SHIM: A Language for Hardware/Software Integration

Stephen A. Edwards*
Department of Computer Science, Columbia University
1214 Amsterdam Avenue, New York, New York, 10027
sedwards@cs.columbia.edu

Abstract

Virtually every system designed today is an amalgam of hardware and software. Unfortunately, software and circuits that communicate across the hardware/software boundary are tedious and error-prone to create. This suggests a more automatic way to synthesize them.

This paper presents the SHIM *language, which combines imperative C-like semantics for software and* RTL*-like semantics for hardware to allow a unified description of hardware/software systems. Hardware processes and software functions communicate through shared variables, hardware for which is automatically synthesized by the* SHIM *compiler, which generates C and synthesizable* VHDL.

I demonstrate the effectiveness of the language by re-implementing an I^2C bus controller. The SHIM *source is half the size of an equivalent manual implementation, slightly faster, and has a smaller memory footprint. Partial and complete hardware implementations in* SHIM *are also presented, showing that* SHIM *is succinct and effective.*

1 Introduction

As integrated circuit technology advances relentlessly, the size and complexity of a typical design continues to spiral upward. As always, managing complexity is the designer's greatest challenge. The design must be right the first time and be completed faster than before. While validation methods such as simulation and formal verification work well to discover mistakes, moving to higher levels of abstraction, such as from the gate level to the register transfer level, is more effective because it helps designers to avoid mistakes in the first place. In this paper, I describe a language that raises the abstraction level for hardware/software systems.

My intention with SHIM (Software/Hardware Integration Medium), the language I propose here, is to provide seamless communication between hardware and software modules. It arose in part from observing beginning design students tackle combined hardware/software systems: they understood C well, and could learn VHDL, but had difficulty making the two worlds communicate.

Rather than propose a completely new semantics for SHIM, I chose to integrate two well-known, well-established semantics: C-like imperative semantics for the software portion of the design and register-transfer level semantics for the hardware. The syntax of SHIM most closely mimics C, which I chose both for its familiarity (C, C++, Java, and C# programmers all know it well) and its succinctness.

The SHIM compiler takes a single specification—a collection of software functions and hardware processes—and generates both C and synthesizable VHDL source. No attempt is made to do automatic partitioning between the two domains: it is the user's responsibility to mark everything as hardware, software, or shared. This is partially to simplify the compiler, but the more serious issue is the very different semantics of the two domains. When I began this project, I wanted to make switching functionality between the two domains as easy as marking a function differently, but the semantics of RTL and software are just too different: how many clock cycles should a piece of software code take, and how is concurrency implemented in software? Hardware-like software and software-like hardware languages (e.g., Esterel [2] and Handel-C [4]) have been proposed, but all deviate significantly from software and RTL semantics and none are widely accepted.

Aside from minor improvements in RTL syntax (SHIM's RTL can be half as verbose as the equivalent VHDL), SHIM's major contribution is the automatic synthesis of communication between the software and hardware components based on a shared memory model. While algorithmically simple, such synthesis simplifies the designer's task, avoids errors, and improves code portability.

Figure 1a shows a simple SHIM program that represents a hardware timer. It manages the shared variable `counter`, whose value is stored in the hardware but can also be read and written by the two software functions `reset_timer` and `get_time`. From this description, the compiler produces a C header file (Figure 1b) that describes the external software interface for the module, a C source file containing code implementing `reset_timer` and `get_time` (Figure 1c), and VHDL for a peripheral that attaches to a proces-

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

```
module timer {
  shared uint:32 counter;

  hw void count() {
    counter = counter + 1;
  }

  out void reset_timer() {
    counter = 0;
  }

  out uint get_time() {
    return counter;
  }
}
```

(a)

```
#ifndef _TIMER_H
#define _TIMER_H
extern void reset_timer(void);
extern unsigned int
    get_time(void);
#endif /* _TIMER_H */
```

(b)

```
#include "timer.h"
#include "xio.h"
#define IO_BASE 0xfeff0200
#define counter (IO_BASE + 0x0)

void reset_timer() {
  XIo_Out32(counter, 0);
}
unsigned int get_time() {
  return XIo_In32(counter);
}
```

(c)

```
signal counter : UNSIGNED(31 downto 0);

count : process(Clk)
begin
  if Clk'event and Clk = '1' then
    counter <= counter + 1;
    if cs1 = '1' and RNW = '0' then
      if offset = 0 then
        counter <= DBus;
  end if; end if; end if;
end process count;

read_shared_variables : process(Clk)
begin
  if Clk'event and Clk = '1' then
    if cs1 = '1' and RNW = '1' then
      DBus_out <= read_data;
    else DBus_out <= "0"; end if;
    if offset = 0 then
      read_data <= counter;
  end if; end if;
end process read_shared_variables;
```

(d)

Figure 1: (a) A simple SHIM program: a hardware timer. (b) The C header file generated by the SHIM compiler. (c) Generated C source. (d) A fragment of generated VHDL. These two processes implement the SHIM *count* process and the ability to read the value of *count* from software. In the first process, *count* is incremented or read from the bus if chip select and write are true. The second process places the value of *count* on the bus if chip select and read are true.

sor bus and implements the `count` process and circuitry that allows the shared variable to be read and written from the C program (Figure 1d).

This example illustrates how SHIM makes it easier to write hardware/software systems: the SHIM source is only fifteen lines, but the SHIM compiler generates twenty lines of C and nearly 100 lines of VHDL from it, code a designer would have otherwise had to write manually. The fractional improvement is high for such a small example because most of the code is related to the bus interface, but large examples remain smaller and easier to code.

As its name suggests, SHIM is designed for creating "glue" that connects hardware and software. It specifically does not try to be a general-purpose software language or a full hardware description language. Instead, it provides facilities for connecting subsystems written in hardware and software and external interfaces: externally-visible functions and variables in software, ports in hardware.

SHIM can also be thought of as a language for simultaneously writing peripherals and their device drivers. As such, it is applicable to systems where hardware can be customized, such as ASICs with processor cores; field-programmable gate arrays, especially those with processors such as the Virtex II Pro; and more programmable SoCs that consist of multiple processor cores, hard peripherals, and a substantial amount of programmable logic. This latter class of chip, a platform with a mix of programmable hardware and software, seems a likely architecture of the future since the cost of designing complete chips is skyrocketing; it seems more likely that standard programmable platforms will grow more popular. When the hardware is provided and immutable, the NDL language [8] would be better-suited.

SHIM currently generates code (hardware and software) for the Xilinx Microblaze soft processor core driving IBM's CoreConnect On-Chip Peripheral Bus (OPB). I chose this configuration because we have target boards from XESS and a development environment from Xilinx, but there is nothing OPB-specific about the SHIM language. The synthesis code is about 1/6th of the compiler and could be easily retargeted to a different processor and bus.

2 Related Work

Unlike other hardware/software codesign systems, SHIM uses imperative C semantics for software and RTL for hardware. Polis [1] describes its systems with communicating extended finite-state machines. COSMOS (Jerraya et al. [11]), COSYMA (Ernst et al. [9]), and CoWare (Bolsens et al. [3]) all use concurrently-running processes communicating through remote procedure calls (RPC). FIFO-based communication among concurrently-running processes is another choice (see, e.g., Gupta and De Micheli [10]).

Although such higher-level semantics enable automatic hardware/software partitioning (a focus of earlier work), they raise efficiency issues. While RPC is natural in software, it seems overly sequential for hardware. Furthermore, some techniques employ high-level hardware synthe-

sis, which industry has largely rejected due to efficiency concerns. COSYMA describes a hardware/software system using a C-like imperative language, partitions it, and passes certain processes to high-level synthesis.

Most other work proposes synthesizing communication mechanisms far more complicated than shared memory. Jerraya et al. [5] propose synthesizing wrappers. CoWare layers protocols. It is unclear whether these approaches are general enough and produce efficient hardware.

SHIM synthesizes bus-based communication, but is not wedded to it. Chou et al. [6, 7] target microcontrollers communicating through a limited number of I/O pins.

Other languages, such as Mérillon et al.'s Devil [13] (generates C macros for communicating with peripherals), Thibault et al.'s GAL [14] (generates graphics adapter drivers), and Conway et al.'s NDL [8] (synthesizes Unix device drivers), focus only on synthesizing software and assume the hardware is given.

Lavagno and Sentovich's ECL [12], which combines Esterel-like synchrony [2] with imperative C, partially inspired SHIM. Like SHIM, their compiler uses simple rules to split a program into the two domains, but they interleave Esterel and C at the statement (instead of function) level and they do not target hardware/software systems.

3 The SHIM Language

SHIM was designed to be simple. A SHIM program is a module containing global variables and functions. Each variable is either unmarked, marked `hw`, or marked `shared` indicating it is to be visible in software only, hardware only, or both. The translation of hardware- and software-only variables is straightforward; the state of a shared variable is held in hardware and the compiler synthesizes circuitry that allows it to be read and written from software. Each variable can also be marked `in` or `out`, indicating its value comes from outside the module or is visible outside (variables are only module-visible by default).

Functions may either be unmarked (indicating software) or marked `hw`. The translation of a software function is nearly one-to-one. A hardware function becomes a concurrently-running synchronous hardware process clocked by the bus to which the synthesized peripheral is attached. Currently, all hardware functions take no arguments, do not return a value, and may not be explicitly called (they are implicitly called once per clock cycle).

SHIM supports bit vectors, arrays of bit vectors, and string literals. Bit vectors can be manipulated as integers and are either signed (`int`) or unsigned (`uint`). The width of each bit vector may be specified or may be omitted (it defaults to a standard value). Array dimensions must be given.

The body of a SHIM function contains the familiar set of C statements, `if`, `switch`, `for`, `while`, and `return`; expressions; and local variable declarations. Expressions follow the usual C syntax with a few extensions: applying the array index operator to a bit vector returns a bit, e.g., `a[1]` is the second least-significant bit of vector `a`. Looping statements are not allowed in hardware processes.

Software functions have the usual sequential semantics; there is a single program counter; bit vectors are call-by-value, and arrays are call-by-reference.

Hardware processes have RTL semantics. Every hardware process is effectively invoked once per clock cycle. Each shared or hardware-only variable may be written by a single process only, a syntactic constraint common in RTL, but may be read by multiple processes. Outputs are latched, meaning that if a process writes a variable, the new value can be seen by other processes only in the next cycle, but the same process can see the new value in the same cycle.

When software-only variables have an initial expression, the value is assigned once when the variable enters scope (e.g., when the program starts for global variables, and when control enters the enclosing block for local variables) and stored. Software-only variables marked `const` must be given an initial value that can be determined at compile-time and may not be written.

Hardware-only variables may also be assigned to an expression, but their semantics differ: hardware variables assigned to an expression always take the value of that expression (i.e., are effectively continuous assignments) and may not be assigned in other processes. Such variables generate combinational logic and conceptually execute after all the processes have executed for the cycle and updated their outputs. That is, they can be used to communicate among processes, but only across clock-cycle boundaries.

4 Compiling SHIM

The SHIM compiler comprises some some 3000 lines of OCAML code. The largest single module (about 500 lines) performs the interface synthesis. Much of it comes from explicit construction of SHIM code for the bus controller.

The compiler's structure is typical: an automatically-generated scanner and parser generates an abstract syntax tree, which is sent through a static semantic analyzer that resolves names and types and dismantles certain constructs to produce an AST-like intermediate representation.

The interface generator takes the interface-agnostic IR from the dismantler, enumerates the shared variables (a simple walk through the symbol table), and assigns each an I/O address. It transforms the body of each software function by changing reads and writes of shared variables to the appropriate I/O function call.

Interface synthesis for hardware is more complicated. At the end of each process, the compiler adds code that reads from the bus each shared variable written in that process. This generates at most one read of each shared variable since each such variable may be written by at most one

hardware process (software functions have no such restriction). Furthermore, placing the read at the end of the process means a write from the software domain takes precedence over a write from hardware. I chose these semantics because hardware writes every cycle are a common idiom (e.g., the `count` variable in Figure 1). Users who dislike software taking precedence over hardware can use separate shared variables for each communication direction.

Two additional hardware processes are synthesized: one reads all the unwritten shared variables from the bus (e.g., those shared variables that are only written from software, never from hardware). The other copies each shared variable to the bus when requested (i.e., when software generates a read cycle to an I/O address).

Finally, the interface synthesis procedure adds interface ports for all the bus signals and processes that decode the address bus and speak the bus protocol. After this, the IR contains all the necessary interface code.

Once the interface synthesis procedure runs, C source, C header, or VHDL source is generated. Syntax-directed translators transform the IR into simple ASTs for C or VHDL, which are then pretty-printed. Using another intermediate representation guarantees syntactically-correct output.

5 An Example: An I^{2}C Bus Controller

To test SHIM, I re-implemented a simple hardware/software interface with it. In an embedded systems class, we use XSB–300E boards from XESS Corporation that contain, among other things, a Xilinx Spartan IIE FPGA and a Philips SAA7114H video decoder. An I^{2}C bus connecting the two can write to the 7114's many configuration registers. The I^{2}C bus is a low-speed two-wire clocked bus designed for exactly such an application, and is very lenient about timing. The bus has no maximum delays between transitions so it can be controlled completely from software.

One of my students (Marcio Buss) implemented a simple all-software I^{2}C bus controller to program the 7114. Its only task is to send a series of commands to write over eighty configuration registers. Marcio spent a few days writing its 259 lines of C and 133 lines of VHDL. The VHDL interfaces with the On-chip Peripheral Bus and provides two shared variables that control the two I^{2}C bus pins. The C program handles the bus protocol.

Table 1 shows how the handwritten implementation of the I^{2}C bus controller compares to equivalent designs written in SHIM. The software version implements the protocol in software; the hardware does little more than control the pair of tristate drivers called for by the bus interface. The three other versions of the I^{2}C controller do more in hardware: the first implements the bit-level protocol in hardware (the bytes are still sequenced by software), the second moves the complete receive functionality to hardware, and the third places both send and receive functionality in hardware (bytes are still sequenced in software).

Example	SHIM	C	VHDL
I^{2}C Software (by hand)		259	133
I^{2}C Software	171	175	136
I^{2}C Bit-level	283	173	337
I^{2}C Byte-level Receive	299	163	358
I^{2}C Byte-level Send/Receive	323	116	344
Timer	15	20	98

Table 1: Lines of code for various examples. The first line is a reference implementation of an I^{2}C bus controller written without the aid of SHIM. The next four lines list SHIM implementations with the same functionality with an increasing fraction of the system in hardware. The final line lists statistics for the simple example in Figure 1. For all but the first line, the SHIM column represents the number of lines written by the designer; the C and VHDL columns are lines of code generated by the SHIM compiler.

The first two lines of this table are most telling: Marcio wrote $259 + 133 = 392$ lines of C and VHDL to implement this controller, while I was able to achieve the same functionality in 171 lines of SHIM, a reduction of over 55%.

Figure 2 shows a fragment of the all-software SHIM implementation of the I^{2}C controller. Controls for the two I^{2}C wires, SDA and SCL, are shared one-bit variables, and the *send* function simply toggles them appropriately.

I then implemented more of this controller in hardware using SHIM, the sizes of which I report in the next three lines of the table. By design, changing the implementation of a function from software to hardware is not as simple as marking it as `hw` (SHIM hardware uses RTL semantics, which are very different from C's). The main difference is that sequencing must be coded as a state machine using, say, a *switch* statement (see Figure 3). This is the main source of the additional 152 lines of SHIM.

Figure 3 is a hardware fragment similar to the software of Figure 2. It toggles the clock and data lines to send a single byte of data. Most of the code is now devoted to sequencing since traditional RTL, and by design SHIM, requires all state machines' next-state functions to be explicit. Additional complexity comes from synchronizing with software, done here with a four-phase handshake. The controller raises `ready` in the `IDLE` state to indicate it is ready to accept a command. The software then writes a command such as `SEND` into the `command` variable, which steps the state machine through a state sequence ending with `IDLE0`. In this state, the controller waits until it receives `IDLE` from the software, which sends it back to the `IDLE` state. The `send` function at the bottom of Figure 3 performs the software half of this handshake.

The SHIM version of a controller can be faster than its handwritten equivalent. In the handwritten version of the

```
shared out bool SCL;    // I2C clock
shared out bool SDA;    // I2C data out
shared out bool SDA_oe; // Output enable for data
shared bool SDA_data;   // I2C data in

void send(uint:8 byte) {
  SDA_oe = 0; delay();
  for (int i = 7 ; i >= 0 ; i = i - 1) {
    SDA = (byte & 0x80) >> 7; delay();
    SCL = 1; byte = byte << 1; delay();
    SCL = 0; delay();
  }
  SDA_oe = 1; delay();
  SCL = 1; delay();
  bool acknowledge_received = SDA_data;
  if (!acknowledge_received)
    xio.print("Acknowledge not received\r\n");
  delay();
  SCL = 0; delay();
}
```

Figure 2: A fragment of the SHIM code in the all-software version of the I^2C protocol that sends a single byte to a slave and looks for an acknowledgement. Each output bit is placed on the data wire and the clock is toggled, then the data wire is set to read, the clock is toggled, and the acknowledge bit from the slave is read.

I^2C controller, which was not written for efficiency, Marcio chose to pack the four control bits for the SDA and SCL lines into a single I/O location. While this simplifies address decoding hardware, it requires a read-modify-write operation to change a single bit from software. The SHIM-generated code is more efficient since it can modify each control bit individually.

The size of the SHIM-generated C code can also be superior to handwritten code. The object file for the all-software version of the I^2C controller generated by SHIM is only 2106 bytes. By contrast, the handwritten C is over twice the size (4370 bytes). The difference is due to additional function calls and read-modify-write operations.

6 Conclusions

I have presented the SHIM language and its compiler. SHIM fuses two widely-accepted computational models—single-threaded imperative software and register-transfer-level hardware—to allow hardware/software systems to be written in a unified language. Declaring a SHIM variable *shared* allows it to be read and written from both hardware and software; the SHIM compiler synthesizes this interface.

I demonstrated the effectiveness of SHIM on an example—an I^2C bus controller—and showed that it can reduce by half the number of lines necessary to describe such a system. The savings comes in part from a more succinct syntax than that of VHDL, but is mostly due to the automatic synthesis of a bus interface.

```
shared uint:8 sreg;    // Send/receive shift register
shared uint:5 state;   // Controller state
shared bool ready;     // true => controller idling
shared uint:3 command; // Command for the controller
shared const uint:3 IDLE    = 0; // Commands
shared const uint:3 SEND    = 2;

hw void controller() {
  const uint:5 IDLE  = 0;  const uint:5 SEND1 = 5;
  const uint:5 SEND2 = 6; const uint:5 IDLE0  = 24;
  uint:3 bit_counter;

  if (reset) state = IDLE;
  if (i2c_clock) {
    ready = 0;
    switch (state) {
    case IDLE:
      ready = 1;
      switch (command) {
      case START:   state = START1;  break;
      case SEND:    state = SEND1;   break;
      case RECEIVE: state = RECV1;   break;
      case STOP:    state = STOP1;   break;
      default:      state = IDLE;    break;
      }
      break;
    case SEND1:
      SDA_oe = 0; bit_counter = 0; state = SEND2;
      break;
    case SEND2:
      SDA = sreg[7]; state = SEND3; break;
    case SEND3:
      SCL = 1; sreg = sreg << 1;
      bit_counter = bit_counter - 1; state = SEND4;
      break;
    case SEND4:
      SCL = 0; if (bit_counter == 0) state = SEND5;
      else state = SEND2;  break;
    // Receive Acknowledge
    case SEND5:
      SDA_oe = 1; state = SEND6; break;
    case SEND6:
      SCL = 1; state = SEND7; break;
    case SEND7:
      acknowledge_received = SDA_in; state = SEND8;
   case SEND8:
      SCL = 0; state = IDLE0; break;
   case IDLE0:
      if (command == IDLE_BIT) state = IDLE;
      else state = IDLE0; break;
    }
  }
}

void send(uint:8 byte) {
  sreg = byte;
  command = SEND_BIT; while (ready) ;
  command = IDLE_BIT; while (!ready) ;
}
```

Figure 3: A fragment of the all-hardware implementation of the I^2C bus controller in SHIM. This shows part of the main hardware state machine responsible for toggling the I^2C clock and data lines to send a byte (cf. Figure 2) and the software function that invokes it with a handshake.

I believe SHIM is successful in its aim to fuse two computational models but it raises the question of whether these two models are the best choices. A particularly glaring issue is that SHIM models are not easy to simulate. This is due to the models themselves: the two domains run asynchronously and while the hardware is timed, the software effectively is not, meaning that the behavior of the system may be nondeterministic or at least very difficult to predict without careful modeling of software timing, such as by using an instruction-set simulator. The timer example in Figure 1 is perhaps the simplest example illustrating this problem: the hardware effectively counts the number of clock cycles between calls of `reset_timer` and `get_time`, which is a complicated function of the processor and C compiler used to implement the system. Of course, such a performance timer is often desired for analyzing software, but more frequently the behavior of hardware/software systems is meant to be timing independent.

My feeling is that the shared-variable model of hardware/software communication, while the de facto standard, is too flexible and more complicated, but timing-independent, protocols are almost always implemented on top of it. For example, a peripheral usually provides status registers and interrupts that inform the software when it is ready for the next command, and expects that software will obey its protocol. In implementing the I^2C bus controller described in Section 5, I implemented a four-phase handshake protocol to insure that the software waited for the hardware to complete its task before starting the next. Although this works and is robust in practice, it can be error-prone and is certainly more verbose than it needs to be.

Part of the problem is that the hardware/software boundary represents true parallelism and concurrency. Peripherals run truly independently from their processors (in most cases, that is the point of a peripheral), and so are truly concurrent systems posing all the classical problems such as races and deadlocks. However, since hardware/software systems are neither completely hardware nor software, using classical software techniques such as semaphores and critical regions or the standard hardware technique of a global clock seems inappropriate.

So I leave this question for future work: what is an appropriate general, timing-independent model of computation for hardware/software systems? Single-threaded imperative software and RTL hardware with shared variables, such as in SHIM, is reasonably general, but is too low-level, easily nondeterministic, and error-prone. There must be something higher-level that avoids tedious manual protocol implementation and ensures correctness.

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

[3] Ivo Bolsens, Hugo J. De Man, Bill Lin, Karl Van Rompaey, Steven Vercauteren, and Diederik Verkest. Hardware/software co-design of digital telecommunication systems. *Proceedings of the IEEE*, 85(3):391–418, March 1997.

[4] Celoxica, http://www.celoxica.com. *Handel-C Language Reference Manual*, 2003. RM-1003-4.0.

[5] W. Cesário, A. Baghdadi, L. Gauthier, D. Lyonnard, G. Nicolescu, Y. Paviot, S. Yoo, A. A. Jerraya, and M. Diaz-Nava. Component-based design approach for multicore SoCs. In *Proceedings of the 39th Design Automation Conference*, pages 789–794, New Orleans, Louisiana, June 2002.

[6] Pai Chou, Ross B. Ortega, and Gaetano Borriello. Synthesis of the hardware/software interface in microcontroller-based systems. In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 488–495, San Jose, California, November 1992.

[7] Pai Chou, Ross B. Ortega, and Gaetano Borriello. Interface co-synthesis techniques for embedded systems. In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 280–287, San Jose, California, November 1995.

[8] Christopher L. Conway and Stephen A. Edwards. NDL: a domain-specifi c language for device drivers. In *Proceedings of Languages, Compilers, and Tools for Embedded Systems (LCTES)*, page FIXME, Washington, DC, June 2004.

[9] Rolf Ernst, Jörg Henkel, Thomas Benner, Wei Ye, Ulrich Holtmann, Dirk Herrmann, and Michael Trawny. The COSYMA environment for hardware/software cosynthesis of small embedded systems. *Microprocessors and Microsystems*, 20(3):159–166, May 1996.

[10] Rajesh K. Gupta and Giovanni De Micheli. Hardware-software cosynthesis for digital systems. *IEEE Design & Test of Computers*, 10(3):29–41, October 1993.

[11] Tarek Ben Ismail, Mohamed Abid, and Ahmed Jerraya. COSMOS: A codesign approach for communicating systems. In *Proceedings of the 3rd International Workshop on Hardware/software Co-Design*, pages 17–24, Grenoble, France, September 1994.

[12] Luciano Lavagno and Ellen Sentovich. ECL: A specifi cation environment for system-level design. In *Proceedings of the 36th Design Automation Conference*, pages 511–516, New Orleans, Louisiana, June 1999.

[13] Fabrice Mérillon, Laurent Réveillère, Charles Consel, Renaud Marlet, and Gilles Muller. Devil: An IDL for hardware programming. In *Proceeedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 17–30, San Diego, California, October 2000.

[14] Scott A. Thibault, Renaud Marlet, and Charles Consel. Domain-specifi c languages: from design to implementation application to video device drivers generation. *IEEE Transactions on Software Engineering*, 25(3):363–377, May 1999.