Achieving Time-Controllable Digital Mockup Execution using SystemC

SCOTT SIROWY, BAILEY MILLER, AND FRANK VAHID*
University of California, Riverside
*also with the Center for Embedded Computer Systems, University of California, Irvine

Medical device software is sometimes initially developed by using a PC simulation environment that executes models of both the device and a physiological system, and then later by connecting the actual medical device to a physical mockup of the physiological system. An alternative is to connect the medical device to a *digital mockup* of the physiological system, such that the device software executes as if it is interacting with a physiological system, but in fact all interaction is digital. Developing medical device software by interfacing with a digital mockup enables development without costly or dangerous physical mockups, allows testing of extreme conditions, and provides for execution that is faster or slower than real-time. We propose a SystemC-based approach for digital mockup capture and execution, which gives the medical device software developer the ability to start, stop, and step execution based on *time* intervals. Such an ability is in contrast with traditional *instruction* based debugging approaches, and gives a more powerful abstraction when testing medical device software. We detail our SystemC-based capture and execution approach, and provide data from experiments for a ventilator device interacting with a human respiration digital mockup.

Categories and Subject Descriptors: B.5.2 [**Hardware**] – RTL, Optimization, Simulation; C.0 [**Computer Systems Organization**] – HW/SW Interfaces; D.1.3 [**Concurrent Programming**] - Parallel Programming;
General Terms: Design, Experimentation, Human Factors, Measurement, Performance
Additional Key Words and Phrases: Digital mockups, SystemC, Spatial Computing, Medical Devices, Software Development, Emulation, Real-Time

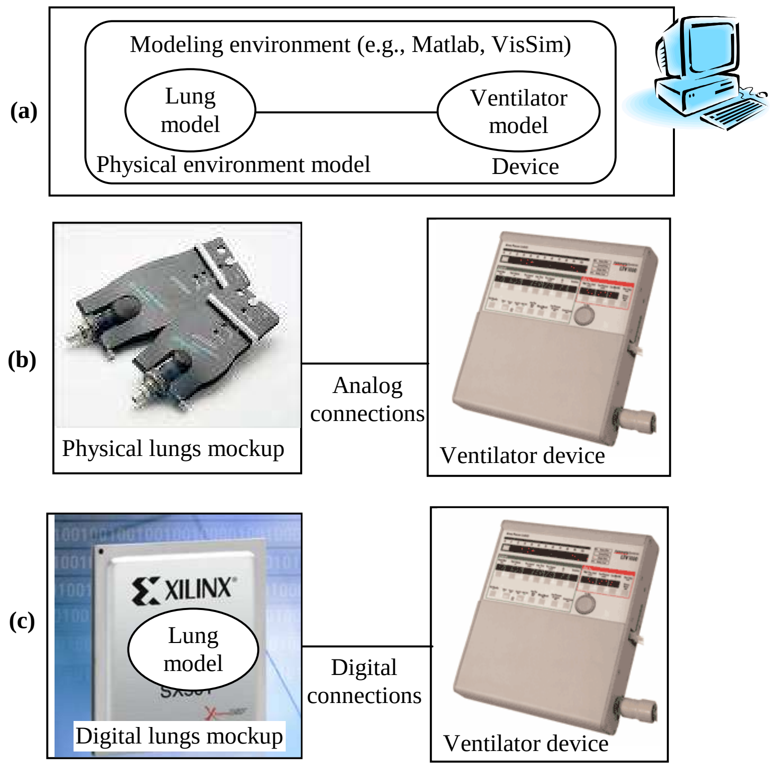
1. INTRODUCTION

Medical device software is commonly developed using one of several approaches. One approach involves modeling on a PC, shown in Figure 1(a). A designer develops models for both a medical device, such as a pacemaker or ventilator, and for the physiological system with which the device interacts, such as a heart or lung. Such a modeling approach supports rapid device software changes, supports simulations that execute faster (or slower) than real-time, and avoids potential safety issues that could arise when interacting with an actual physical system.

A second approach, used after or instead of the modeling approach, runs the medical

This research was supported in part by the National Science Foundation under Grant CNS-0614957 and by the Office of Naval Research under the Contract Number N00014-07-C-0311.
Authors' addresses: Scott Sirowy, Department of Computer Science The University of .California, Riverside, CA, 92507; Bailey Miller, Department of Computer Science, The University of California, Riverside, CA, 92507; Frank Vahid, Department of Computer Science, The University of California, Riverside, CA 92507.

Figure 1: Approaches to integrating an embedded device with the physical environment during design: (a) system model, (b) physical mockup, (c) digital mockup**.**



device software on the actual medical device, which is connected to a physical mockup of the physiological system. Physical mockups range from simple structures, such as a balloon representing a lung, to computerized mechanical parts that dynamically react [Michigan Instruments], that can be set to mimic a range of conditions, and whose internal sensors can be interfaced to a computer for analysis and debugging. One disadvantage of interfacing to off-the-shelf physical mockups is the inability to adapt to new features, especially features not easily mimicked via mechanical means. For example, a future ventilator may sense human-generated nitric oxide concentrations (recently discovered to be significant in respiratory issues [Shin]) and adapt the output gas mix in response. However, no existing computerized mechanical test lung generates nitric oxide, nor is it clear how to create one.

An alternative is to connect the actual medical device to a digital mockup of the physiological system. A *digital mockup* is a behavioral model that emulates the physical system. In such a case, the medical device software executes as if it were interacting with a physiological system, but in fact all interaction is through a digital interface, as in

Figure 2: Digital mock-up platform: (a) The bypass method of integration taps directly into the digital information packets that indicate the data/control values to/from the device sensors/actuators, (b) the method matches hardware-in-the-loop approaches used in industrial practice (*figure courtesy of Boeing, 2009*).

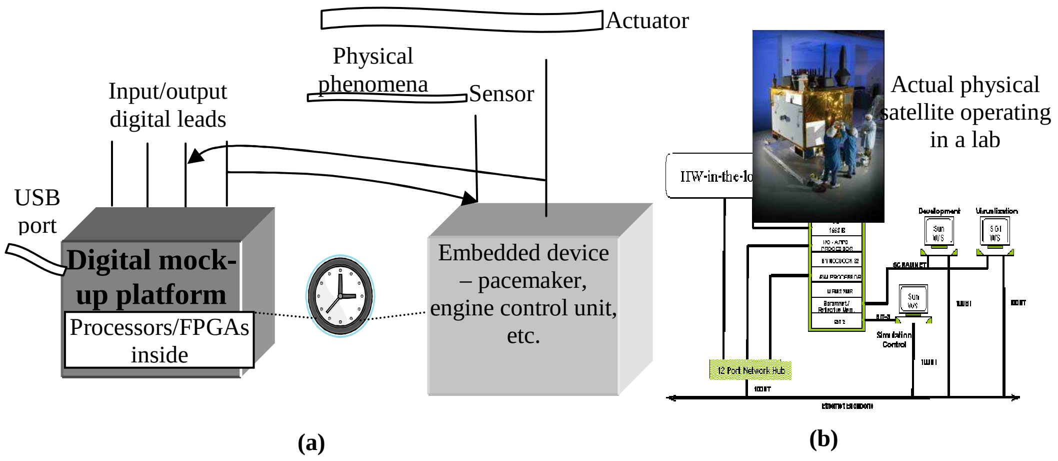


Figure 1(c). We consider a digital mockup platform with a sensor/actuator bypass method of integration [Sirowy], as shown in Figure 2(a). Under this scheme, the digital mockup taps directly into the information packets that carry the control and data bitsto and from the device's sensors and actuators. The digital mockup includes models of the physiological system, of the physical connections between the device and physiological system, and of the sensors and actuators. A supervisory system coordinates execution of the digital mockup and medical device. The sensor/actuator bypass method is in harmony with methods used in industrial "hardware-in-the-loop" practice today, shown in Figure 2(b). Digital mockups combine the flexibility and faster-than-real-time execution benefits of PC simulation models with the advantages of developing software on an actual medical device. Digital mockups are also potentially less costly than physical mockups, which can cost tens of thousands of dollars.

However, no common methodology exists for creating digital mockups. Towards this end, we sought to develop a general approach for time-controllable digital mockup execution. Digital mockups can be implemented through a variety of methods and on a variety of different platforms, trading off performance, complexity, size, and accuracy. While a medical device software developer may run a digital mockup directly on the physical development platform for increased performance and/or accuracy, another approach is to run the digital mockup on top of *virtualized platform* like an in-circuit emulator. By varying the rate at which the digital mockup generates samples, the digital mockup can still run faster than or in real-time to interface with the medical device software under test. A virtualized environment can also provide built-in and unobtrusive debug capabilities, allowing the designer to stop, start, and step through the digital mockup to examine important system variables. The virtualized environment can exploit the digital mockup’s explicit notion of a simulated time step, allowing the designer to monitor the mockup using *time-controllable* debug. For example, a medical device software developer may wish to step through a wheezing lung that just coughed one time step at a time (physiological models are defined to compute the next system values in time based on a *delta time* parameter), observing subtle differences in pressure and volume in the digital lung that might not easily be observed when running in real-time.

We describe a *time-controllable* SystemC-on-a-Chip framework that allows a medical device software developer to interface a medical device to a SystemC-based digital mockup, and start, stop, profile, and advance execution using explicit time-granularized debug commands. This is contrasted to a more traditional debugging approach, where debugging is performed at the *instruction* granularity, and which does not include an explicit notion of time. A time-granularized approach is generally more useful for physiological digital mockups, and provides a more powerful abstraction for developing and testing medical device software.

2. SYSTEMC FOR SYNCHRONIZED PHYSIOLOGICAL MODELS

There are a number of approaches to capturing and implementing physiological systems models. Physiological systems are usually first modeled using systems (hundreds or thousands) of partial and ordinary differential equations. The model can then be captured for PC execution using a particular programming language, typically an expressive mathematical language like MML, Matlab, or VisSim.

Another method to capturing physiological systems models is to use SystemC. SystemC is a set of libraries built on top of the C++ language that provides an event-driven simulation kernel, allowing a designer to simulate a number of concurrently executing processes, and which supports precisely-timed communication based on simulated time. SystemC is a natural fit for capturing physiological systems models for a number of reasons. The equations that represent most physiological systems are naturally expressed as a number of concurrently executing interconnected processes that execute in lockstep. Digital physiological mockups implemented in SystemC have the added advantage that freely available SystemC simulation environments exist that enable comprehensive PC testing. Further, the developer can run SystemC on a real development platform using an in-circuit emulation approach like *SystemC-on-a-Chip* [Sirowy], with the advantage that the SystemC-based digital mockup executes with real peripherals, and with real devices, like medical device platforms

Figure 3: Capturing physiological models in SystemC. (a) Portion of a mathematical model of the human lung. (b) Description of the model in SystemC. (c) Description using POSIX threads. The POSIX threads approach requires implementing explicit lock-stepping mechanisms that detract from the model's readability.

(a)

$$
\begin{aligned}
C_{br} &= Q_{br} / V_{br} \\
F_{br} &= (P_{air} - P_{br}) / R_{br} \\
dQbr/dt &= F_{br} * (C_{air} + C_{br}) + F_{alv} * (C_{alv} - C_{br})
\end{aligned}
$$

(b)

```
Class model : public sc_module {
  sc_in_clk clock;

  integrator Q_integ;

  sc_signal<sc_uint<32> > Qbr, Qbr_t;
  sc_signal<sc_uint<32> > Cbr, Fbr;

  SC_CTOR {
    Q_integ.clock(clock);
    Q_integ.func(Qbr_t);
    Q_integ.dt(dt);
    Q_integ.out(Qbr);

    SC_METHOD(cbr_func);
    sensitive << clock;
    SC_METHOD(fbr_func);
    sensitive << clock;
    SC_METHOD(qbr_t_func);
    sensitive << clock;
  }

  void cbr_func( void ) {
     Cbr = Qbr / Vbr;
  }
  void fbr_func( void ) {
     Fbr = (Pair – Pbr) / Rbr;
  }
  void qbr_t_func( void ) {
     Qbr_t = Fbr * (Cair + Cbr) + \
               Falv * (Calv – Cbr);
  }
};
```

SystemC implementation much closer to the mathematical model

(c)

```
int cbr,fbr,qbr_t;
sem_t  timestep_done,cbr_done;
sem_t fbr_done,qbr_t_done;

void * Cbr( void * arg ) {
   while (1) {
      sem_wait(&timestep_done);
      cbr = Qbr / Vbr;
      sem_post(&cbr_done);
   }
}

void * Fbr( void * arg ) {
   while(1) {
      sem_wait(&timestep_done);
      fbr = (Pair – Pbr) / Rbr;
      sem_post(&fbr_done);
   }
}

void * Qbr_t( void * arg ) {
   while(1) {
      sem_wait(&timestep_done);
      sem_wait(&cbr_done);
      sem_wait(&fbr_done);
      qbr_t = fbr*(Cair + cbr) +
Falv*(Calv – cbr);
      sem_post(&qbr_t_done);
   }
}

void * ClockTick( void * arg ) {
   while(1){
      sem_wait(&qbr_t_done);
      sem_post(&cbr_done);
      sem_post(&fbr_done);
      sem_post(&qbr_t_done);
      sem_post(&timestep_done);
   }
}

int main(){
   pthread_t pCbr;
   pthread_t pFbr;
   pthread_t pQbr_t;
   …
   pthread_create(&pCbr);
   pthread_create(&pFbr);
   pthread_create(&pQbr_t);
   pthread_join(pCbr, NULL);
   pthread_join(pFbr, NULL);
   pthread_join(pQbr_t, NULL);

   return 0;
}
```

Extraneous code to implement synchronous locksteps detract from actual model

While solutions can be implemented in other parallel programming paradigms like POSIX threads or Java threads that also operate with precise timing constraints, physiological models are more naturally represented in SystemC, where lock-stepped execution is an intrinsic part of the language. A SystemC description can require less code, is more readable, and is also more extendable. Figure 3(a) shows a portion of a human lung model captured with three interconnected equations. Figure 3(b) shows the SystemC description and Figure 3(c) shows a more traditional POSIX–based parallel programming description of the model. The POSIX threads approach requires describing explicit tightly-coupled, time lock-stepping mechanisms that make the description more difficult to read, maintain, and extend. Additionally, there is no clear way to step through a POSIX implementation at the simulated time level without further introducing extraneous code into the model. Matlab can also model a number of interconnected equations using a mathematical approach, but like POSIX and Java descriptions, Matlab does not support explicit timing constructs, and debugging is still performed using standard instruction-granularity debug features.

3. RELATED WORK

[Pimentel and Tirat-Gefen] developed real-time digital mockups that interfaced to medical devices by connecting symmetric D/A (digital-to-analog) and A/D (analog-to-digital) cards to each side. Previous work by [Sirowy] focused on modest modifications to the medical device hardware and software such that a digital mockup could be connected directly. Sirowy's approach still allows the addition of D/A and A/D attachments, but with the added advantage of allowing a designer to completely stay in the digital domain, and to accommodate situations where D/A or A/D conversions are complex (e.g., in the case of gas generation or sensing). Other researchers have developed real-time physiological models [Botros], with a focus on describing the necessary architectures to achieve real-time.

Several research efforts have emphasized creating and cataloging detailed physiological models [IUPS, NSR Physiome, Tawhai]. Those models are targeted for PC-based simulation, yet could be used as a basis for digital mockups. Further, many physiological models are highly complex, often requiring hours or days to simulate a few seconds [Medgadget]. Our initial focus is on real-time digital mockups.

There has been much work in the domain of synchronization mechanisms for distributed systems. [Lamport] describes methods to order events in a distributed system. [Kopetz] also specifies clock synchronization methods, but describes techniques used for a more general network topology. In contrast, our system consists of only two directly connected components, and thus is a simpler synchronization problem because uncertainties in a general network need not be considered.

There have been some efforts that focus on making time an explicit first class entity when designing and programming systems. [Lee] calls for the need to bring time to the forefront of programming languages and models, especially with the rise in cyber physical systems research. [Lee] presents a taxonomy detailing several timing properties that should be explicitly expressed in programming languages for timing oriented behaviors. [Benini] develops methods for performing time granularity debugging by calculating time through knowledge of the system's clock speed and the number of cycles between breakpoints.

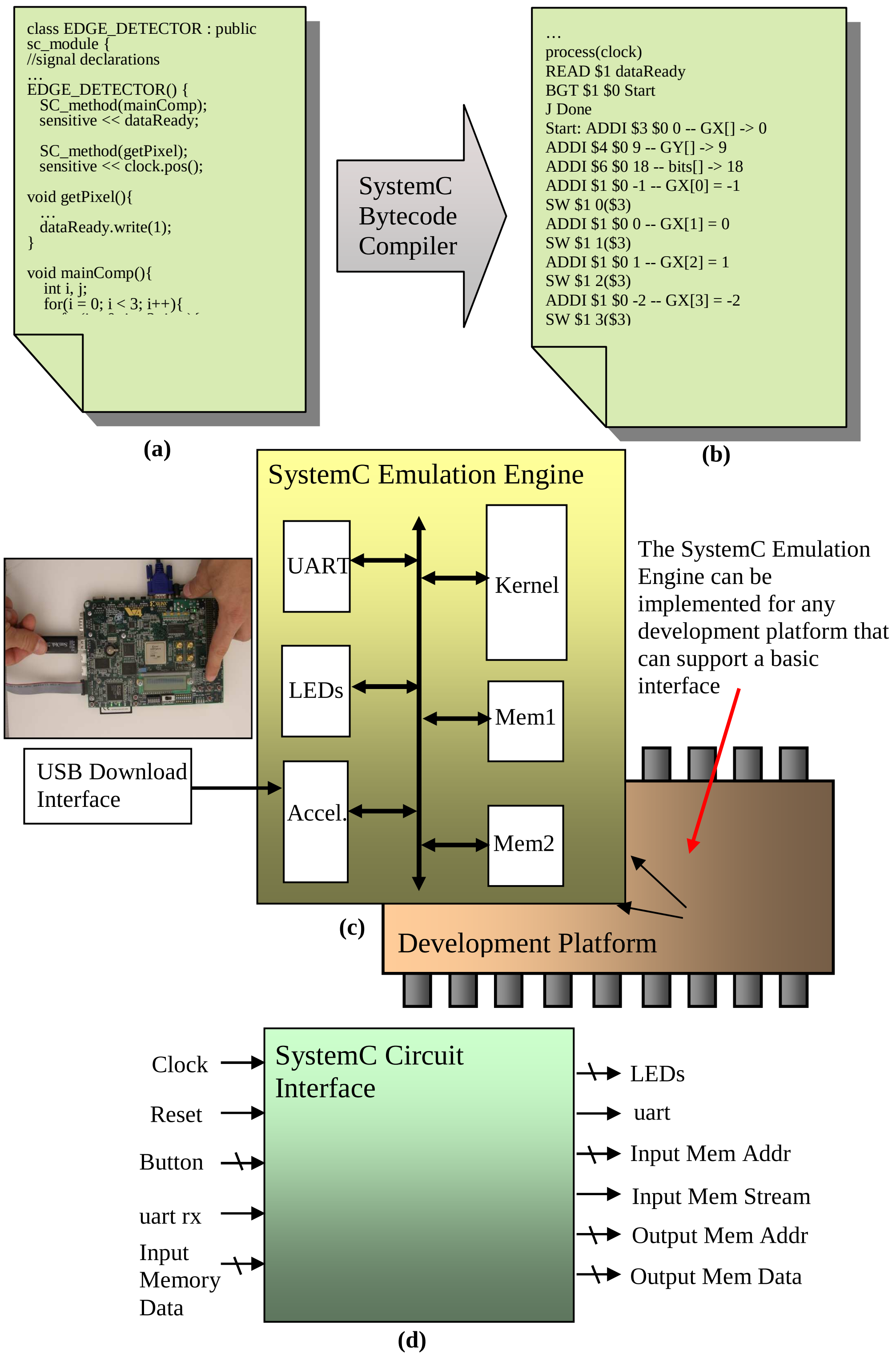
Much research has involved virtualization [Levis, Smith], with several commercial products developed in response to the need for portable virtual machines. VMware [Vmware] and the open source product Xen [Xen] concentrate on developing virtual machines that allow the end-user to run multiple operating systems concurrently. The Java Virtual Machine [Stark] allows the programmer to write operating system independent code, and tools like DOS Box and console emulators allow the user to run legacy applications in modern operating systems. Fornaciari extends virtualization to FPGA platforms, giving the application designer a virtual view of an FPGA that is then physically mapped via operating system functionality. Some work has focused on accelerating Java bytecode through the design of custom bytecode accelerators [Grulan, Parnis]. Virtualization has also been used to abstract complex microcontroller details from the beginning embedded systems student [Sirowy].

There has also been some research in the field of hardware emulation for verification and testing, including the BEE reconfigurable platform [Chang], and network-on-chip emulation platforms [Genko]. [Nakamura] describes a hardware/software verification platform that uses shared register communication between a processor simulator and FPGA emulator. [Benini] describes virtual in-circuit emulation of SystemC circuits for co-verification and timing accurate prototyping. [Rissa] evaluates the emulation speeds of several SystemC models compared to standard HDL models. Our work emphasizes an emulation framework with an explicit notion of time for time-controllable debug.

4. SYSTEMC-ON-A-CHIP FRAMEWORK

The SystemC-on-a-Chip framework, developed by [Sirowy], enables a designer/programmer to capture applications using SystemC, and *immediately* run them

Figure 4: SystemC-on-a-Chip framework. The SystemC-on-a-Chip framework allows a designer to write a SystemC application **(a)**, compile that SystemC through the SystemC bytecode compiler to SystemC bytecode **(b)**, and run that SystemC application immediately on any platform that supports the SystemC emulation engine **(c)**. **(d)**



```
class EDGE_DETECTOR : public
sc_module {
//signal declarations
…
EDGE_DETECTOR() {
   SC_method(mainComp);
   sensitive << dataReady;

   SC_method(getPixel);
   sensitive << clock.pos();

void getPixel(){
   …
   dataReady.write(1);
}

void mainComp(){
    int i, j;
    for(i = 0; i < 3; i++){
```

```
…
process(clock)
READ $1 dataReady
BGT $1 $0 Start
J Done
Start: ADDI $3 $0 0 -- GX[] -> 0
ADDI $4 $0 9 -- GY[] -> 9
ADDI $6 $0 18 -- bits[] -> 18
ADDI $1 $0 -1 -- GX[0] = -1
SW $1 0($3)
ADDI $1 $0 0 -- GX[1] = 0
SW $1 1($3)
ADDI $1 $0 1 -- GX[2] = 1
SW $1 2($3)
ADDI $1 $0 -2 -- GX[3] = -2
SW $1 3($3)
```

The SystemC Emulation Engine can be implemented for any development platform that can support a basic interface on any development platform which supports the SystemC emulation engine, without the need for costly synthesis and mapping tool suites. The SystemC-on-a-Chip framework follows the “write once, run anywhere” paradigm made popular by language frameworks like Java and C#.

Figure 4 highlights the SystemC-on-a-Chip framework. A designer first writes SystemC code, a combination of traditional programming features and spatial programming constructs, which captures the desired functionality (shown in Figure 4(a)). The designer’s SystemC code runs through a SystemC *bytecode compiler.* The SystemC bytecode compiler parses the source SystemC code and outputs *SystemC* bytecode, shown in Figure 4(b). Akin to Java bytecode, SystemC bytecode is an intermediate form of the original SystemC code. SystemC bytecode is a combination of MIPS-like assembly level language instructions and special SystemC instructions, which retains all spatial and timing information from the original SystemC application.

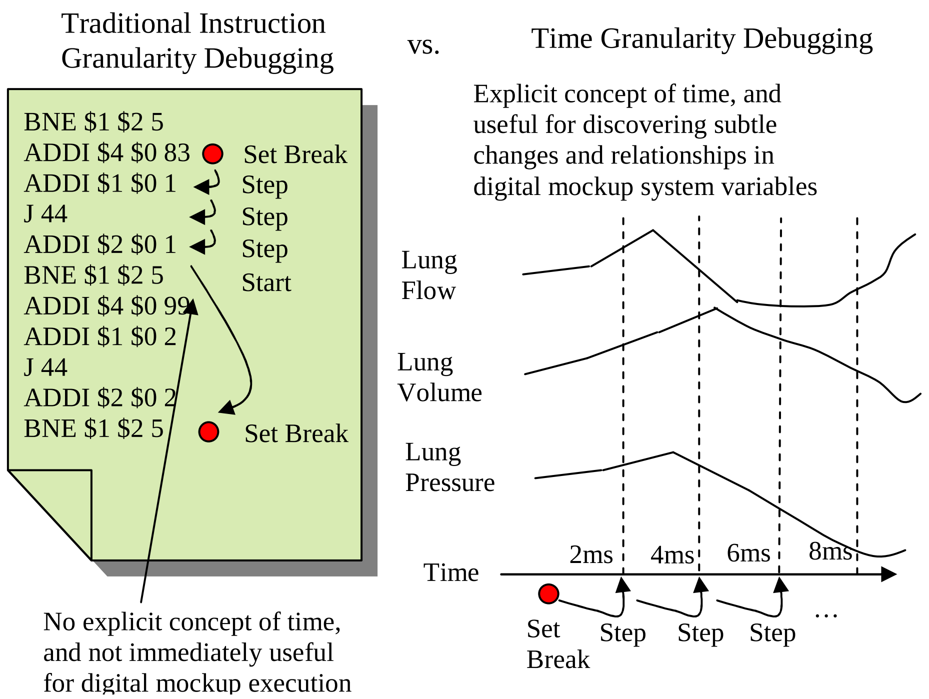
SystemC bytecode is supported by the *SystemC emulation engine,* shown in Figure 4(c). The SystemC emulation engine can run on any development platform that supports a basic interface of a number of different peripherals, memories, and internal components. The SystemC emulation engine’s core is a *SystemC emulation kernel*. The SystemC emulation kernel consists of a lean event-driven kernel, a virtual machine to execute the SystemC bytecode instructions, and hooks and access to the development platform’s peripheral set. The lean event kernel continually processes a series of ready-to-run *events.* An *event* is placed on a queue when a signal value is updated and that signal is on the sensitivity list of a process. Each time step might consist of multiple *delta* time steps, in which a process may execute multiple times during a time step. After each delta step, the event kernel updates the signal values, and places any new sensitive processes onto the event queue. The event-driven kernel calls a *bytecode virtual machine* to execute each event in the event queue. The *bytecode virtual machine* supports the SystemC bytecode instruction set. Each process is allocated an instruction memory, register file, and local data memory. The virtual machine also contains proper hooks to communicate with the standard peripheral and I/O set. The emulation engine supports platform I/O and peripheral access. The set of peripherals includes buttons, LEDs, UART, and input and output memories. The basic emulation engine supports SystemC descriptions that implement the interface shown in Figure 4(d). The SystemC application writer does not have to follow the standard interface, but the standard interface provides a convenient mapping between description’s signals and the available peripherals. More advanced platforms might choose to support a greater range of input and output peripherals. For instance, a SystemC-on-a-Chip framework specialized for digital mockup execution might support a large number of one-way serial connections for easy interfacing to existing medical devices.

Additionally, the SystemC emulation engine optionally supports a portable USB download interface, allowing a designer to download a SystemC application (the bytecode version) to a SystemC emulation engine by simply inserting a USB stick into the platform. The SystemC emulation engine is responsible for running the SystemC bytecode, preserving the correct spatial and timing information.

5. TIME-CONTROLLABLE DIGITAL MOCKUP EXECUTION

The SystemC-on-a-Chip framework can be augmented to give the developer unobtrusive *time-granularized* debug and test capabilities. In contrast to the standard *instruction granularity* debugging approaches, the SystemC-on-a-Chip framework can start, stop, and step a digital mockup's simulated time, advancing time forward as slow or fast as the developer requires. Figure 5 highlights the differences between instruction level and time granularity debugging.

Figure 5: Time-Controllable Debugging. In contrast to traditional instruction granularity debugging, time granularity debugging allows a developer to monitor system variables by explicitly controlling simulated time.



Traditional Instruction Granularity Debugging
vs.
Time Granularity Debugging
BNE $1 $2 5
ADDI $4 $0 83
ADDI $1 $0 1
J 44
ADDI $2 $0 1
BNE $1 $2 5
ADDI $4 $0 99
ADDI $1 $0 2
J 44
ADDI $2 $0 2
BNE $1 $2 5
Set Break
Step
Step
Step
Start
Set Break
No explicit concept of time, and not immediately useful for digital mockup execution
Explicit concept of time, and useful for discovering subtle changes and relationships in digital mockup system variables
Lung Flow
Lung Volume
Lung Pressure
Time
2ms
4ms
6ms
8ms
Set Break
Step
Step
Step
…

Time-controllable SystemC emulation possesses a number of advantages for digital mockup execution. First, the medical device software developer can control time by running simulations between the digital mockup and medical device faster than real-time. Running faster than real-time might allow a developer to simulate a night’s worth of breathing in just a few hours, or make possible the ability to test several different control algorithms on the medical device in a timely manner. The ability to run faster than real-time is of course determined by the delta time step at which the digital mockup is executing and how powerful the underlying platform is, but for many examples, running faster than real-time is feasible.

Another advantage is the debugger can step through the execution of the digital mockup at the level of time granularity the digital mockup computes. Stepping using an explicit notion of time might allow a medical device software developer to step through a simulated cough of a digital lung mockup, a heart murmur in a digital heart mockup, or other anomalies and subtleties that might not otherwise be seen, or easily observed, executing at faster speeds.

6. EXPERIMENTS

We conducted several experiments to test the feasibility of capturing digital mockups using SystemC, interfacing those models using the SystemC-on-a-Chip framework to a medical device, and testing the ability to control time by configuring faster than real-time execution and incrementally stepping through time. We built a SystemC-on-a-Chip framework to run on a Xilinx Virtex5 FPGA platform. We wrote the SystemC-on-Chip framework in approximately 20,000 lines of C, C++, and VHDL. The main emulation kernel was built on top of a Xilinx Microblaze processor, with custom bytecode accelerators [Sirowy] built on the native FPGA fabric for increased performance. We also built SystemC-on-a-Chip frameworks for a Xilinx Virtex4 Ml403 platform, and a Xilinx Spartan 3E platform. The Virtex4 implementation was built on top of a PowerPC-based system. All of the SystemC-on-a-Chip implementations could execute the same SystemC bytecode without recompiling for any particular platform.

We described a number of physiological models in SystemC that we obtained from the NSR Physiome Project. Figure 6 shows a portion of the SystemC code used to capture a two-compartmental respiratory system, one bronchial compartment and one alveolar compartment. The respiratory system model computes airway pressure, lung pressure, flow, and volume values for a healthy human lung at a simulated time step of approximately 4 milliseconds. The respiratory system was modeled using a series of four

Figure 6: SystemC Implementation of a two-compartment respiratory system digital mockup.

```
#include "systemc.h"

template<int bit = 32>
class integrator : public sc_module {
  sc_in_clk clock;
  sc_in<sc_uint<32> > dt;
  sc_in<sc_uint<32> > funct;
  sc_out<sc_uint<32> > out;

   sc_signal<sc_uint<32> > reg;

  integrator( sc_module_name n ) sc_module (n)
{
    sc_method(process);
    sensitive << clock;
  }
  void process(void) {
    reg = funct.read() * dt.read() + reg;
    out.write(reg);
  }
};

class model : public sc_module {
  sc_in_clk clock;
  sc_in<sc_uint<32> > qalv, valv, qbr, vbr;
  sc_out<sc_uint<32> > qalv_t, valv_t;
  sc_out<sc_uint<32> > qbr_t,    vbr_t;

  sc_signal<sc_uint<32> > pbr, palv, fbr;
 sc_signal<sc_uint<32> > falv, cbr, calv;

  model( sc_module_name n ) : sc_module(n)   {
    SC_METHOD(pbr_func);
    sensitive << clock;
    //…
    SC_METHOD(qalv_t_func);
    sensitive << clock;
  }

  void pbr_func(void) {
    int COM_BR = 0x100;
    int VBR_0 = 0x9600;
    pbr = vbr.read() - VBR_0 / COM_BR;
  }

  //…

  void qalv_func(void) {
    qalv_t.write(falv * (cbr + calv));
  }
};
```

```
class top : public sc_module {
  sc_in_clk clock;
  sc_in<sc_uint<4> > buttons;
  sc_in<sc_uint<32> > memory_in;
  sc_in<sc_uint<8> > uart_rx;
  sc_out<sc_uint<8> > uart_tx;
  sc_out<sc_uint<32> > fb_h;
  sc_out<sc_uint<32> > fb_v;
  sc_out<sc_uint<32> > fb_data;
  sc_out<sc_uint<4> > leds;

  sc_signal<sc_uint<32> > Qbr_t, Qalv_t;
  sc_signal<sc_uint<32> > Vbr_t, Valv_t;
  sc_signal<sc_uint<32> > Qbr, Qalv;
sc_signal<sc_uint<32> > Vbr, Valv;
  sc_signal<sc_uint<32> > dt;

  model model_1;
  integrator<32> integrator_Qalv;
  integrator<32> integrator_Qbr;
  integrator<32> integrator_Valv;
  integrator<32> integrator_Vbr;

  top( sc_module_name n ) : sc_module(n)
  {
    dt.write(0x1);

    model_1->clock(clock);
    model_1->qalv(Qalv);
    model_1->qbr(Qbr);
    model_1->valv(Valv);
    model_1->vbr(Vbr);
    model_1->qalv_t(Qalv_t);
    model_1->qbr_t(Qbr_t);
    model_1->valv_t(Valv_t);
    model_1->vbr_t(Vbr_t);

    integrator_Qalv->clock(clock);
    integrator_Qalv->dt(dt);
    integrator_Qalv->funct(Qalv_t);
    integrator_Qalv->out(Qalv);

    //…

    integrator_Vbr->clock(clock);
    integrator_Vbr->dt(dt);
    integrator_Vbr->funct(Vbr_t);
    integrator_Vbr->out(Vbr);
  }
};
```

ordinary differential equations, and nine linear equations. We modeled the respiratory system using approximately 400 lines of behavioral SystemC. The SystemC description compiled to approximately 500 lines of SystemC bytecode, and compiled through the SystemC bytecode compiler in less than a second.

We executed the digital respiratory mockup on the Xilinx Virtex5 implementation of the SystemC-on-a-Chip development platform. At full speed, the SystemC-on-a-Chip platform could execute a full simulated time step in 1.6 milliseconds, or about 3X faster than real-time. We also modeled an alternate implementation of a lung that computes concentration, lung mass, flow, bronchial pressure, and alveolar pressure. The system consisted of four equations, one of which was an ordinary differential equation. We modeled the system using 600 lines of structural SystemC. The SystemC bytecode compiler compiled the model to approximately 300 lines of SystemC bytecode. While the model computed fewer equations than the previous model, the SystemC-on-a-Chip framework took longer to compute one time step because the model was captured structurally with more interconnected processes. Figure 7 summarizes the models.

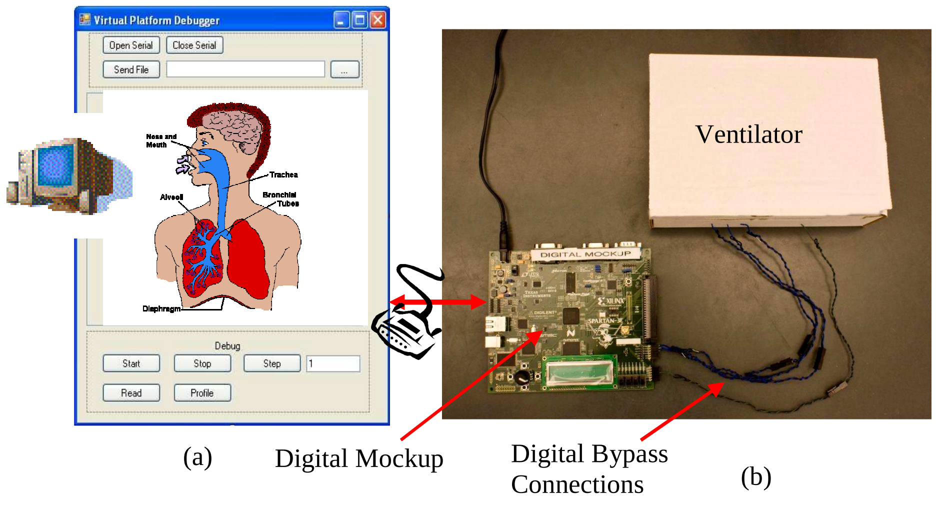
Figure 8 illustrates one of our prototype setups for a ventilator and the respiratory system digital mockup. The digital mockup communicates to the ventilator through four dedicated serial connections and one synchronization channel. The dedicated serial connections bypass the ventilator's airway pressure, lung pressure, flow, and volume transducers. The synchronization channel is used to ensure that both models are sampling at the same frequency. Since the digital mockup can simulate time 3X faster than real-time when running on the virtualized platform, the medical device and digital mockup use the synchronization channel to agree on a rate at which both devices operate [Sirowy]. The rate at which the devices operate is user-defined by a separate PC-based debug interface, and shown in Figure 8(a).

We tested the usefulness of the time controllability of the test platform by developing a prototype PC-based debugging application. The debugger is able to stop, start, and advance time at the smallest simulated time rate the digital mockup can achieve (approx.

Figure 7: SystemC Digital Mockup Implementation Summary. Both respiration models were obtained from the NSR Physiome Project and manually converted to concurrently executing SystemC implementations.

Digital Mockup	# of Eqns.	# of ODEs	SystemC LOC	Simulate Dt	Simulated Freq
Alveolar Bronchial Lung w/ Gas Exchange	13	4	430 (Behavioral)	2^{-8} s	~800 Hz
First Order Non-Linear Lung	4	1	570 (Structural)	2^{-8} s	~600 Hz

Figure 8: Medical device(ventilator) and digital mockup(lung) prototype setup. (a)The digital mockup can be time-controlled using a simple PC-based debug interface. (b)The digital mockup and ventilator communicating digitally.



4 milliseconds). Figure 8(b) shows that even with a simple debugging interface we can step through several steps of lung breathing, monitor pressures, volumes, and gas concentrations, and also make sure the ventilator software is performing correctly. The time-controllable debug commands given to the digital mockup propagate to the ventilator via the synchronization channel.

7 CONCLUSIONS

Developing medical device software by interfacing with a digital mockup enables development without costly or dangerous physical mockups, and enables execution that is faster or slower than real-time. Developing digital mockups in SystemC has the added advantages that the description closely models the high level mathematical and physical model, can be tested extensively with freely available SystemC support libraries, and can interface to real medical device software through the use of the SystemC-on-a-Chip framework. The SystemC-on-a-Chip framework enables *time-controllable* debug features, making possible the ability to step through a digital mockup's execution through simulated time. We tested the feasibility of such an approach by modifying the existing SystemC-on-a-Chip framework to support time-controllable debug, and also tested multiple respiratory digital mockup examples. We currently are modifying a commercial ventilation system to interact with SystemC-based digital mockups.

REFERENCES

BENINI, L., BERTOZZI, D., BRUNI, D., DRAGO, N., FUMMI, F., AND PONCINO, M. 2003. SystemC Cosimulation and Emulation of Multiprocessor SoC Designs. */Computer/* 36, 4 (Apr. 2003), 53-59.

BENINI, L., BRUNI, D., DRAGO, N., FUMMI, F., AND PONCINO, M. "Virtual in-circuit emulation for timing accurate system prototyping," in *Proc. IEEE Int. Conf. ASIC/- SoC*, 2002

BOTROS, N., AKAABOUNE, M., ALGHAZO, J., AND ALHREISH, M. 2000. Hardware Realization of Biological Mechanisms Using VHDL and FPGAs

CHANG, C., KUUSILINNA, K., RICHARDS, B., AND BRODERSEN, R. W. 2003. Implementation of BEE: a real-time large-scale hardware emulation engine. In *Proceedings of the 2003 ACM/SIGDA Eleventh international Symposium on Field Programmable Gate Arrays* (Monterey, California, USA, February 23 - 25, 2003). FPGA '03. ACM, New York, NY, 91-99

FORNACIARI, W. AND PIURI, V. Virtual FPGAs: Some Steps Behind the Physical Barriers. In Parallel and Distributed Processing (IPPS/SPDP'98 Workshop Proceedings), LNCS. 1998

GENKO, N., ATIENZA, D., MICHELI, G. D., MENDIAS, J. M., HERMIDA, R., AND CATTHOOR, F. 2005. A Complete Network-On-Chip Emulation Framework. In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1* (March 07 - 11, 2005). Design, Automation, and Test in Europe. IEEE Computer Society, Washington, DC, 246-251

GRUIAN, F. AND WESTMIJZE, M. 2007. BlueJEP: A flexible and high-performance Java embedded processor. In *Proceedings of the 5th international Workshop on Java Technologies For Real-Time and Embedded Systems* (Vienna, Austria, September 26 - 28, 2007). JTRES '07, vol. 231. ACM, New York, NY, 222-229

IUPS PHYSIOME PROJECT. http://www.physiome.org.nz/

KOPETZ, H. AND OCHSENREITER, W. 1987. Clock synchronization in distributed real-time systems. *IEEE Trans. Comput.* 36, 8 (Aug. 1987), 933-940

LAMPORT, L. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (Jul. 1978), 558-56

LEE, E. Computing Needs Time. Communications of the ACM. May 2009. Vol 52. Number 5.

LEE, I. DAVIDSON, S., AND WOLFE, V. Motivating Time as a First-Class Entity. Technical Report MS-CIS-87-54. Department of Computer and Information Science. University of Pennsylvanis, Philadelphia, PA. Aug 1987.

LEVIS, P. AND CULLER, D. 2002. Maté: a tiny virtual machine for sensor networks. *SIGOPS Oper. Syst. Rev.* 36, 5 (Dec. 2002), 85-95

MEDGADGET INTERNET JOURNAL. 2008. Supercomputer Creates Most Advanced Heart Model.
http://medgadget.com/archives/2008/01/worlds_biggest_heart_model_simulated_1.html

MICHIGAN INSTRUMENTS. Training and Test Lung (TTL) and PneuView software, http://www.michiganinstruments.com/resp-ttl.htm, 2009

NAKAMURA, Y., HOSOKAWA, K., KURODA, I., YOSHIKAWA, K., AND YOSHIMURA, T. 2004. A fast hardware/software co-verification method for system-on-a-chip by using a C/C++ simulator and FPGA emulator with shared register communication. In *Proceedings of the 41st Annual Conference on Design Automation* (San Diego, CA, USA, June 07 - 11, 2004). DAC '04

NSR PHYSIOME PROJECT. http://nsr.bioeng.washington.edu

PARNIS, J. AND LEE, G. 2004. Exploiting FPGA concurrency to enhance JVM performance. In *Proceedings of the 27th Australasian Conference on Computer Science - Volume 26* (Dunedin, New Zealand). Estivill-Castro, Ed. ACSC, vol. 56. Australian Computer Society, Darlinghurst, Australia, 223-232

PIMENTEL, J. AND TIRAT-GEFEN, Y. 2006. Hardware Acceleration for Real-time Simulation of Physiological Systems. EMBS. pp 218-223

PIMENTEL, J. AND TIRAT-GEFEN, Y. 2006. Real-Time Simulation of Physiological Systems. Proceedings of the IEEE 32nd Annual Northeast Bioengineering Conference. pg. 159-160

RISSA, T., DONLIN, A., AND LUK, W. 2005. Evaluation of SystemC Modelling of Reconfigurable Embedded Systems. In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 3* (March 07 - 11, 2005). Design, Automation, and Test in Europe. IEEE Computer Society, Washington, DC, 253-258

SHIN, H. AND GEORGE, S. Impact of Axial Diffusion on Nitric Oxide Exchange in the Lungs. Journal of Applied Physiology. 2002

SIROWY, S., GIVARGIS, T. AND VAHID, F. Digitally-Bypassed Transducers: Interfacing Digital Mockups to Real-Time Medical Equipment. IEEE Engineering and Biology Society (EMBS). 2009. Minneapolis.

SIROWY, S. MILLER, B., AND VAHID. F. SystemC-on-a-Chip. 2009. International Conference on Codesign and Synthesis (CODES). Grenoble, France.

SIROWY, S. SHELDON, D., GIVARGIS, T. AND VAHID. F. Virtual Microcontrollers. ACM Sigbed Review. 2008

SMITH, J. AND NAIR, R. VIRTUAL MACHINES: Versatile Platforms for Systems and Processes. Morgan-Kaufman Publishers. 2005

STARK, R., SCHMID, J, AND BORGER, E. Java and the Virtual Machine- Definition, Verification, and Validation. 2001

SYSTEMC. http://www.systemc.org

TAWHAI, M., AND BEN-TAL, A. 2004. Multiscale Modeling for the Lung Physiome. Cardiovascular Engineering: An International Journal, Vol. 4, No. 1., March 2004. pp 19-26

VMWARE. http://www.vmware.com

XEN. http://www.xen.org