International Journal of Modeling, Simulation,
and Scientific Computing
Vol. 4, No. 2 (2013) 1340005 (26 pages)

DOI: 10.1142/S1793962313400059

World Scientific
www.worldscientific.com

HIGH FIDELITY VIRTUALIZATION OF CYBER-PHYSICAL SYSTEMS

YU ZHANG*,†,‡, FEI XIE†, YUNWEI DONG*,
GANG YANG* and XINGSHE ZHOU*

*School of Computer Science
Northwestern Polytechnical University
Xi'an, 710072, P. R. China

†Department of Computer Science
Portland State University
Portland, Oregon 97207, USA
‡yuzhang.nwpu@gmail.com

Received 1 April 2013
Accepted 4 April 2013
Published 19 June 2013

Cyber-physical systems (CPS) tightly integrate cyber and physical components and transcend discrete and continuous domains. It is greatly desired that the synergy between cyber and physical components of CPS is explored even before the complete system is put together. Virtualization has potential to play a significant role in exploring such synergy. In this paper, we propose a CPS virtualization approach based on the integration of virtual machine and physical component emulator. It enables real software, virtual hardware, and virtual physical components to execute in a holistic virtual execution environment. We have implemented this approach using QEMU as the virtual machine and Matlab/Simulink as the physical component emulator, respectively. To achieve high-fidelity between the real system and its virtualization, we have developed a strategy for synchronizing the virtual machine and the physical component emulator. To evaluate our approach, we have successfully applied it to real-world control systems. Experiments results have shown that our approach achieves high-fidelity in capturing dynamic behaviors of the entire system. This approach is promising in enabling early development of cyber components of CPS and early exploration of the synergy of cyber and physical components.

Keywords: Cyber-physical systems; virtualization; virtual machine; co-simulation.

1. Introduction

Cyber-physical systems (CPS) are engineered systems that are built from and depend upon the synergy of cyber and physical components.[1] CPS engineering must account for the interacting and interdependent behaviors of both types of components. Representative application domains of CPS include medical devices

and systems, automobiles, robotics, avionics, and other critical infrastructures. Due to the criticality of CPS, they are often required to be high-confidence.

CPS are generally difficult to analyze, design and validate because of the following two main reasons:

- In contrast to the traditional embedded systems view where the focus tends to be more on the cyber components, CPS emphasize the holistic system view over both the cyber and physical components. CPS engineering must account for the interacting and inter-dependent behaviors of cyber and physical components to achieve system level functionalities;
- CPS conjoin two different semantic domains: the continuous dynamics of the physical components (often modeled by differential equations) and the discrete dynamics of cyber components (often modeled in discrete mathematics). These different semantics make the integration of these components and also their abstractions a major challenge.

The above two reasons significantly complicate analysis of CPS. For certain components of CPS, particularly physical components, the only abstractions available are usually simulation models, which makes complete formal analysis impractical and simulation the key method for validating these systems. The need for effective simulation techniques for CPS integrating cyber and physical components has been recognized for some time. Most current techniques abstract cyber components into state machines running in parallel with the simulation models of physical components. The omission of programming-language level details of cyber components make it hard to achieve holistic validation of the complete CPS and detect the implementation-level errors of cyber components.

In order to address this drawback, simulation techniques need to model the full closed-loop system including both the cyber components in implementation level and physical components. It is greatly desired that the synergy of cyber and physical components is explored even before the complete system is put together, for instance, the cyber components can be developed and evaluated even before the physical components are manufactured. Virtualization has the potential to play a significant role in exploring the synergy between the cyber and physical components of CPS. Recently, virtual machine has been increasingly used in the analysis, design, verification and deployment of computer systems, particularly embedded systems. It enables early software development even before silicon prototypes become available. A notable example is how Intel used virtual devices to enable early driver development for their 40 Gigabit Ethernet adapter before the device became available.[2] However, virtual machine lacks the capabilities to capture the continuous dynamics of physical components and enable holistic CPS visualization.

In this paper, we present a CPS virtualization approach based on the integration of virtual machine and physical component emulator. Virtual machine emulates the hardware components such as the processor, bus, and peripheral devices, etc. It supports execution of a full-fledged operating system such as Linux and

application-specific software components such as software controllers. Physical component emulator provides the virtualization of physical components such as sensors, actuators, and control plants. Their integration enables real software, virtual hardware and virtual physical components to execute in a holistic virtual execution environment.

Central to this integration is the synchronization between the virtual machine and the physical component emulator, which greatly affects fidelity and efficiency of virtualization. Although differential equations modeling physical components are solved with approximations, the events generated in the virtual machine and the physical component emulator are quite different. To synchronize these events, we designed a synchronization strategy in which we unify the essential synchronization events along a synchronization clock. With this strategy, we only synchronize the emulators on the events that affect the system behaviors, to avoid unnecessary overheads.

We have implemented this approach using QEMU[3] as the virtual machine and Matlab/Simulink[4] as the physical component emulator, respectively. And we have also implemented the synchronization strategy. To evaluate our approach, we have successfully applied it to real-world control systems. Experiment results show that our approach can achieve high-fidelity in capturing system dynamic behaviors efficiently.

This paper makes the following contributions. (1) We present a CPS virtualization framework, which enables real software, virtual hardware and virtual physical components to execute in a holistic virtual execution environment. (2) We have developed a synchronization strategy between the virtual machine and physical component emulator, for tight integration of virtualized cyber and physical components. This approach is promising in enabling early development of cyber components of CPS and exploration of the synergy between cyber and physical components.

The reminder of this paper is organized as follows. Section 2 reviews related work. Section 3 introduces the background of this paper. Sections 4 and 5 present the design and implementation of our approach. Section 6 elaborates on case studies we have conducted and discusses the experimental results. Section 7 concludes the paper and discusses future work.

2. Related Work

2.1. *Co-simulation*

The CPS concept transcends embedded systems and hybrid systems. Embedded systems research has been largely focused on hardware and software and, particularly, their interactions. Hybrid systems research has been largely focused on the interactions between discrete and continuous domains. CPS research takes a comprehensive view of the whole system and, instead of focusing on a single semantic gap, it focuses on multiple semantic gaps simultaneously.

In embedded systems research, closely related to CPS simulation is HW/SW co-simulation. Co-simulation is low-cost and efficient in detecting shallow bugs. There has been much research[5–9] on co-simulation that has led to industrial tools such as Seamless.[10] For hybrid systems, continuous components are typically modeled with tools such as LabView,[11] Mathematica,[12] Matlab/Simulink and Modelica.[13] These tools are interfaced to the simulators for discrete components such as ModelSim.[14]

CPS co-simulation is based on HW/SW co-simulation by utilizing the above tools. The most closely related work is presented in Ref. 15, where a methodology and an open toolset for the virtual prototyping of CPS were proposed. The focus was on integration of tools while the synchronization issues and fidelity evaluation are not explicitly addressed. In Ref. 16, a comprehensive co-simulation platform for CPS and examples showing the capabilities of the platform were presented. The simulation platform is built on Modelica and ns-2 tools. Modelica is used to simulate software and physical components. Software components are simulation models in high level. This makes it difficult for developers to observe and verify system on the implementation level. In addition, there is no guarantee about the consistency between the simulation model and their implementation.

2.2. *Virtual machine*

The other research closely related is the virtual machine. A virtual machine is a software implementation of a machine (i.e., a computer) that executes programs like a physical machine. The virtual machine research has been largely focused on hardware and software and, particularly, their interactions, which lacks ability to capture the physical dynamics. In Ref. 17, an approach to constructing a virtual platform by integrating hardware models in SystemC into the QEMU virtual machine is proposed. It can be used to facilitate the co-design of hardware models and device drivers at the early stage of electronic system level design flow. In Ref. 18, DDT was proposed for testing closed-source binary device drivers against undesired behaviors. DDT combines virtualization with a specialized form of symbolic execution to thoroughly exercise tested drivers. However, CPS research requires taking a comprehensive view of the whole system of software hardware and physical.

3. Background

In this section, we introduce three related concepts: QEMU, Matlab/Simulink and synchronous languages. We use QEMU as virtual machine in our implementation, and Matlab/Simulink to simulate physical dynamics. Our integration of QEMU and Matlab/Simulink is based on the synchronous language principle.

3.1. *QEMU*

QEMU is an open source system emulator and provides the flexibility of developing customized virtual machine environment. QEMU emulates several different

processor architectures, such as x86, SPARC and ARM. It relies on dynamic binary translation to achieve a reasonable speed. It also provides a set of virtual devices, such as Industry Standard Architecture (ISA) devices, PCI devices and USB devices. A virtual device is essentially a software implementation of a device integrated into QEMU. The communication between the processor emulator and the virtual device is done via registered callback functions for the corresponding memory regions on the system bus.

We illustrate the virtual device concept with the 16550A UART (universal asynchronous receiver/transmitter), an ISA device. As shown in Fig. 1, the virtual device has the following components:

- ISA device state, as defined by 16550A UART, which keeps track of the state of the ISA device;
- functions simulating basic behaviors of the device: functions `serial_mm_write` and `serial_mm_read` simulate how 16550A UART respond when the driver issues I/O command; function `serial_update_irq` simulates how 16550A UART respond when an interrupt is generated.

```
1  typedef struct ISASerialState {
2      ISADevice dev;
3      uint32_t index;
4      uint32_t iobase;
5      uint32_t isairq;
6      SerialState state;
7  } ISASerialState;
8
9  static void serial_mm_write(void *opaque,target_phys_addr_t addr,
       uint64_t value, unsigned size)
10 {
11     ...
12     serial_ioport_write(s, addr>>s->it_shift, value);
13     ...
14 }
15
16 static uint64_t serial_mm_read(void *opaque,target_phys_addr_t
       addr, unsigned size)
17 {
18     ...
19     return serial_ioport_read(s, addr >> s->it_shift);
20 }
21
22 static void serial_update_irq(SerialState *s)
23 {
24     uint8_t tmp_iir = UART_IIR_NO_INT; /* No interrupts pending */
25     ...
26     if (tmp_iir != UART_IIR_NO_INT) {
27         qemu_irq_raise(s->irq);
28     } else {
29         qemu_irq_lower(s->irq);
30     }
31 }
```

Fig. 1. QEMU virtual device code structure.

3.2. *Matlab/Simulink*

Physical components are often modeled using differential and integral calculus with continuous semantics. Matlab/Simulink is a powerful numerical computing and simulation environment for multi-domain dynamic systems, which enables effectively solving the differential and integral equations.

Simulink supports extension via well-defined S-function (system-functions) interfaces. S-functions are written in Matlab, C, C++, or Fortran. We illustrate the S-function simulation flow as shown in Fig. 2. An S-function block flow includes a simulation loop. In each iteration of the simulation loop, Matlab first determines the next sampling time, then calculates the output of the block, updates each discrete variable, and finally calculates the differential equations with suitable approximation solvers like Runge–Kutta and updates the output simultaneously.

3.3. *Synchronous languages*

Synchronous languages such as Esterel[19] and Lustre[20] have been developed to describe close-loop control systems. These systems interact continuously with their environment in terms of a discrete sequence of reaction steps, at a speed imposed by the environment.[21]

Execution of programs in synchronous languages follows instants of a global clock. The operational semantics of these languages is defined by so-called micro

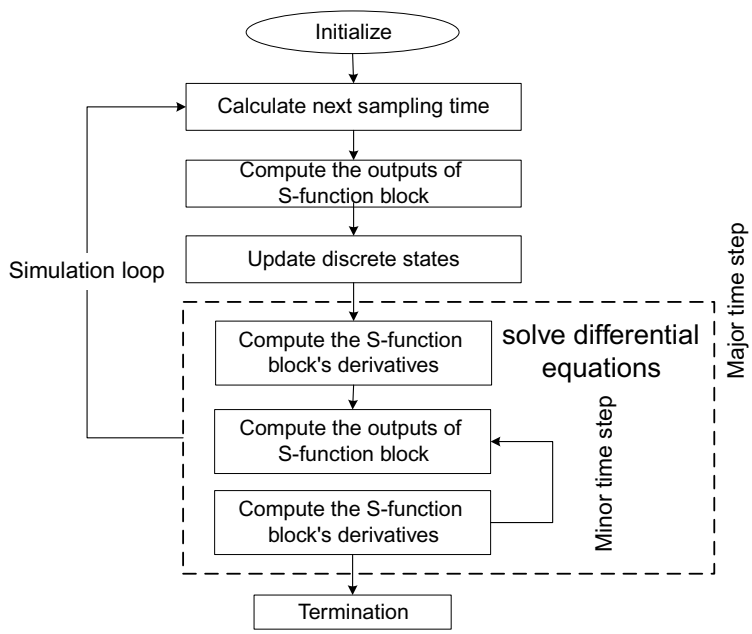

Fig. 2. S-function simulation flow.

and macro steps, where a macro step consists of finitely many micro steps. Macro steps correspond with reaction steps of a control system, and micro steps correspond with atomic actions. Variables of a synchronous program are synchronously updated between macro steps, so that the execution of the micro steps of one macro step is done in the same variable environment of their macro step.

The synchronous language abstraction does not only lead to a convenient programming model for control systems as well as a simplified estimation of worst-case reaction times. It is also the key to a compositional formal semantics which is necessary for simulation and verification. Hence, synchronous languages lend themselves to the development of embedded control systems.

4. Design of CPS Virtualization

In this section, we discuss the framework for CPS virtualization. CPS include not only software and hardware components, but also physical components. The holistic virtual execution environment is shown in Fig. 3.

The virtual machine emulates the hardware components and provides a virtual platform for executing the software controller. And we model physical components with differential equations and compute them with suitable approximation solvers in the physical component emulator. Then we integrate the virtual machine and the physical component emulator to virtualize the entire CPS. The software controller executes on the virtual machine and, through it, interacts with the virtualized physical components. In what follows, we characterize the dynamic of the main components in detail and discuss how their integration is handled.

4.1. *Cyber components*

Cyber components include both software and hardware components. The virtual machine emulates hardware components such as the processor, bus, and peripheral

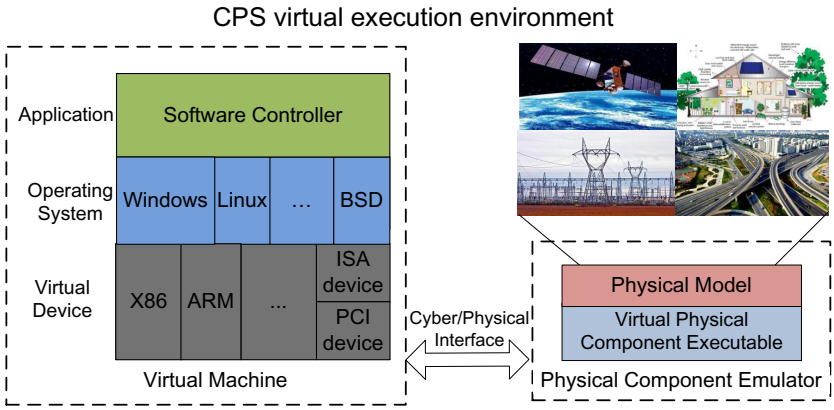

Fig. 3. Architecture of CPS virtualization.

devices, etc. It supports execution of a full-fledged operating system such as Linux and application-specific modules for a software controller. We maximize realism by running the software controller in the complete environment: libraries, kernel, drivers, etc. Within a real software stack, the control program can read/write files, send/receive network packets or be preempted/blocked by the operating system. We capture all interactions of the software controller with its surrounding system, not just with a simplified abstraction of that system. The software controller implements the desired sequences of actions according to the control algorithm. Figure 4 shows an example software controller.

Software components represent sequences of causally related actions. This means the execution of software controller follows the sequence of events, whenever these events occur. We associate each statement in software with a temporal event. We define the temporal event as a tuple (n, t), where $n \in \mathbb{N}$ is a natural number, and $t \in \mathbb{R}$ is a real-time stamp. The tuple (n, t) identifies an ordered sequence number n and a corresponding real-time instant t when the software actions are taken. Let μ as a partial function defined on the tuple (n, t). So our time model of software components has the form,

$$\mu : \mathbb{N} \times \mathbb{R} \to \mathbb{R}.$$

We also assume a lexical ordering on the tuple (n, t): $(n_1, t_1) \leq (n_2, t_2) \Leftrightarrow t_1 \leq t_2 \vee (t_1 = t_2 \wedge n_1 \leq n_2)$. This allows software components to have a real-time stamp at each action. At shown in Fig. 4, when initializing the controller, the software will set a timer to invoke the control loop periodically. After that, the controller will wait to invoke the control loop. In the control loop, the controller first reads sensor values from related channels (line 5, read sensor event E_{state}), then calculates outputs according to the control algorithm, writes commands to relevant actuators (line 15, write command event E_{comm}) and updates states in the end. Although we associate a temporal event with each statement, we still do not know exact time t when the software actions are taken. Actually, the exact occurring time instant $\mu(n, t)$ of most software actions do not affect the whole system behavior. We focus on the sequence of actions n instead of exact occurring time instant t for these events. For the other temporal events (such as E_{state} and E_{comm}), we will associate with exact real-time stamp t. We will discuss these in detail in Sec. 4.4.1.

4.2. *Physical components*

We describe physical components evolving on a continuous-time basis, typically as a mathematical model. The mathematical model can be represented by a set of first-order differential equations on a vector-valued state. The most general state-space representation of a linear system with inputs, outputs and state variables is written in the following form:

$$\begin{cases} \dot{x}(t) = g(x(t), u(t), t) \\ y(t) = f(x(t), u(t), t), \end{cases} \tag{1}$$

```
1  int ReadSensors(SensorReadings_t *sensors)
2  {
3    ...
4    /*associate with read sensor event E_state*/
5    if((result = dscADScan(dscb, &dscadscan, samples))!= DE_NONE){
6      free(samples);
7    }
8    ...
9  }
10
11 void commandMotor(double *v)
12 {
13   ...
14   /*associate with write command event E_comm*/
15   if((result = dscDAConvert(dscb, pos_channel, pos_counts)) !=
         DE_NONE){
16     dscGetLastError(&errorParams);
17   }
18   ...
19 }
20
21 int main(int argc, char **argv, char **envp){
22   ...
23   initial();
24   ...
25   while(TSrunning){
26     ...
27     waitClock();
28     /* Read the raw sensor values */
29     if (!readSensors(&SensorReadings)) {
30         printf("A/D error in ReadSensors -- aborting\n");
31         return;
32     }
33     ...
34     calculateOutput();
35     ...
36     /* Actually fire off the motors */
37     commandMotor(Actuator.FanVoltage);
38     ...
39     updateState();
40   }
41   ...
42 }
```

Fig. 4. Control software example.

where $x(t) \in \mathbb{R}^n$, $y(t) \in \mathbb{R}^m$, $u(t) \in \mathbb{R}^p$ and $t \in \mathbb{R}$ are state, output, input vectors and real time. The function, $g : \mathbb{R}^n \times \mathbb{R}^p \times \mathbb{R} \to \mathbb{R}^n$ and $f : \mathbb{R}^n \times \mathbb{R}^p \times \mathbb{R} \to \mathbb{R}^m$ are state functions and output functions, respectively. The differential equations must be approximated by a solver. Take classical fourth-order Runge–Kutta method as an example. The fourth-order Runge–Kutta method is one member of the family of Runge–Kutta methods which are iterative methods for the approximation of solutions of ordinary differential equations. For a step-size $h > 0$,

$$x_n = x_{n-1} + \frac{1}{6}h(K_1 + 2K_2 + 2K_3 + K_4), \tag{2}$$

where x_n is the approximation of $x(t_n)$, and the value x_n is determined by the value x_{n-1} plus the weighted average of four increments, where each increment is the product of the size of the interval, h, and an estimated slope specified by function g on the right-hand side of the differential equation,

$$K_1 = g(x_{n-1}, t_{n-1});$$
$$K_2 = g\left(x_{n-1} + \frac{1}{2}hK_1, u\left(t_{n-1} + \frac{1}{2}h\right), t_{n-1} + \frac{1}{2}h\right);$$
$$K_3 = g\left(x_{n-1} + \frac{1}{2}hK_2, u\left(t_{n-1} + \frac{1}{2}h\right), t_{n-1} + \frac{1}{2}h\right);$$
$$K_4 = g(x_{n-1} + hK_3, u(t_{n-1} + h), t_{n-1} + h).$$

The solver provides samples of the continuous dynamics, and typically, to maintain adequate accuracy, must control the time steps between such samples. For example, in order to complete one step h, the solver will evaluate the function g at the intermediate time $t_{n-1} + \frac{1}{2}h$.

4.3. *Cyber/physical interface*

Cyber components and physical components are mostly asynchronous and only transition synchronously when they interact through the cyber/physical interface. The interface is determined by the cyber and physical components that it connects. This enables more flexible composition of cyber and physical components and creation of composite components including both cyber and physical subcomponents. A cyber/physical interface has two parts: interface states and interface events. Interface states, (I_p, I_c), are state variables provided either by cyber or physical and accessible by both. Interface events have two types: cyber or physical. When cyber updates the physical interface states, a cyber interface event occurs (such as when entry stack symbol of `dscDAConvert` is reached), and vice versa.

We illustrate the concept of cyber/physical interface with an example as shown in Fig. 5. The interface is derived from the interfaces of software controller and physical components. The specification of the interface defines:

- What variables in cyber components are mapped to input vectors in physical components, for instance, the `pos_counts` variable is mapped to the input vectors $u(t)$ at time $t_i + d$ in physical components (line 7);
- What output vectors in physical components are mapped to variables in cyber components, for instance, the output vectors, $y(t)$ at time t_i, is mapped to the variable `dscadscan.sample_values` (line 10);
- What is sensor noise characterization in control system, such as mean and variance of the sensor signal (line 13).

```
1  Interface:
2    Ip = {var = {u(ti + d), y(ti)}}
3
4    Ic = {var = {dscadscan.sample_values, pos_counts}}
5
6    /*cyber variable to physical input vectors mapping*/
7    (pos_counts, u(ti + d))
8
9    /*physical output vectors to cyber variable mapping*/
10   (y(ti), dscadscan.sample_values)
11
12   /*sensor noise characterization*/
13   Noise = {Mean, Variance}
```

Fig. 5. A cyber/physical interface example.

4.4. *Synchronization of cyber and physical component emulators*

As already mentioned, physical components in CPS operate in a time continuum, whereas cyber components are composed of discrete, step-by-step actions. Their interaction has to be faithfully modeled to achieve high-fidelity virtualization.

4.4.1. *Timing parameters of a control task*

According to the analysis in Secs. 4.1 and 4.2, we can see that the approximation solver for physical components operates similarly to the virtual machine for cyber components. While the virtual machine uses an event queue to determine the advancement of time for software controller, the physical emulator consults an approximation solver for physical components. It is critical to place discrete controller and continuous physical behavior on a unified temporal semantic basis where it can be proven that the results of simulation are mutually consistent.

We assume the system must react to a stimuli from the environment within hard time bounds. That is, we associate with a fixed constant time between synchronization events. Building on the synchronous language principles, the execution of such a system is a discrete sequence of reaction steps following a unified temporal semantic basis. In each macro step, new inputs are read and corresponding outputs and next states of the system are computed. At each micro step, interface states must be synchronously updated when interface events occur. Otherwise, cyber components and its controlled physical components execute asynchronously in corresponding emulators. The basic timing parameters of a control task are shown in Fig. 6.

Assume that the control task is executed periodically at times given by $t_k = T * k$, where T is the fixed sampling interval of the controller and k is the number of controller iterations. Also assume time bound d for the fixed interval from A/D conversion to D/A conversion. There are two types of synchronization temporal events: read sensor event E_{state} and write command event E_{comm}. We define these temporal events E_{state} and E_{comm} as (n_1, t_{ci}) and $(n_2, t_{ci} + d)$, respectively.

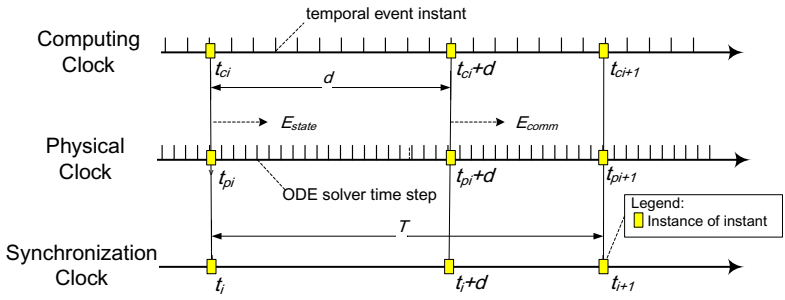

Fig. 6. Timing analysis of control task.

The time $\mu(n_1, t_{ci})$ is earlier than $\mu(n_2, t_{ci} + d)$. The A/D conversion is started by the controller when the entry stack symbol of `dscADScan` is reached (see Fig. 4, line 5) at t_{ci} (triggering E_{state}). Meanwhile, the states of physical components at t_{pi} are sampled by the controller. The controller gets the state data and executes the control logic. When the entry stack symbol of `dscDAConvert` is reached (see Fig. 4, line 15), the control task will send the control command at $t_{ci} + d$ (triggering E_{comm}). Simultaneously, the physical components are controlled at $t_{t_{pi}+d}$. According to the causal dependencies in the control loop, we abstract the essential instants $\{t_{ci}, t_{ci} + d, t_{ci+1}\}$ on the computing clock and $\{t_{pi}, t_{pi} + d, t_{pi+1}\}$ on the physical clock. According to the way how the control task is executed, the time stamps $\{t_{ci}, t_{ci} + d, t_{ci+1}\}$ and $\{t_{pi}, t_{pi} + d, t_{pi+1}\}$ occur pairwise simultaneously. Therefore, we unify these instants on a synchronization clock and identify them as synchronization instants $\{t_i, t_i + d, t_{i+1}\}$ (shown in the bottom horizontal axis of Fig. 6). Synchronization events are then triggered in the emulators on these instants.

4.4.2. *Synchronization protocol*

The holistic simulation is based on the interaction of the virtual machine and the physical component emulator. A single control iteration in the simulation begins at the point the physical component's state is sensed and ends after the plant evolves for a sampling period based on the controller's actions. As shown in Fig. 7, we design the synchronization protocol as follows:

(i) Start the virtual machine and physical component emulator at t_0;
(ii) the scheduler of physical component emulator triggers the earliest future event at time t_i $(i = 0, 1, 2, 3, \ldots)$. The physical component state at t_i is sent to the virtual machine;
(iii) the physical component emulator and the virtual machine will then run asynchronously until $t_i + d$:
    (a) The physical component emulator executes with the latest command until $t_i + d$. Then it pauses at $t_i + d$, waiting for the new command;

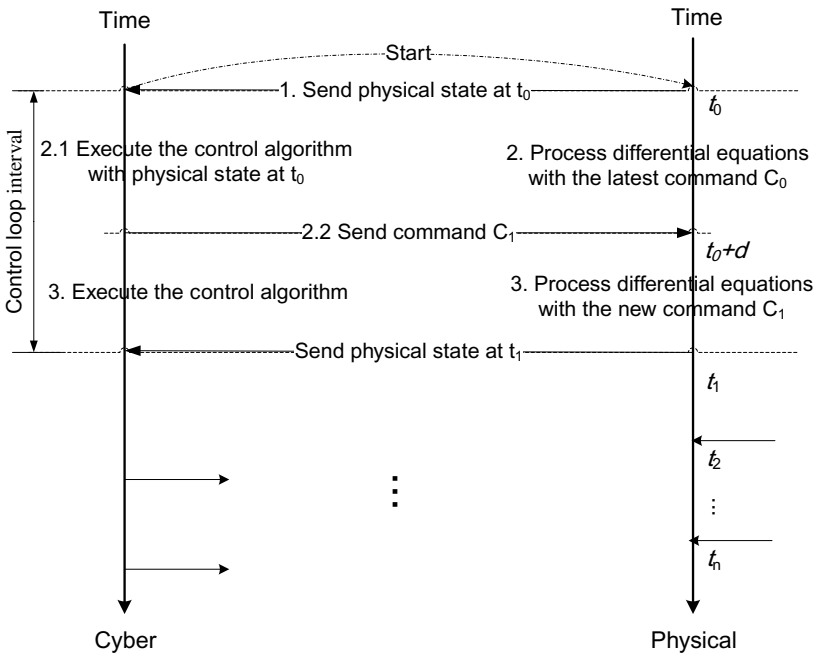

Fig. 7. Synchronization protocol between cyber and physical.

(b) meanwhile, the virtual machine receives the state data at the time t_i, executes the control algorithm, and pauses at the time $t_i + d$, where it sends the new command to the physical component emulator.

(iv) Then the physical component emulator and the virtual machine will run asynchronously during the time interval $(t_i + d, t_{i+1}]$:

(a) The physical component emulator will process differential equations from $t_i + d$ to t_{i+1} with the new command;

(b) meanwhile, the virtual machine will execute the control algorithm to time t_{i+1} and pause there waiting for receiving the physical component state at t_{i+1}.

(v) Repeat steps (ii), (iii) and (iv) until the end of simulation.

The strategy will not miss any essential synchronization event between the virtual machine and physical component emulator. So our virtualization has the potential to achieve high fidelity efficiently.

5. Implementation of CPS Virtualization

We chose QEMU 1.0 as the virtual machine and Matlab 7.13 as the physical component emulator. Inputs of our virtualization include the source code of the controller,

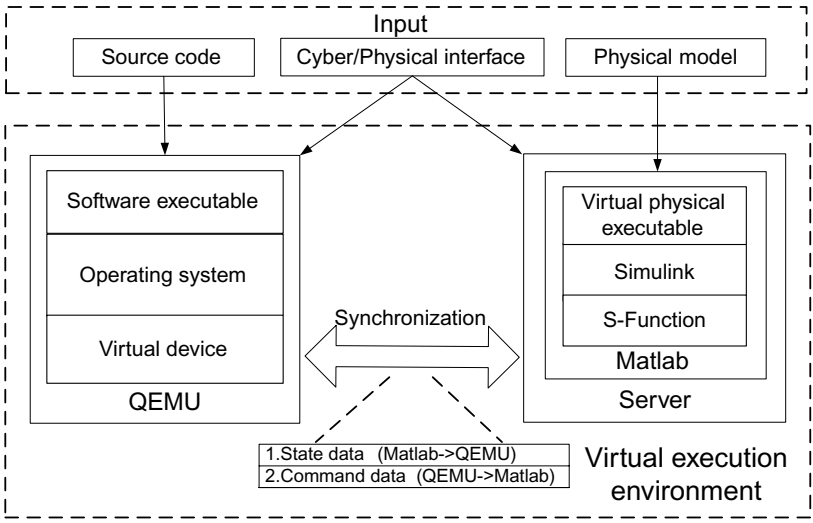

Fig. 8. Implementation of virtual execution environment.

the cyber/physical interface design, and the mathematical models of the physical. Then we instrument the source code and the physical mathematical model based on the cyber/physical interface. Once model instrumentation is done, we can run the software controller and the physical model in our virtual execution environment. Figure 8 shows the implementation of the CPS Virtualization.

We instrument the source code based on the cyber/physical interface and rewrite `dscADScan` and `dscDAConvert` functions. We compile the source code of the software controller to execute in QEMU. QEMU and Matlab/Simulink communicate through a socket. QEMU run as a client to communicate with a server that employs Matlab as a computation engine. The Matlab engine provides a library of functions that allows the server to start and end a Matlab process and send commands to be processed in Matlab. The physical components are modeled in Simulink and set up by the Matlab engine. The communication block in Simulink is represented by an S-function. The S-function, which we generate based on the cyber/physical interface, sends physical state data to or receives cyber commands from the server via shared memory. Matlab/Simulink executes all blocks in the Simulink model at each sampling step. During each step, the S-function block is executed to communicate with QEMU.

6. Evaluation

In this section, we report two experiments applying our approach to real-world control systems: TableSat[22,23] and Automatic Transmission Controller.[4] Through the first experiment, we show how the virtualization can be used as a platform to develop controllers by comparing results from the real and virtual TableSat.

The second experiment is from an automotive application in Simulink. Through the second experiment, we show how our approach can achieve higher fidelity than Simulink/Stateflow. All experiments were performed on a machine with 2.93 GHz Intel(R) Xeon and 8G memory.

6.1. *TableSat*

TableSat (shown in Fig. 9) is an interactive platform from the University of Michigan, which emulates in one degree-of-freedom the dynamics, sensing, and actuation capabilities required for satellite attitude control. We have obtained the TableSat via collaboration and rebuilt it to suit our needs.

TableSat is driven by two computer fans and spin down by the friction. It contains a high-precision rate gyro, Silicon Ring Gyro CRS03, to measure angular velocity. An onboard Diamond Systems Athena II SBC computer running the Debian operating system communicates to a ground station via a wireless 802.11b interface. The computer interfaces to sensors through 16-bit analog-to-digital converters and to actuators through amplified 12-bit digital-to-analog channels. The cyber component gathers the angular velocity from the gyroscope sensor. Then based on the gyro data, the control algorithm calculates suitable voltage which is applied to the fan to change the motion of TableSat. The controller relies only on the gyroscope sensor to stabilize the TableSat motion.

6.1.1. *TableSat virtualization*

We have developed a virtual execution environment for TableSat, which is illustrated in Fig. 10. We utilize the X86 processor model to emulate the Athena II SBC

Fig. 9. TableSat.

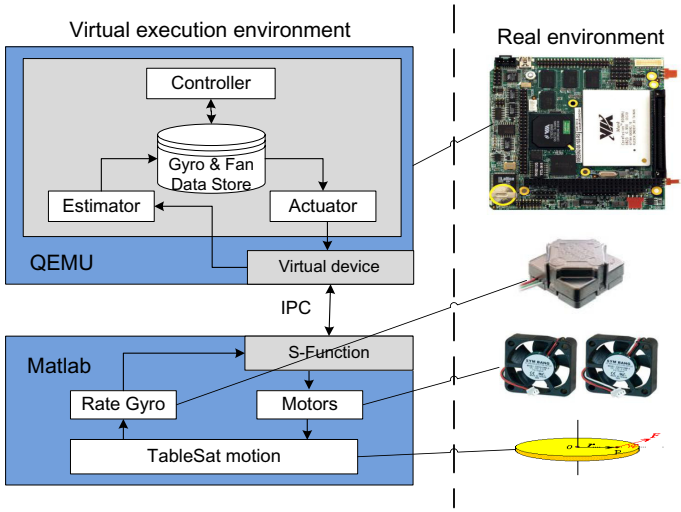

Fig. 10. TableSat virtualization.

in QEMU. The software controller is implemented in the C language. The physical components include TableSat dynamics, friction, fans and sensor. They are modeled mathematically according to respective physical characteristics in Matlab/Simulink. The equations of TableSat motion are:

$$I\dot{\omega} = lK_{\omega f}(v_1 + v_2) - f_{\mathrm{TS}}(\omega), \tag{3}$$

$$\dot{v_1} = -\alpha v_1 + K_{v\dot{\omega}}(V_1 - f_{\mathrm{fan1}}(v_1)), \tag{4}$$

$$\dot{v_2} = -\alpha v_2 + K_{v\dot{\omega}}(V_2 - f_{\mathrm{fan2}}(v_2)), \tag{5}$$

where I is the TableSat moment of inertia, ω is the TableSat angular velocity, v_1 is the speed of the fan1, v_2 is the speed of the fan2, l is the fan moment arm, f_{TS} is the TableSat friction and is a function of ω, $K_{\omega f}$ is the fan speed to force constant, V_1 is the voltage applied to the fan1, V_2 is the voltage applied to the fan2, α is the fan time constant, $K_{v\dot{\omega}}$ is the fan voltage to change in speed constant, and f_{fan1} and f_{fan2} are the frictions in the fans and are functions of v_1 and v_2, respectively.

We simulate the physical dynamics in Matlab/Simulink as shown in Fig. 11. There are three blocks: Communication, TableSat and Gyro Noise. Matlab communicates with the external environment through an S-function named Communication: sends the angular velocity to the external environment and receives voltage commands. The S-function named TableSat computes the Tablesat rotary (TableSat motion, Rate Gyro and Motors), as described in Eqs. (3)–(5). The random block named Gyro Noise models the sensor noise characterization. The noise characterization of gyro signal is that: Mean = 0.0025, Variance = 0.1936. The two blocks work together to generate the final angular velocity data.

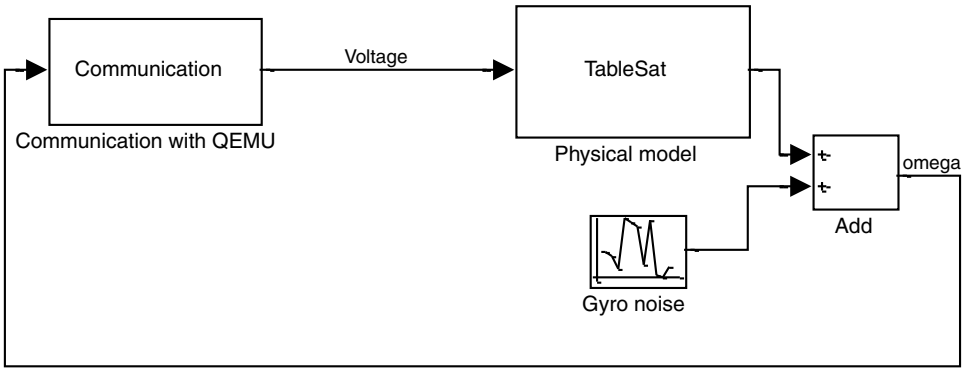

Fig. 11. Physical model in Simulink.

6.1.2. *Experiment result*

In this experiment, the angular velocity of the TableSat is controlled by manipulating voltage applied to the fan. The overall design requirements are:

- Rise time: the time required for the response to rise from 10% to 90% of its final value under 10 s;
- Peak overshoot ratio: the ratio of the first highest peak value reached by the response to the desired value below 0.4;
- Steady-state error: the difference between the desired final output and the actual one when the system reaches a steady state below 5°/s;
- Settling time: the time required for the response curve to reach and stay within a steady-state error band under 15 s.

We have developed a Proportional Integral Derivative (PID) controller and tested it in the holistic virtual execution environment. The final discretization of the PID algorithm is an incremental form:

$$\begin{aligned}\Delta u(t) &= u(t) - u(t-1)\\ &= K_p(1 + T_s/T_i + T_d/T_s)e(t) + K_p(-1 - 2T_d/T_s)e(t-1) + K_pT_d/T_se(t-2)\\ &= Ae(t) + Be(t-1) + Ce(t-2), \qquad (6)\end{aligned}$$

where u is the control inputs, e is the error between the measured value and desired value, K_p is the proportional gain, T_i is the integral time, T_d is the derivative time, T_s is the sampling time, t is the instantaneous time, $A = K_p(1 + T_s/T_i + T_d/T_s)$, $B = K_p(-1 - 2T_d/T_s)$, $C = K_pT_d/T_s$.

To evaluate the PID controller, we conducted tests for a series of step input of expected angular velocity. In the virtual TableSat, we set the fixed sampling interval $T = 0.2s$ and the fixed interval from A/D conversion to D/A conversion $d = 0.1s$. As shown in Fig. 12, we run the controller eight times. Input of expected

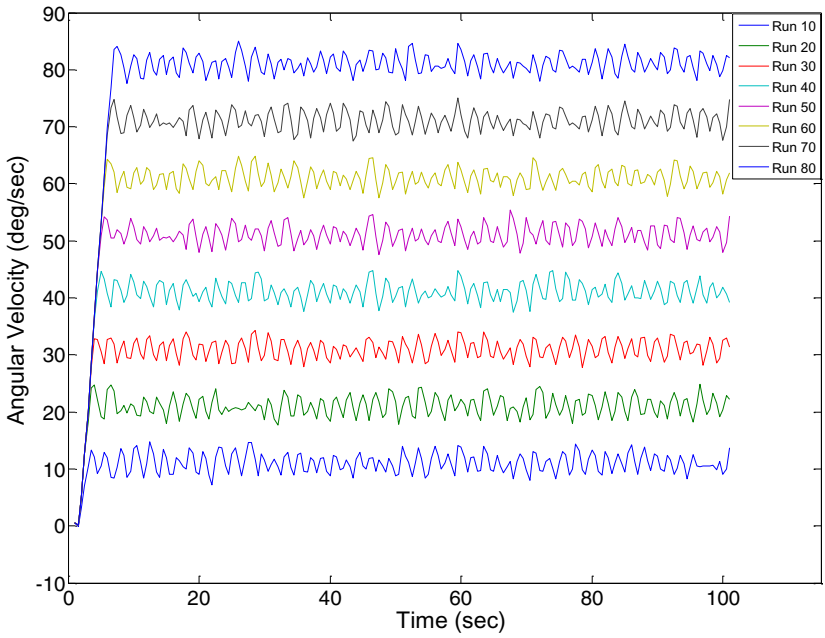

Fig. 12. TableSat test results gathered in virtual environment.

angular velocity from 10°/s to 80°/s (Run 10: 10°/s, Run 20: 20°/s, Run 30: 30°/s, Run 40: 40°/s, Run 50: 50°/s, Run 60: 60°/s, Run 70: 70°/s, Run 80: 80°/s). The curves of angular velocity, given the expected angular velocities, show that with increasing voltage there is an initial step rise to a maximum value followed by a fall to a first minimum value and then stabilization on the expected value. All the angular velocity response of TableSat shows that its control properties meet the design requirement.

The designed controller is then tested on the real TableSat system. In the real TableSat, we use a timer (0.2 s) to wake up the software periodically. Figure 13 shows results gathered from the real TableSat. Comparing Fig. 12 with Fig. 13, we can see the characters of angular velocity curves are closely matched.

Before quantifying the divergence between the real TableSat and its virtualization, we first define an evaluation metric, absolute divergence, which is the difference between the actual velocity and expected ideal velocity. We can calculate this divergence by measuring the difference between the actual velocity and the ideal velocity. Table 1 shows statistics of absolute divergence over eight runs. Each row in the table shows statistics of a system run. We recorded the angular velocity at every 0.5 s. The average absolute divergence over all time instant is relatively low and below 2.4°/s. All the maximum absolute divergence values occur in the first two 2 s. The main reason is precision of TableSat static friction measurement. The static friction in TableSat is represented by the voltage magnitude, applied to the

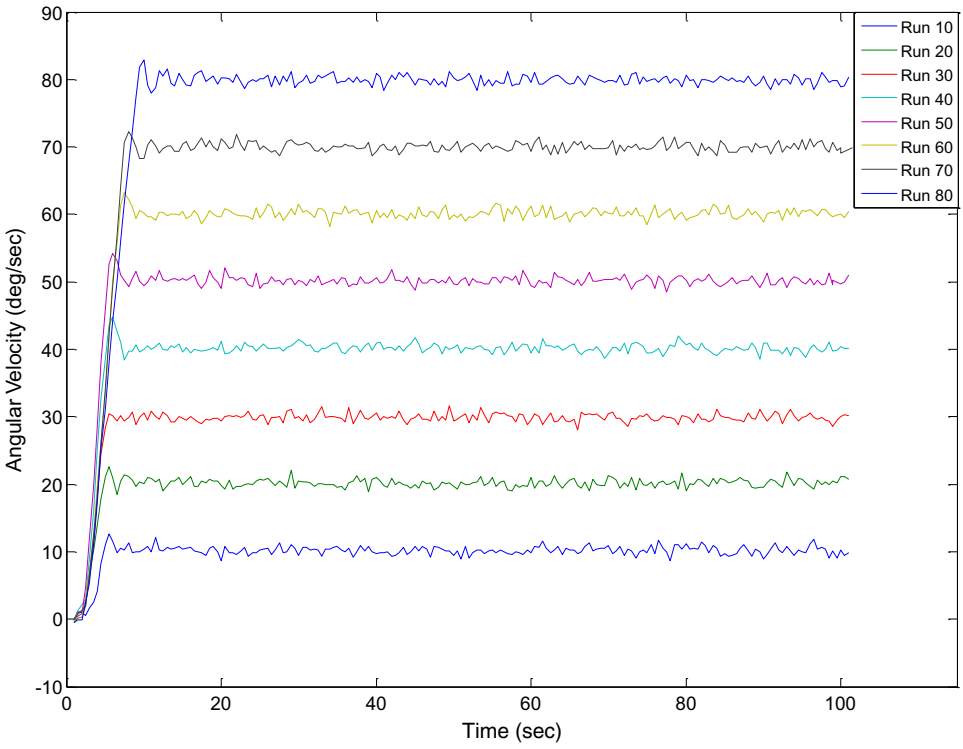

Fig. 13. TableSat test results gathered in real environment.

Table 1. Summary of absolute divergence.

	Max (°/s)	Min (°/s)	Mean (°/s)	Std (°/s)
Run 10	3.637	−10.84	−1	2.069
Run 20	3.788	−13.88	−1.115	2.424
Run 30	2.702	−16.38	−1.429	2.514
Run 40	4.219	−12.22	−1.177	2.376
Run 50	2.738	−8.529	−1.105	2.132
Run 60	4.792	−19.44	−1.528	3.478
Run 70	3.847	−19.39	−1.818	3.864
Run 80	2.352	−27.44	−2.356	5.004

fans, below which TableSat will not move. The power supply may change frequently due to physical or mechanical causes such as battery level. The precision of static friction will influence initial motion of TableSat.

We evaluate the time usages of the simulation (shown in Fig. 14). We set the fixed interval $T = 2\,\text{s}$ and $d = 1\,\text{s}$. Comparing the virtualized time with the simulation time used, the results show a nonlinear relationship between the virtualized

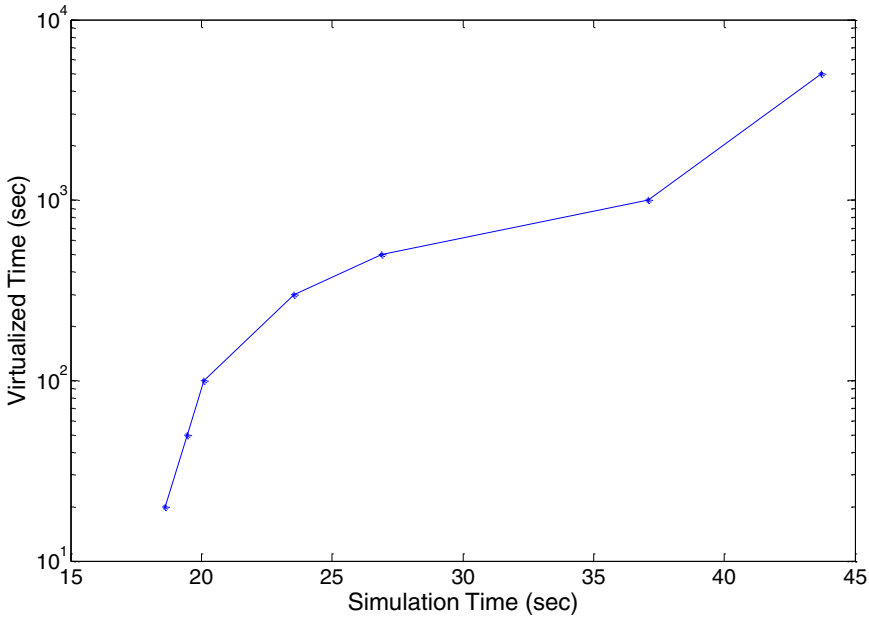

Fig. 14. Summary of time utilizations.

time and simulation time used. It spends most time on starting Matlab engine and initializing the Simulink model. After the initialization, the growth of simulation time tends to slow as the virtualized time increases. Therefore, our approach is reasonably fast and efficient.

The experiment shows that our virtual environment can simulate the real system with reasonable accuracy. This can enable development of software on the virtualization before the real physical environment becomes available.

6.2. *Automatic transmission controller*

The second experiment is on an automatic transmission model given as a sample in the Simulink package. The automatic transmission adjusts gear ratios as the vehicle moves. The gear shift controller performs the function of gear selection in the automatic transmission. The model describes the control of a four-speed automatic transmission. Figure 15 illustrates the physical model of the automatic transmission.

In the automatic transmission model, Engine block, Transmission block and Vehicle block model the engine, four-speed automatic transmission, and vehicle, respectively. The communication block in this model enables physical components to communicate with QEMU. Engine block interpolates engine torque (Impreller-Torque), versus throttle (Throttle) and engine speed (EngineRPM). The Torque-Converter and the Transmission Ratio subsystems make up the Transmission block.

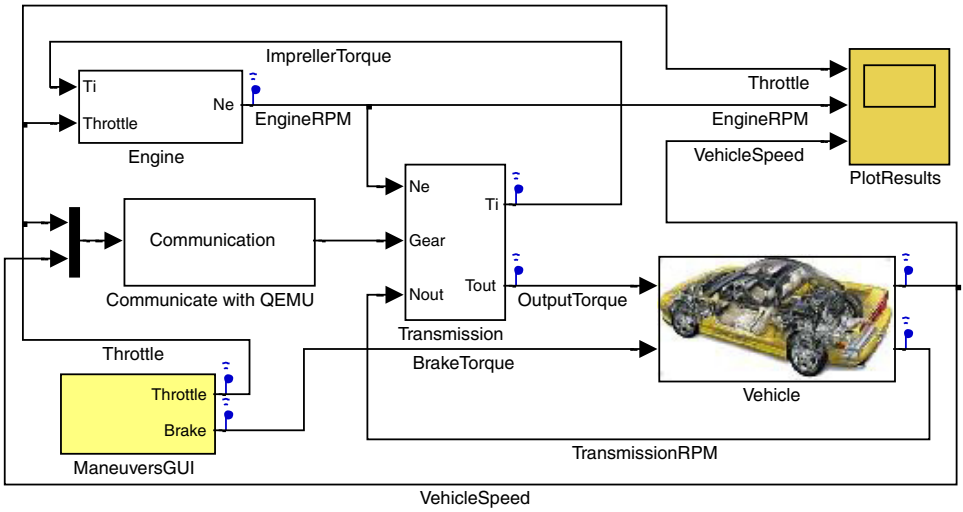

Fig. 15. Automatic transmission physical model in Simulink.

The transmission model first calculates the turbine speed based on the engine speed, and then outputs torque (OutputTorque) to the vehicle via the static gear ratio. The Vehicle block uses the torque (OutputTorque) to compute the acceleration and integrates it to compute the vehicle speed (VehicleSpeed). User inputs to the model are in the form of throttle and brake torque. There are four maneuvers: passing maneuver, gradual acceleration, hard braking and coasting. Each maneuver has different throttle and brake torque schedule.

We carry out a series of controller tests on both virtualization and Simulink/Stateflow simulation. In the original model, the Stateflow block labeled Shift Logic implements gear selection for the transmission. The model takes throttle and vehicle speed as the inputs and the desired gear number as the output. The controller selects a gear based on the input throttle values and computes based on table lookups. The selected gear is used to look up a Boolean value for each of the output clutch pressure. The overall Stateflow block is executed as a discrete-time system, sampled every 40 ms. In our virtualization, controller source code was generated from the Stateflow block automatically by Simulink Coder.[4] Then we port the control software into QEMU. We set the fixed interval $T = 0.04\,\text{s}$ and $d = 0.02\,\text{s}$.

We summarize the statistics of error between the output from the virtualization and that from the Simulink/Stateflow simulation (see Table 2). We use absolute divergence, the deviation of the Simulink/Stateflow simulation value from the virtualization value, to define the error. Each column in the sub-table Maneuver shows error statistics of a system simulation run for a maneuver. For instance, the last column shows the error statistics of Throttle, EngineRPM, VehicleSpeed,

Table 2. Summary of error statistics.

Type	Maneuver			
	Passing maneuver	Gradual acceleration	Hard braking	Coasting
Throttle (degree)				
max	0	0	0	0
min	0	0	0	0
mean	0	0	0	0
std	0	0	0	0
EngineRPM (RPM)				
max	480.9	0.7287	1.561	351.4
min	−989.1	−106.7	−251.2	−689.1
mean	−3.333	−0.4703	−1.388	6.425
std	49.83	5.746	12.44	48.45
VehicleSpeed (mph)				
max	0.3192	0.05764	0.2032	0.5237
min	−0.1662	0	−0.04715	−0.3215
mean	−0.01688	0.006017	−0.03145	0.07979
std	0.03248	0.007769	0.01972	0.04198
Gear				
max	1	1	1	1
min	−1	−1	−1	−1
mean	0	0	−0.002663	0.003995
std	0.07303	0.05164	0.07298	0.08155
ImprellerTorque (ft/lb)				
max	445.8	24.21	185.8	578.6
min	−503.3	−25.2	−81.59	−323.7
mean	−0.5863	−0.09836	−0.1297	0.4167
std	30.72	1.507	7.949	31.13
OutputTorque (ft/lb)				
max	519.5	111.4	356.4	1135
min	−544.3	−41.98	−219	−825.5
mean	−0.7031	0.08794	−0.1985	0.3177
std	35.21	4.853	16.84	56.24

Note: Gears: 1, 2, 3, or 4.

Gear, ImprellerTorque, OutputTorque in the coasting maneuver. We record these errors between the output produced with our tool and that produced using the Simulink/Stateflow not only for the response engine speed and vehicle speed but also for the other values (Throttle, Gear, ImprellerTorque and OutputTorque). We carry out tests in four different maneuvers and find errors in all recorded variables except for Throttle. After analysis, we find these errors are all caused by gear selection. Here we choose the test generated in coasting maneuver to discuss these errors in detail. We recorded these variables at every 0.04 s.

The curves shown in Fig. 16 are composed of the gear selections gathered from the virtualization and the Simulink/Stateflow simulation, for controlling the throttle and brake torque in the coasting maneuver. There are three times of gear upshift in this maneuver. We indeed found the errors that occur only for few points during gear

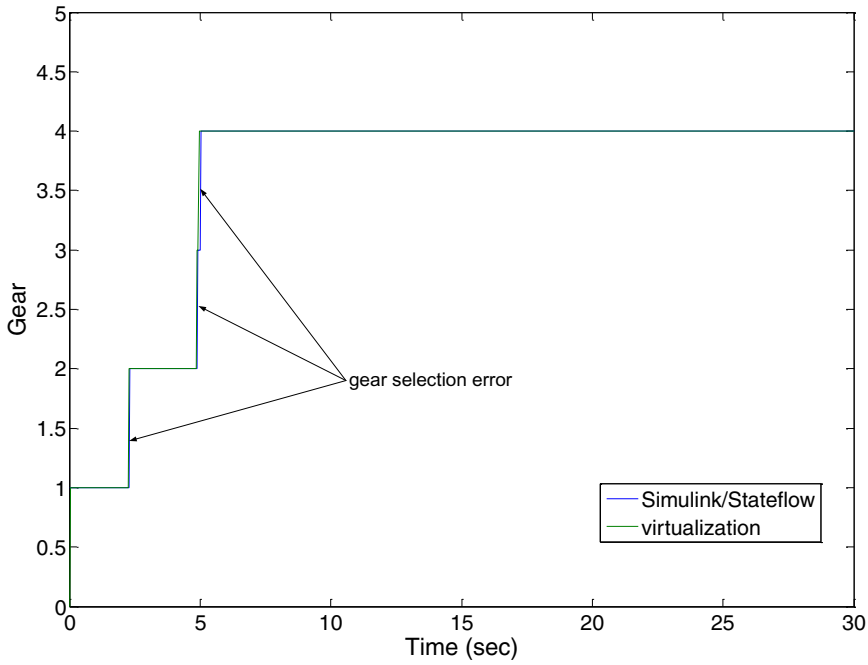

Fig. 16. Gear selection error between virtualization and Simulink/Stateflow simulation.

Table 3. Time of gear selection in the virtualization and the Simulink/Stateflow.

	2.28 s	2.32 s	2.36 s	4.88 s	4.92 s	4.96 s	5.00 s	5.04 s	5.08 s
Gear selection 1	1	2	2	2	3	3	4	4	4
Gear selection 2	1	1	2	2	2	3	3	3	4

Note: Gear selection 1 and gear selection 2 are gathered in the virtualization and Simulink/Stateflow, respectively. Gears: 1, 2, 3, or 4.

upshift. Table 3 shows different times of gear upshift. We can find the controller in virtualization performs gear selection with an earlier trigger time, when the vehicle transmission needs gear upshift. For example, the gear upshift from gear 1 to gear 2 occurs at 2.32 s in virtualization, whereas in Simulink/Stateflow the time is 2.36 s. In the virtualization environment, the physical components will be controlled by the current command until the D/A conversion finished ($[t_{ci}, t_{ci} + d)$) and then evolves with new command in the rest of reactive interval ($[t_{ci} + d, t_{ci+1})$). But in the Simulink/Stateflow environment, it will be controlled by the current command until next reactive interval ($[t_{ci}, t_{ci+1})$). Therefore, the error occurs during the time ($[t_{ci} + d, t_{ci+1})$). In addition, the initial value of gear in Simulink/Stateflow is 1, where the initial value is 0 in the source code. These will ultimately cause deviation of vehicle speed (see Fig. 17). The curve shown in Fig. 17 represents the absolute divergence of vehicle speed. The error of vehicle speed has always existed even it

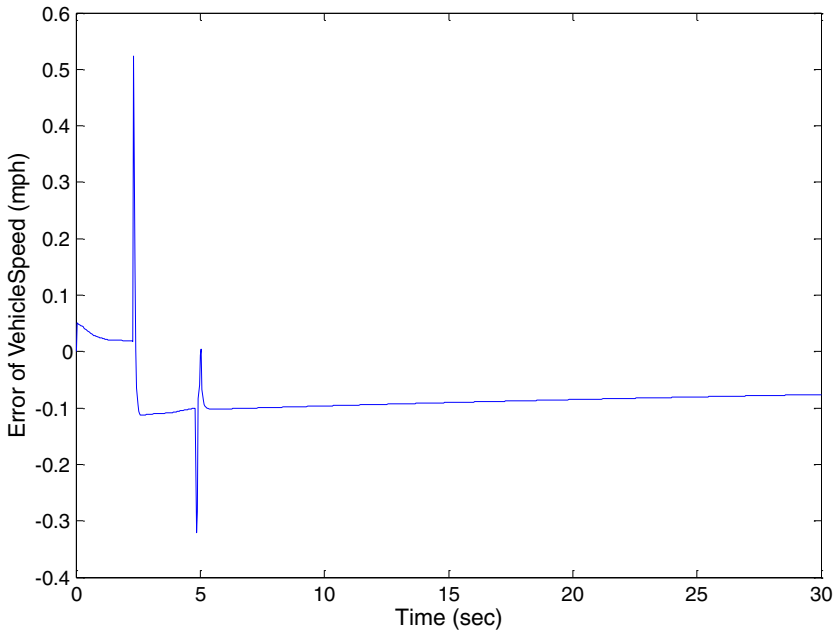

Fig. 17. Vehicle speed error between virtualization and Simulink/Stateflow Simulation.

is quite small. And the error increases sharply during gear upshift. Therefore, we conclude that our virtual environment can achieve higher fidelity than Simulink/Stateflow.

7. Conclusions

This paper proposes a high-fidelity approach to CPS virtualization. Our approach extends virtual machines and incorporates the ability to emulate physical components. We have developed a synchronization strategy to integrate two different domains: cyber and physical. We have evaluated our approach on the real-world control systems. The experiments and evaluations demonstrate that our approach is effective: the comprehensive virtualization environment can capture the dynamic behaviors of the system in high-fidelity, and efficient: the costs of simulation are low. Our approach is promising in enabling early development of cyber components and early exploration of the synergy of cyber and physical components of CPS.

Formal co-verification has the potential for exhaustive state space coverage of a whole system. Thus, it is desired to integrate formal verification and virtualization, so that high-confidence achieved by formal verification and practicality supported by virtualization can be properly leveraged for system validation. The virtualization technique presented in this paper is a preliminary step towards this direction. We will extend the CPS virtualization environment and further investigate the integration of virtualization and formal verification under this common framework.

Acknowledgments

This research received financial support from the National Science Foundation of the United States (Grant #: 0720546 and Grant #: 0916968), the National High-Tech Research and Development Plan of China (Grant #: 2011AA010105 and Grant #: 2011AA010102), and the National Infrastructure Software Plan of China (Grant #: 2012ZX01041-002-003).

References

1. Lee E. A., Cps foundations, *Proc. 47th Design Automation Conf.* (*DAC*), Anaheim, CA, ACM, June, pp. 737–742, 2010.
2. Nelson S., Waskiewicz P. J., Virtualization, Writing (and testing) device drivers without hardware, *Linux Plumbers Conf.*, Santa Rasa, CA, August, 2011.
3. Bellard F., Qemu, a fast and portable dynamic translator, *Proc. Annu. Conf. USENIX Annu. Tech. Conf.*, ATEC '05, Anaheim, CA, USENIX Association, Berkeley, CA, USA, pp. 41–46, 2005.
4. MathWorks, Matlab/simulink, http://www.mathworks.com.
5. Becker D., Singh R., Tell S., An engineering environment for hardware/software co-simulation, *Design Automation Conf. 1992. Proc.*, *29th ACM/IEEE*, June, pp. 129–134, 1992.
6. Eker J., Janneck J., Lee E. A., Liu J., Liu X., Ludvig J., Sachs S., Xiong Y., Neuendorffer S., Taming heterogeneity — the ptolemy approach, *Proc. IEEE* **91**(1):127–144, 2003.
7. Hoffmann A., Kogel T., Meyr H., A framework for fast hardware-software co-simulation, *Design*, *Automation and Test in Europe*, *2001*, *Conference and Exhibition 2001*, *Proc.*, Munich, ACM, pp. 760–764, 2001.
8. Semeria L., Ghosh A., Methodology for hardware/software co-verification in c/c++, *Design Automation Conf.*, *2000. Proc. ASP-DAC 2000. Asia and South Pacific*, Yokohama, Japan, ACM, June, pp. 405–408, 2000.
9. Passerone C., Lavagno L., Chiodo M., Sangiovanni-Vincentelli A., Fast hardware/software co-simulation for virtual prototyping and trade-off analysis, *Proc. 34th Annual Design Automation Conf.*, DAC '97, Anaheim, CA, ACM, New York, NY, pp. 389–394, 1997.
10. Mentor graphics, Seamless, http://www.mentor.com.
11. National instruments, Labview, http://www.ni.com/labview.
12. Wolfram research, Mathematica, http://www.wolfram.com.
13. Modelica association, Modelica, https://modelica.org.
14. Mentor graphics, Modelsim, http://www.model.com.
15. Mueller W., Becker M., Elfeky A., DiPasquale A., Virtual prototyping of cyber-physical systems, *Design Automation Conf.* (*ASP-DAC*), *2012 17th Asia and South Pacific*, Sydney, ACM, February, pp. 219–226, 2012.
16. Al-Hammouri A. T., A comprehensive co-simulation platform for cyber-physical systems, *Comput. Commun.* **36**(1):8–19, 2012.
17. Yeh T.-C., Chiang M.-C., On the interfacing between qemu and systemc for virtual platform construction: Using dma as a case, *J. Syst. Archit.* **58**(3–4):99–111, 2012.
18. Kuznetsov G. C. V., Vitaly Chipounov, Testing closed-source binary device drivers with ddt, *USENIX Annual Technical Conf.* (*USENIX*), Boston, MA, pp. 159–172, June, 2010.

19. Benveniste A., Caspi P., Edwards S., Halbwachs N., Le Guernic P., de Simone R., The synchronous languages 12 years later, *Proc. IEEE* **91**(1):64–83, 2003.
20. Halbwachs N., Caspi P., Raymond P., Pilaud D., The synchronous data flow programming language lustre, *Proc. IEEE* **79**(9):1305–1320, 1991.
21. Bauer K., A New Modelling Language for Cyber-Physical Systems, Ph.D. Thesis, Technical University of Kaiserslautern, Germany, 2012.
22. Vess M. F., System modeling and controller design for a single degree of freedom spacecraft simulator, Master's Thesis, University of Maryland, USA, 2005.
23. Atkins E., Green J., Yi J., Woo H., Browne J., Mok A., Xie F., The tablesat platform and its verifiable control software, American Institute of Aeronautics and Astronautics, Seattle, USA, 2009.