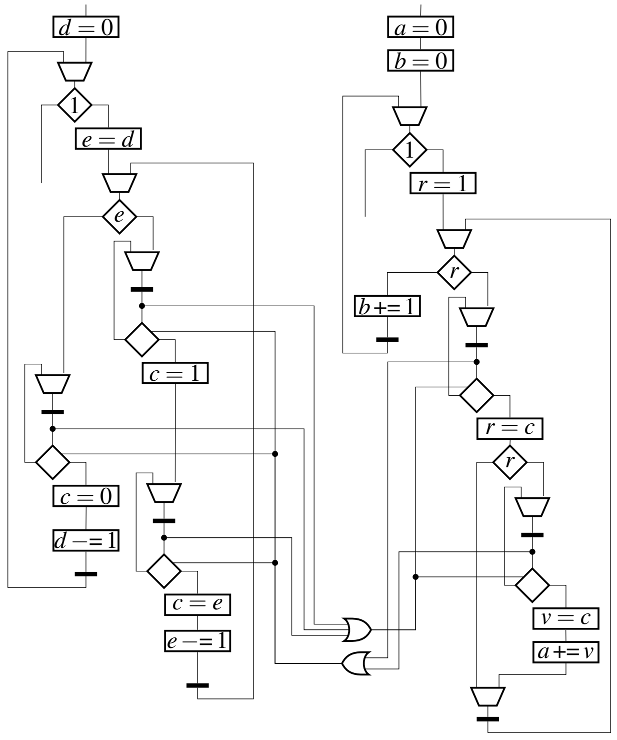

Figure 11: The translation of the processes in Figure 10.

translated into hardware using the templates of Figure 8. The two OR gates in the center of Figure 11 determine when the processes attempt to communicate.

The circuit implied by Figure 11 has a lot of redundancy and presents many opportunities for optimization. In addition to the usual Boolean simplifi cations, 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 [23] performs exhaustive analysis to determine communication patterns in a model much like ours, although he uses the result for software synthesis.

7. RELATIONSHIP TO OTHER MODELS

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

7.1 CSP

SHIM differs from Hoare's CSP [14] 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 is fundamental to guaranteeing determinism.

7.2 Kahn Networks

SHIM systems are deterministic for much the same reason as Kahn's [18], but are more restrictive. Kahn's processes communicate through unbounded buffers, which can be an advantage (our systems are subject to deadlock from buffers fi lling up; Kahn's are not) and a liability. Adding unbounded buffers makes Kahn networks Turing-complete and diffi cult to schedule since it is desirable to use bounded buffer memory wherever possible. Parks [25] scheduling algorithm does this, but it can be diffi cult to implement and (understandably) provides no a priori bounds on buffer sizes, a real liability for resource-constrained embedded systems.

Once buffer sizes are fi xed, a Kahn network can easily be translated into SHIM. Determining these sizes can be diffi cult 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 insuffi cient buffer space.

Like ours, other formalisms are restrictions of Kahn's networks. Karp and Miller's [19] and Lee and Messerschmitt's [22, 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 fi xed-sized buffers, these other models can be translated into SHIM with no loss of behavior.

7.3 Asynchronous Hardware Models

SHIM was inspired in part by van Berkel's asynchronous handshake circuits [26], which show among other things the practicality of implementing a fairly traditional imperative language with assignments, conditionals, and loops using nothing but rendezvous communication. Handshake circuits and the Balsa/Tangram (now called 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 make simulation substantially more diffi cult. Janin, Bardsley, and Edwards [15] describe a simulator that takes snapshots of the system at every nondeterministic (arbitrated) choice to allow the simulation to be restarted from these points.

Related to CSP, Josephs's Receptive Processes [16] 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 [17], but it remains at a level inappropriate for software.

The Polis project [1] had aims similar to ours. They proposed a unifying model of computation that could support both hardware and software implementations (CFSMs [11]) and constructed simulators and hardware and software synthesizers around it. Their model, however, is nondeterministic and its specifi cation of processes rather abstract, making it diffi cult to synthesize large pieces of software.

7.4 Synchronous Models

Synchronous models [3] 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 videogame described in Section 4 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 [10], Esterel [5], and Signal [20] can all handle multi-rate dataflow, but only Esterel really supports an imperative style of coding—natural for software—and unfortunately its support for multi-rate behavior is currently poor, despite a number of attempts. Berry and Sentovich's construction [6] show that the Esterel semantics can be implemented in an asynchronous model.

7.5 Heterogeneous Models

Projects such as Lee's Ptolemy [8] 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 never been automated implementations. SHIM could easily be implemented as a domain in Ptolemy.

The Metropolis project [2], a follow-on to Polis, tries to provide a structured environment for multiple models of computation. The system provides a meta-modeling language in which different models of computation, such as SHIM, can be specifi ed. Metropolis is therefore orthogonal to SHIM, and perhaps could be used as an implementation environment.

8. MODELING TECHNIQUES

8.1 Buffers

Although communication in SHIM is unbuffered, it is easy to create fi nite-size buffered communication channels. For a buffered channel of size *n*, introduce a chain of *n* single-place buffer processes 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.

8.2 Interrupts

We cannot directly model traditional software interrupts since they are nondeterministic in their full generality. In practice, however, interrupts are generally used in a deterministic way to emulate concurrency in software. Most interrupt handlers take great pains not to modify the state of the program they run above and usually perform little more than a simple buffering action.

As a result we suggest the effects of well-behaved interrupts be represented with a pair of processes: one for the program being interrupted, the other for the interrupt handler that communicates with the interrupt source and then the other process.

Using interrupts to implement our systems is an obvious possibility. Processes that buffer data coming from hardware processes and pass them to software are obvious candidates.

8.3 Pure Synchrony

Synchronous processes march to a common clock and communicate in a broadcast style. In SHIM, such systems can be modeled 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 in our model that repeatedly read from an input channel and replicate the data on every output channel.

8.4 Synchronous Dataflow

Lee and Messerschmitt's Synchronous Dataflow [22]—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.

Obvious future work includes trying to identify SDF-like subsystems in SHIM systems and applying some of the very sophisticated SDF scheduling techniques [7].

8.5 Timing Issues

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 the 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, this is more difficult for software because of its more unpredictable nature.

8.6 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.

8.7 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.

9. 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, software and hardware implementation techniques, and addressed a variety of modeling issues.

Our next step will be to create a hardware/software codesign environment around this model and 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.

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.

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, although work is progressing. One of the long-range goals of SHIM is to bring some of the discipline of hardware concurrency to software development, and this will be one of the important ways.

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.

In short, 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.

10. REFERENCES

[1] Felice Balarin, Paolo Giusto, Attila Jurecska, Claudio Passerone, Ellen Sentovich, Bassam Tabbara, Massimiliano Chiodo, Harry Hsieh, Luciano Lavagno, Alberto Sangiovanni-Vincentelli, and Kei Suzuki. *Hardware-Software Co-Design of Embedded Systems: The POLIS Approach*. Kluwer, Boston, Massachusetts, 1997.

[2] Felice Balarin, Yosinori Watanabe, Harry Hsieh, Luciano Lavagno, Claudio Passerone, and Alberto Sangiovanni-Vincentelli. Metropolis: An integrated electronic system design environment. *IEEE Computer*, 36(4):45–52, April 2003.

[3] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, January 2003. Invited.

[4] Gérard Berry. Esterel on hardware. *Philosophical Transactions of the Royal Society of London. Series A*, 339:87–103, April 1992. Issue 1652, Mechanized Reasoning and Hardware Design.

[5] Gérard Berry and Georges Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, November 1992.

[6] Gérard Berry and Ellen Sentovich. An implementation of constructive synchronous programs in POLIS. *Formal Methods in System Design*, 17(2):165–191, October 2000.

[7] Shuvra S. Bhattacharyya, Ranier Leupers, and Peter Marwedel. Software synthesis and code generation for signal processing systems. *IEEE Transactions on Circuits and Systems—II: Analog and Digital Signal Processing*, 47(9):849–875, September 2000.

[8] Joseph T. Buck, Soonhoi Ha, Edward A. Lee, and David G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *International Journal of Computer Simulation*, 4:155–182, April 1994.

[9] Joseph Tobin Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory using the Token Flow Model*. PhD thesis, University of California, Berkeley, 1993. Available as UCB/ERL M93/69.

[10] Paul Caspi, Daniel Pilaud, Nicholas Halbwachs, and J. A. Plaice. LUSTRE: A declarative language for programming synchronous systems. In *ACM Symposium on Principles of Programming Languages (POPL)*, Munich, January 1987. Association for Computing Machinery.

[11] Massimiliano Chiodo, Paolo Giusto, Attila Jurecska, Luciano Lavagno, Harry Hsieh, and Alberto Sangiovanni-Vincentelli. A formal specifi cation model for hardware/software codesign. In *Proceeding of the International Workshop on Hardware-Software Codesign*, Cambridge, Massachusetts, October 1993.

[12] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Effi ciently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[13] Stephen A. Edwards, Tony Ma, and Robert Damiano. Using a hardware model checker to verify software. In *Proceedings of the 4th International Conference on ASIC (ASICON)*, Shanghai, China, October 2001.

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

[15] Lilian Janin, Andrew Bardsley, and Doug A. Edwards. Simulation and analysis of synthesised asynchronous circuits. *International Journal of Simulation Systems, Science & Technology*, 4(3–4):31–43, 2003.

[16] Mark B. Josephs. Receptive process theory. *Acta Informatica*, 29(1):17–31, February 1992.

[17] Mark B. Josephs. 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)*, pages 121–130, Vancouver, BC, Canada, May 2003.

[18] Gilles Kahn. The semantics of a simple language for parallel programming. In *Information Processing 74: Proceedings of IFIP Congress 74*, pages 471–475, Stockholm, Sweden, August 1974. North-Holland.

[19] Richard M. Karp and Raymond E. Miller. Properties of a model for parallel computations: Determinacy, termination, and queueing. *SIAM Journal on Applied Mathematics*, 14(6):1390–1411, November 1966.

[20] Paul Le Guernic, Thierry Gautier, Michel Le Borgne, and Claude Le Maire. Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1336, September 1991.

[21] Edward A. Lee and David G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, C-36(1):24–35, January 1987.

[22] Edward A. Lee and David G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987.

[23] Bill Lin. Effi cient compilation of process-based concurrent programs without run-time scheduling. In *Proceedings of Design, Automation, and Test in Europe (DATE)*, pages 211–217, Paris, France, February 1998.

[24] Tadao Murata. Petri nets: Properties, analysis, and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[25] Thomas M. Parks. *Bounded Scheduling of Process Networks*. PhD thesis, University of California, Berkeley, 1995. Available as UCB/ERL M95/105.

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

[27] Xiaohan Zhu and Bill Lin. Compositional software synthesis of communicating processes. In *Proceedings of the IEEE International Conference on Computer Design (ICCD)*, pages 646–651, Austin, Texas, October 1999.