Concurrency and Communication: Lessons from the SHIM Project

Stephen A. Edwards

Columbia University, New York, NY, USA
sedwards@cs.columbia.edu

Abstract. Describing parallel hardware and software is difficult, especially in an embedded setting. Five years ago, we started the SHIM project to address this challenge by developing a programming language for hardware/software systems. The resulting language describes asynchronously running processes that has the useful property of scheduling-independence: the I/O of a SHIM program is not affected by any scheduling choices. This paper presents a history of the SHIM project with a focus on the key things we have learned along the way.

1 Introduction

SHIM, an acronym for "software/hardware integration medium," started as an attempt to simplify the challenges of passing data across the hardware-software boundary. It has since turned into a language development effort centered around a scheduling-independent (i.e., race-free) concurrency model and static analysis.

The purpose of this paper is to lay out the history of the SHIM project with a special focus on what we learned along the way. It is deliberately light on technical details (which can be found in the original publications) and instead tries to contribute intuition and insight.

We begin by discussing the original motivations for the project, how it evolved into a study of concurrency models, how we chose a particular model, and how we have added language features to that model. We conclude with a section highlighting the central lessons we have learned along with open problems.

2 Embryonic SHIM

We started developing SHIM in 2004 after observing the difficulties our students were having building embedded systems [1,2] that communicated across the hardware/software boundary. The central idea was to provide variables that could be accessed equally easily by either hardware processes or software functions, both written in C-like dialect. Figure 1 shows a simple counter in this dialect of the language. The *count* function resides in hardware; the other two are in software. When a software function like *get_time* would reads the hardware register *counter*, the compiler would automatically insert code that would fetch its value from the hardware and synthesize VHDL that could send the data

```
module timer {
  shared uint:32 counter; /* Hardware register visible from software */

  hw void count() { /* Hardware function */
    counter = counter + 1; /* Direct access to hardware register */
  }

  out void reset_timer() { /* Software function */
    counter = 0; /* Accesses register through bus */
  }

  out uint get_time() { /* Software function */
    return counter; /* Accesses register through bus */
  }
}
```

Fig. 1. An early fragment of SHIM [1]

on a bus when requested. We wrote a few variants of an I^{2}C bus controller in the language, starting with an all-software version and ending with one that implemented byte-level communication completely in hardware.

The central lesson of this work was that the shared memory model, while simple, was a very low-level way to implement such communication. Invariably, it is necessary to layer over it another communication protocol (e.g., some form of handshake) to ensure coherence. We had not included an effective mechanism for implementing communication libraries that could hide this fussy code, so it was back to the drawing board.

3   Kahn, Hoare, and the SHIM Model

We decided we wanted reliable communication, including any across the hardware/software boundary, to be a centerpiece of the next version of SHIM. Erroneous communication is a central source of bugs in hardware designs: our embedded system students' favorite mistake was to generate a value in one cycle and read it in another. This rarely produces even a warning in usual hardware simulation, so it can easily go unnoticed.

We also found the inherent nondeterminism of the first iteration of SHIM a key drawback. The speed at which software runs on processors is rarely known, let alone controlled. Since software and hardware run in parallel and communicate using shared variables, the resulting system was nondeterministic, making it difficult to test. It also ran counter to what we had learned from Esterel [3].

Table 1 shows our wishlist. We wanted a concurrent, deterministic (i.e., independent of scheduling) model of computation and started looking around. The synchronous model [4] was unsuitable because it generally assumes either a single or harmonically related clocks and would not work well with software.

Table 1. The SHIM Wishlist

Trait	Motivation
Concurrent	Hardware/software systems fundamentally parallel
Mixes synchronous and asynchronous styles	Software slower and less predictable than hardware; need something like multirate dataflow
Only requires bounded resources	Fundamental restriction on hardware
Formal semantics	No arguments about meaning or behavior
Scheduling-independent	I/O should not depend on program implementation

Steve Nowick steered us toward the body of work on delay-independent circuits (e.g., van Berkel's Handshake circuits [5]). We compared this class of processes to Kahn's networks [6] and found them to be essentially the same [7]. We studied how to characterize such processes [8], finding that we could characterize them as functions that, when presented with more inputs or output opportunities, never produced less or different data.

In their classic form, the unbounded buffers of Kahn networks actually make them Turing-complete [9] and difficult to schedule [10], so we decided on a model in which Kahn networks were restricted to CSP-like rendezvous [11]. Others, such as Lin [12] had also proposed using such a model.

In 2005, we presented our new SHIM model and a skeleton of the language, "Tiny-SHIM," and its formal semantics [13]. It amounted to *read* and *write* operations sewn together with the usual C-like expressions and control-flow statements. We later extended this work with further examples, a single-threaded C implementation, and an outline of a hardware translation [14].

In 2006, we published our first real research result with SHIM: a technique for very efficient single-threaded code generation [15]. The centerpiece of this work was an algorithm that could compile arbitrary groups of processes into a single automaton whose states abstracted the control states of the processes. Our goal was to eliminate synchronization overhead, so the automaton captured which processes were waiting on which channels, but left all other details, such as variable values and details of the program counters, to the automaton.

Figure 2 demonstrates the algorithm from Edwards and Tardieu [15]. The automaton's states are labeled with a number (e.g., S0), the state of each channel in the system (ready "-", blocked reading "R", or blocked writing "W"), and, for each process, whether it is runnable ($\sqrt{}$) or blocked on a channel ($\times$), and a set of possible program counters. From each state, the automaton generator (a.k.a., the scheduler) nondeterministically chooses one of the runnable processes to execute and generates a state by considering each possible PC value for the process. The code generated for a state with multiple PC values begins with a C *switch* statement that splits control depending on the PC value.
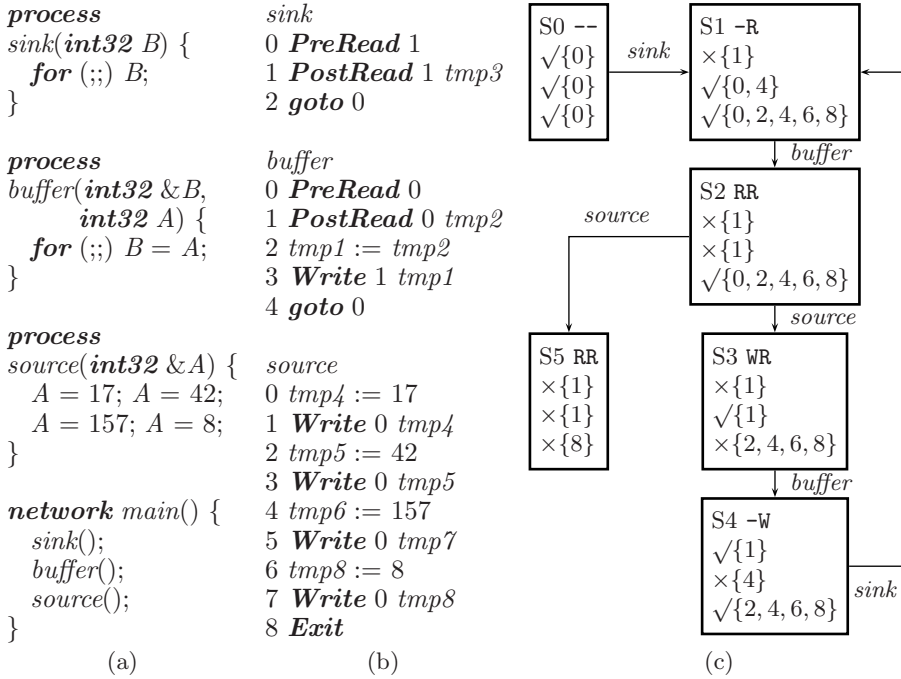
process
$sink(\textbf{\textit{int32}}\ B)$ {
  for (;;) B;
}

process
$buffer(\textbf{\textit{int32}}\ \&B,$
        $\textbf{\textit{int32}}\ A)$ {
  for (;;) $B = A$;
}

process
$source(\textbf{\textit{int32}}\ \&A)$ {
  $A = 17;\ A = 42;$
  $A = 157;\ A = 8;$
}

network $main()$ {
  $sink();$
  $buffer();$
  $source();$
}

(a)

sink
0 **PreRead** 1
1 **PostRead** 1 *tmp3*
2 **goto** 0

buffer
0 **PreRead** 0
1 **PostRead** 0 *tmp2*
2 *tmp1 := tmp2*
3 **Write** 1 *tmp1*
4 **goto** 0

source
0 *tmp4 := 17*
1 **Write** 0 *tmp4*
2 *tmp5 := 42*
3 **Write** 0 *tmp5*
4 *tmp6 := 157*
5 **Write** 0 *tmp7*
6 *tmp8 := 8*
7 **Write** 0 *tmp8*
8 **Exit**

(b)

```
S0 --          sink      S1 -R
√{0}     ────────────▶   ×{1}
√{0}                     √{0,4}
√{0}                     √{0,2,6,8}
                              │ buffer
                              ▼
                          S2 RR
        source            ×{1}
                          ×{1}
                          √{0,2,4,6,8}
                              │ source
                              ▼
S5 RR                     S3 WR
×{1}                      ×{1}
×{1}                      √{1}
×{8}                      ×{2,4,6,8}
                              │ buffer
                              ▼
                          S4 -W
                          √{1}
                          ×{4}          sink
                          √{2,4,6,8}
```

(c)

Fig. 2. An illustration of the SHIM language and its automaton compilation scheme from Edwards and Tardieu [15]. A source program (a) is dismantled into intermediate code (b), then simulated to produce an automaton (c). Each state is labeled with its name, the state of each channel (blocked on read, blocked on write, or idle), and the state of each process (runnable, and possible program counter values).

At this point, the language fairly closely resembled the Tiny-SHIM language of the Emsoft paper [13]. A system consisted of a collection of sequential processes, assumed to all start when the system began. It could also contain *networks*—groups of connected processes that could be instantiated hierarchically.

One novel feature of this version, which we later dropped, was the ability to instantiate processes and networks without supply explicit connections. Instead, the compiler would examine the interface to each instantiated process and make sure its environment supplied such a signal. Connections were made implicitly by name, although this could be overridden. This feature arose from observing how in VHDL it is often necessary to declare and mention each channel many times: once for each process, once for each instantiation of each process, and once in the environment in which it is instantiated.

However, in the process of writing more elaborate test cases, such as a JPEG decoder [16], we decided that this connection-centric specification style (which we adopted from hardware description languages) was inadequate for any sort of interesting software. We wanted function calls.

4   Recursion

In 2006, we introduced function calls and recursion to SHIM, making it very C-like [17]. Our main goal was to make basic function calls work, allowing the usual re-use of code, but we also found that recursion, especially bounded recursion, was a useful mechanism for specifying more complex structures.

```
void buffer( int i, int &o) {        void fifo(int i, int &o, int n) {
   for (;;) {                           int c; int m = n − 1;
      recv i;                           if (m)
      o = i;                              buffer(i, c) par fifo(c, o, m);
      send o;                          else
   }                                     buffer(i, o);
}                                     }
```

Fig. 3. An *n*-place FIFO specified using recursion, from Tardieu and Edwards [17]

Figure 3 illustrates this style. The recursive *fifo* procedure calls itself repeatedly in parallel, effectively instantiating *buffer* processes as it goes. This recursion runs only once, when the program starts, to set up a chain of single-place buffers.

5   Exceptions

Next, we added exceptions [18], certainly the most technically difficult addition we have made. Inspired by Esterel [3], where exceptions are used not just for occasional error handling but as widely as, say, if-then-else, we wanted our exceptions to be widely applicable and be concurrent and scheduling-independent.

For sequential code, the semantics of exceptions were clear: throwing an exception immediately sends control to the most-recently-entered handler for the given exception, terminating any functions that were called in between.

For concurrently running functions, the right behavior was less obvious. We wanted to terminate everything leading up to the handler, including any concurrently running relatives, but we insisted on maintaining SHIM's scheduling independence, meaning we had to carefully time when the effect of an exception was felt. Simply terminating siblings when one called an exception would be nondeterministic: the behavior would then depend on the relative execution rates of of the processes and thus not be scheduling independent.

Our solution was to piggyback the exception mechanism on the communication system, i.e., a process would only learn of an exception when it attempted to communicate, the only point at which processes agree on the time.

To accommodate exceptions, we introduced a new, "poisoned," state for a process that represents when it has been terminated by an exception and is waiting for its relatives to terminate. Any process that attempts to communicate with a poisoned process will itself become poisoned. In Figure 5, the first thread throws

```
void main() {                          void main() {
   int i; i = 0;                           int i; i = 0;
   try {                                   try { // thread 1
      i = 1;                                  throw T;
      throw T;                             } par { // thread 2
      i = i * 2; // is not executed           for (;;) i = i + 1; // runs forever
   } catch(T) { i = i * 3; } // i = 3    } catch(T) {}
}                                        }
```

(a)   (b)

Fig. 4. (a) Sequential exception semantics are classical. (b) Thread 2 never feels the effect of the exception because it never communicates. From Tardieu and Edwards [18].

```
void main() {
   chan int i = 0, j = 0;
   try { // thread 1
      while (i < 5) next i = i + 1;
      throw T; // poisons itself
   } par { // thread 2
      for (;;) next j = next i + 1; // poisoned by thread 1
   } par { // thread 3
      for (;;) recv j; // poisoned by thread 2
   } catch (T) {}
}
```

Fig. 5. Transitive Poisoning: *throw T* poisons the first process, which poisons the second when the second attempts *next i*. Finally the third is poisoned when it attempts *recv j* and the whole group terminates.

an exception; the second thread is poisoned when it attempts to rendezvous on i, and the third is poisoned by the second when it attempts to rendezvous on j.

The idea was simple enough, and the interface it presented to the programmer could certainly be used and explained without much difficulty, but implementing it turned out to be a huge challenge, despite there being fairly simple set of structural operational semantics rules for it.

The real complexity came from having to consider exception scope, which limits how far the poison propagates (it does not propagate outside the scope of the exception) and the behavior of multiple, concurrently thrown exceptions.

6   Static Analysis

SHIM has always been designed for aggressive compiler analysis. We have attempted to keep its semantics simple, scheduling-independent, and restrict it to finite-state models. Together, these have made it easier to analyze.

We developed a technique for removing bounded recursion from SHIM programs [19]. One goal was to simplify SHIM's translation into hardware, where general recursion would require memory for a stack and choosing a size for it, but it has found many other uses. In particular, if a program has only bounded recursion, it is finite-state, simplifying other analysis steps.

The basic idea of our work was to unroll recursive calls by exactly tracking the behavior of variables that control the recursion. Our insight was to observe that for a recursive function to terminate, the recursive call must be within the scope of a conditional. Therefore, we need to track the predicate of this conditional, see what can affect it, and so forth.

Figure 6 illustrates what this procedure does to a simple FIFO. To produce the static version in Figure 6(b), our procedure observes that the n variable controls the predicate around *fifo*'s recursive call of itself. It then notices that n is initially bound to 3 by *fifo3* and generates three specialized versions of *fifo*—one with $n = 3$, $n = 2$, and $n = 1$—simplifies each, then inlines each function, since each is only called once.

Of course, in the worst case our procedure could end up trying to track every variable in the program, which would be impractical, but in many examples we tried, recursion control only involved a few variables, making it easy to resolve.

A key hypothesis of the SHIM project has been that scheduling independence should be a property of any practical concurrent language because it greatly simplifies reasoning about a program, both by the programmer and by automated tools. Our work on static deadlock detection reinforces this key point.

SHIM is not immune to deadlocks (e.g., { recv a; recv b; } par { send b; send a; } is about the simplest example), but they are simpler in SHIM because of its scheduling-independence. Deadlocks in SHIM cannot occur because of race conditions. For example, because SHIM does not have races, there are no race-

```
void fifo3(chan int i, chan int &o) {
  fifo(i, o, 3);
}

void fifo(chan int i, chan int &o,
          int n) {
  if (n > 1) {
    chan int c;
    buf(i, c); par fifo(c, o, n−1);
  } else buf(i, o);
}

void buf(chan int i, chan in &o) {
  for (;;) next o = next i;
}
```

```
void fifo3(chan int i,
           chan int &o) {
  chan int c1, c2, c3;
  buf(i, c1);
  par
  buf(c1, c2);
  par
  buf(c2, o);
}

void buf(chan int i, chan in &o) {
  for (;;) next o = next i;
}
```

(a)   (b)

Fig. 6. Removing bounded recursion, controlled by the n variable, from (a) gives (b). After Edwards and Zeng [19].

induced deadlocks, such as the "grab locks in opposite order" deadlock race present in many other languages.

In general, SHIM does not need to be analyzed under an interleaved model of concurrency since most properties, including deadlock, are the same under any schedule. So all the clever partial order tricks used by model checkers such as SPIN [20], are not necessary for SHIM.

We first used the synchronous model checker NuSMV [21] to detect deadlocks in SHIM [22]—an interesting choice since SHIM's concurrency model is fundamentally asynchronous. Our approach was to abstract away data operations and choose a specific schedule in which each communication event takes a single cycle. This reduced the SHIM program to a set of communicating state machines suitable for the NuSMV model checker.

We continue to work on deadlock detection in SHIM. Most recently [23], we took a compositional approach where we build an automaton for a complete system piece by piece. Our insight is that we can usually abstract away internal channels and simplify the automaton without introducing or avoiding deadlocks. The result is that even though we are doing explicit model-checking, we can often do it much faster than a state-of-the art symbolic model checker such as NuSMV.

We have also used model checking to search for situations where buffer memory can be shared [24]. In general, each communication channel needs storage for any data being communicated over it, but in certain cases, it is possible to prove that two channels can never be active simultaneously. We use the NuSMV model checker to identify these cases, which allow us to share potentially large buffers across multiple channels. Because this is optimization, if the model checker becomes overloaded, we can we safely analyze the system in smaller pieces.

7 Backends

We have developed a series of backends for the SHIM compiler; each works off a slightly different intermediate representations.

First, we developed a code generator that produced single-threaded C [14] for a variant of Tiny-SHIM, which had only point-to-point channels. The runtime system maintained a linked list of runnable processes, and for each channel, tracked what process, if any, was blocked on it. Each process was compiled into a separate C function, which stored its state as a global integer and used an *switch* statement to restore it. This worked well, although we could improve runtimes by compiling away communication overhead through static scheduling [15].

To handle multi-way rendezvous, exceptions, and recursion on parallel hardware we needed a new technique. Our next backend [25] generated C code that made calls to the POSIX thread library to ask for parallelism. The challenge was to minimize overhead. Each communication action would acquire the lock on a channel, check whether every process connected to it had also blocked (i.e., whether the rendezvous could occur), and then check if the channel was connected to a poisoned process (i.e., a relevant exception had been thrown). All of these checks ran quickly; actual communication and exceptions took longer.

We also developed a backend for IBM's CELL processor [26]. A direct offshoot of the pthreads backend, it allows the user to assign computationally intensive tasks to the CELL's synergistic processing units (SPUs); remaining tasks run on the CELL's PowerPC core (PPU). Our technique replaces the offloaded functions with wrappers that communicate across the PPU-SPU boundary. Cross-boundary function calls are technically challenging because of data alignment restrictions on function arguments, which we would have preferred to be stack-resident. This, and many other fussy aspects of coding for the CELL, convinced us that such heterogeneous multicore processors demand languages at a higher level than C.

8   Lessons and Open Problems

8.1   Function Calls

Early version of the language did not support classical software-like function calls. However, these are extremely useful, even in dataflow-centric descriptions, that they really need to be part of just about any language. We were initially deceived by the rare use of function calls in VHDL and Verilog, but we suspect this is because they do not fit easily into the register-transfer model.

8.2   Two-Way vs. Multi-way Rendezvous

Initial versions of SHIM only used two-way rendezvous, but after a discussion with Edward Lee, we became convinced that multi-way rendezvous was useful to provide at the language level. Debugging was one motivation: with multiway rendezvous, it becomes easy to add a monitor that can observe data flowing through a channel; modeling the clock of a synchronous system was another.

Unfortunately, implementing multiway rendezvous is much more complicated than implementing two-way rendezvous, yet we found that most communication in SHIM programs is point-to-point, so we are left with a painful choice: slow down the common case to accommodate the uncommon case, or do aggressive analysis to determine when we can assume point-to-point communication.

We would like to return SHIM to point-to-point communication only but provide multiway rendezvous as a sort of syntactic sugar, e.g., by introducing extra processes responsible for communication on channels. How to do this correctly and elegantly remains an open question, unfortunately.

8.3   Exceptions

Exceptions have been an even more painful feature than multi-way rendezvous. They are extremely convenient from a programming standpoint (e.g., SHIM's rudimentary I/O library wraps each program in an exception to allow it to terminate gracefully; virtually every compiler testcase includes at least a single exception), but extremely difficult to both implement and reason about.

We have backed away from exceptions for now (all our recent work addresses the exception-free version of SHIM); we see two possibilities for how to proceed.

One is to restrict the use of exceptions so that the complicated case of multiple, concurrent exceptions is simply prohibited. This may prohibit some interesting algorithms, but should greatly simplify the implementation, and probably also analysis, of exceptions.

Another alternative is to turn exceptions into syntactic sugar layered on the exception-free SHIM model. We always had this in the back our minds: an exception would just put a process into an unusual state where it would communicate its poisoned state to any process that attempts to communicate with it. The problem is that the complexity tends to grow quickly when multiple, concurrent exceptions and scopes are considered. Again, exactly how to translate exceptions into a simpler SHIM model remains an open question.

8.4   Semantics and Static Analysis

We feel we have proven one central hypothesis of the SHIM project: that simple, deterministic semantics helps both programming and automated program analysis. That we have been able to devise truly effective mechanisms for clever code generation (e.g., static scheduling) and analysis (e.g., deadlock detection) that can gain deep insight into the behavior of programs vindicates this view. The bottom line: if a programming language does not have simple semantics, it is really hard to analyze its programs quickly or precisely.

We have also validated the utility of scheduling independence. Our test suite, which consists of many parallel programs, has reproducible results that lets us sleep at night. We have found few cases where the approach has limited us.

Algorithms where there is a large number of little, variable-sized, but independent pieces of work to be done do not mesh well with SHIM's scheduling-independent philosophy as it currently stands. The obvious way to handle this is to maintain a bucket of tasks and assign each task to a processor once it has finished its last task. The order in which the tasks is performed, therefore, depends on their relative execution rates, but this does not matter if the tasks are independent. It would be possible to add scheduling-independent task distribution and scheduling to SHIM (i.e., provided the tasks are truly independent or, equivalently, confluent); exactly how is an open research question.

8.5   Buffers

That buffering is mandatory for high-performance parallel applications is hardly a revelation; we confirmed it anyway. The SHIM model has always been able to implement FIFO buffers (e.g., Figure 3), but we have realized that they are sufficiently fundamental to be a first-class type in the language. We are currently working on a variant of the language that replaces pure rendezvous communication with bounded, buffered communication. Because it will be part of the language, it will be easier to map to unusual environments, such as the DMA mechanism for inter-core communication on the CELL processor.

8.6   Other Applications

The most likely future role of SHIM will be as inspiration for other languages. For example, Vasudevan has ported its communication model into the Haskell functional language [27] and proposed a compiler that would impose its scheduling-independent view of the work on arbitrary programs [28]. Certain SHIM ideas, such as scheduling analysis [29], have also been used in IBM's X10 language.

Acknowledgments

References

1. Edwards, S.A.: Experiences teaching an FPGA-based embedded systems class. In: Proceedings of the Workshop on Embedded Systems Education (WESE), Jersey City, New Jersey, September 2005, pp. 52–58 (2005)
2. Edwards, S.A.: SHIM: A language for hardware/software integration. In: Proceedings of SYNCHRON, Schloss Dagstuhl, Germany (December 2004)
3. Berry, G., Gonthier, G.: The Esterel synchronous programming language: Design, semantics, implementation. Science of Computer Programming 19(2), 87–152 (1992)
4. Benveniste, A., Caspi, P., Edwards, S.A., Halbwachs, N., Guernic, P.L., de Simone, R.: The synchronous languages 12 years later. Proceedings of the IEEE 91(1), 64–83 (2003)
5. van Berkel, K.: Handshake Circuits: An Asynchronous Architecture for VLSI Programming. Cambridge University Press, Cambridge (1993)
6. Kahn, G.: The semantics of a simple language for parallel programming. In: Information Processing 74: Proceedings of IFIP Congress 74, Stockholm, Sweden, pp. 471–475. North-Holland, Amsterdam (1974)
7. Edwards, S.A., Tardieu, O.: Deterministic receptive processes are Kahn processes. In: Proceedings of the International Conference on Formal Methods and Models for Codesign (MEMOCODE), Verona, Italy, July 2005, pp. 37–44 (2005)
8. Tardieu, O., Edwards, S.A.: Specifying confluent processes. Technical Report cucs–037–06, Columbia University, Department of Computer Science, New York, USA (September 2006)
9. Buck, J.T.: 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. Parks, T.M.: Bounded Scheduling of Process Networks. PhD thesis, University of California, Berkeley (1995); Available as UCB/ERL M95/105
11. Hoare, C.A.R.: Communicating sequential processes. Communications of the ACM 21(8), 666–677 (1978)
12. Lin, B.: Software synthesis of process-based concurrent programs. In: Proceedings of the 35th Design Automation Conference, San Francisco, California, June 1998, pp. 502–505 (1998)

13. Edwards, S.A., Tardieu, O.: SHIM: A deterministic model for heterogeneous embedded systems. In: Proceedings of the International Conference on Embedded Software (Emsoft), Jersey City, New Jersey, September 2005, pp. 37–44 (2005)
14. Edwards, S.A., Tardieu, O.: SHIM: A deterministic model for heterogeneous embedded systems. IEEE Transactions on Very Large Scale Integration (VLSI) Systems 14(8), 854–867 (2006)
15. Edwards, S.A., Tardieu, O.: Efficient code generation from SHIM models. In: Proceedings of Languages, Compilers, and Tools for Embedded Systems (LCTES), Ottawa, Canada, June 2006, pp. 125–134 (2006)
16. Vasudevan, N., Edwards, S.A.: A jpeg decoder in SHIM. Technical Report cucs–048–06, Columbia University, Department of Computer Science, New York, USA (December 2006)
17. Tardieu, O., Edwards, S.A.: R-SHIM: Deterministic concurrency with recursion and shared variables. In: Proceedings of the International Conference on Formal Methods and Models for Codesign (MEMOCODE), Napa, California, July 2006, p. 202 (2006)
18. Tardieu, O., Edwards, S.A.: Scheduling-independent threads and exceptions in SHIM. In: Proceedings of the International Conference on Embedded Software (Emsoft), Seoul, Korea, October 2006, pp. 142–151 (2006)
19. Edwards, S.A., Zeng, J.: Static elaboration of recursion for concurrent software. In: Proceedings of the Workshop on Partial Evaluation and Program Manipulation (PEPM), San Francisco, California, January 2008, pp. 71–80 (2008)
20. Holzmann, G.J.: The model checker SPIN. IEEE Transactions on Software Engineering 23(5), 279–294 (1997)
21. Cimatti, A., Clarke, E.M., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An openSource tool for symbolic model checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 359–364. Springer, Heidelberg (2002)
22. Vasudevan, N., Edwards, S.A.: Static deadlock detection for the SCHIM concurrent language. In: Proceedings of the International Conference on Formal Methods and Models for Codesign (MEMOCODE), Anaheim, California, June 2008, pp. 49–58 (2008)
23. Shao, B., Vasudevan, N., Edwards, S.: Compositional deadlock detection for rendezvous communication. In: Proceedings of the International Conference on Embedded Software (Emsoft), Grenoble, France (October 2009)
24. Vasudevan, N., Edwards, S.A.: Buffer sharing in CSP-like programs. In: Proceedings of the International Conference on Formal Methods and Models for Codesign (MEMOCODE), Cambridge, Massachusetts (July 2009)
25. Edwards, S.A., Vasudevan, N., Tardieu, O.: Programming shared memory multiprocessors with deterministic message-passing concurrency: Compiling SHIM to Pthreads. In: Proceedings of Design, Automation, and Test in Europe (DATE), Munich, Germany, March 2008, pp. 1498–1503 (2008)
26. Vasudevan, N., Edwards, S.A.: Celling SHIM: Compiling deterministic concurrency to a heterogeneous multicore. In: Proceedings of the Symposium on Applied Computing (SAC), Honolulu, Hawaii, March 2009, vol. III, pp. 1626–1631 (2009)
27. Vasudevan, N., Singh, S., Edwards, S.A.: A deterministic multi-way rendezvous library for Haskell. In: Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS), Miami, Florida, April 2008, pp. 1–12 (2008)
28. Vasudevan, N., Edwards, S.A.: A determinizing compiler. In: Proceedings of Program Language Design and Implementation (PLDI), Dublin, Ireland (June 2009)
29. Vasudevan, N., Tardieu, O., Dolby, J., Edwards, S.A.: Compile-time analysis and specialization of clocks in concurrent programs. In: de Moor, O., Schwartzbach, M. (eds.) CC 2009. LNCS, vol. 5501, pp. 48–62. Springer, Heidelberg (2009)