An Automata-Theoretic Approach to Hardware/Software Co-verification

Juncao Li[1], Fei Xie[1], Thomas Ball[2], Vladimir Levin[2], and Con McGarvey[2]

[1] Department of Computer Science, Portland State University
Portland, OR 97207, USA
{juncao,xie}@cs.pdx.edu
[2] Microsoft Corporation
Redmond, WA 98052, USA
{tball,vladlev,conmc}@microsoft.com

Abstract. In this paper, we present an automata-theoretic approach to Hardware/Software (HW/SW) co-verification. We designed a co-specification framework describing HW/SW systems; synthesized a hybrid Büchi Automaton Pushdown System model for co-verification, namely Büchi Pushdown System (BPDS), from the co-specification; and built a software tool for deciding reachability of BPDS models. Using our approach, we succeeded in co-verifying the Windows driver and the hardware model of the PIO-24 digital I/O card, finding a previously undiscovered software bug. In addition, our experiments have shown that our co-verification approach performs well in terms of time and memory usages.

1 Introduction

Computer systems are pervasive, ranging from embedded control to banking to education. Users demand high-confidence in these systems, and high-confidence is traditionally achieved by extensive testing which is becoming increasingly cost-prohibitive. As a result, formal verification such as model checking [1] is playing a greater role in verifying the correctness of these systems. In practice, engineers typically attempt to verify hardware and software independently. In order to verify complete systems, the correctness of the Hardware/Software (HW/SW) interfaces must be established.

HW/SW co-verification, verifying hardware and software together, is essential to establishing the correctness of HW/SW interfaces. One major challenge in co-verification is the integration of hardware and software representations within the same formal model. Hardware and software verification utilize different models. For verification of software implementations, one of the most popular models has been pushdown systems whose semantics closely resemble the semantics of software programs which are often infinite systems. Hardware designs are finite-state and often modeled as some kind of finite-state state machines. However, for co-verification it is not desired to model both hardware and software as pushdown systems or finite state machines (see related work).

In this paper, we present an automata-theoretic approach to HW/SW co-verification. The foundation of this approach is a hybrid Büchi Automaton Pushdown System as a unifying model for HW/SW co-verification, namely the Büchi Pushdown System (BPDS). It synchronizes a single Pushdown System (PDS) that has an unbounded stack

D.S. Rosenblum and G. Taentzer (Eds.): FASE 2010, LNCS 6013, pp. 248–262, 2010.

and a Büchi Automaton (BA). The co-verification flow as supported by this approach is shown in Figure 1. The main components of this flow include:

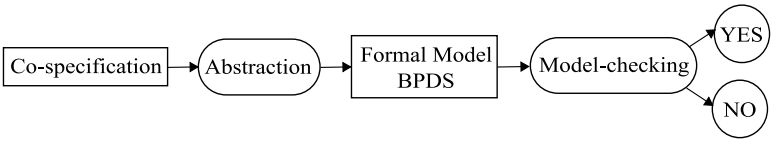

Fig. 1. Co-verification flow

- *Co-specification.* Co-specification is essential in order to present system designs at proper levels of details. We developed a co-specification framework that describes the hardware model, the software model, and the HW/SW interface.
- *Co-verification model.* We designed a formal co-verification model, BPDS, to capture hardware and software designs, as well as their concurrent executions and interactions. The core contribution is our process for constructing a BPDS by synchronizing a BA that abstracts hardware and a PDS that abstracts software.
- *Model-checking BPDS.* We developed a method for checking reachability properties of a BPDS and analyzed its complexities. To evaluate the effectiveness of our approach, we implemented an automatic verification tool for BPDS.

Another component of this flow is abstraction, which we will elaborate in another paper.

Related work. Kurshan, et al. presented a co-verification framework that models hardware and software designs using finite state machines [2]. Xie, et al. extended this framework to hardware and software implementations and improves its scalability via component-based co-verification [3]. However, finite state machines are limited in modeling software implementations, since they are not suitable to represent software features such as a stack.

Another approach to integrating hardware and software within the same model is exemplified by Monniaux in [4]. He modeled a USB Open Host Controller Interface (OHCI) device using a C program and instrumented the device driver, another C program, in such a way as to verify that the USB OHCI controller driver correctly interacts with the device. The hardware and software were both modeled by C programs, thus formally Pushdown Systems (PDS). However, straightforward composition of the two PDSs to model the HW/SW concurrency is problematic, because it is known, in general, that verification of reachability properties on concurrent PDS with unbounded stacks is undecidable [5]. Based on an approximation of the OHCI HW/SW interface, Monniaux merges the C program models of both hardware and software into one sequential program, formally a single PDS. Monniaux's approach has three key drawbacks: (1) programming languages such as C do not have semantics for concurrency, so the concurrent nature of hardware is not fully modeled; (2) the HW/SW concurrency is not accurately modeled; (3) the complexity of checking the resulting PDS is often unnecessarily high, due to the way hardware and concurrency is modeled in the PDS.

Schwoon used a combination of PDS and BA to verify Linear Temporal Logic (LTL) properties of PDS and his approach has been implemented in the Moped tool [6]. A LTL

formula is first negated and then represented as a BA. Moped combines the BA and PDS in such a way that the BA monitors the state transitions of the PDS, so the model-checking problem is to compute if the BA has an accepting run. Schwoon's goal was to verify software only; however, our goal is to co-verify safety properties of HW/SW systems. We will discuss more details about Schwoon's work in Section 3.1.

Outline. The rest of this paper is organized as follows. In Section 2, we present our co-specification framework, which is illustrated by a Windows PCI device driver example. In Section 3, we introduce our co-verification model, BPDS. In Section 4, we describe how to construct a BPDS by synchronizing a BA and a PDS. In Section 5, we discuss how to conduct reachability analysis on a BPDS. In Section 6, we present the evaluation and experimental results. In Section 7, we conclude and discuss future work.

2 Co-specification

Co-specification describes the HW/SW system to be verified. The essential parts include the hardware model, the software model, and the HW/SW interface. The level of detail varies due to (1) platform differences, e.g., embedded system or PC; (2) verification foci, e.g., verifying software by providing hardware models, verifying hardware using software models, or verifying both. As an example, we show how to specify the HW/SW interface for the verification of a device driver implementation and its device model. The goal of this specification is to verify if the driver implementation is correct in terms of the HW/SW interface properties. In order to facilitate the understanding of our modeling approach, we first introduce a simple Windows PCI driver example.

2.1 A Windows PCI Driver Example

Device drivers check device status or send commands to devices by reading or writing device registers, and receive notification of state changes from devices through interrupts. In Windows [7], device drivers are organized through driver stacks. Each layer of a driver stack services a specific type of device in the corresponding hardware stack. Usually, different driver layers have different I/O interfaces. In this paper, we utilize PCI (Peripheral Component Interconnect) device drivers as an example. PCI drivers read/write device registers using functions such as `READ_REGISTER_UCHAR`, `WRITE_REGISTER_UCHAR`, `READ_PORT_UCHAR`, `WRITE_PORT_UCHAR`, etc. Depending on whether a driver uses memory or port mapped I/O to represent its device interface registers in virtual memory, the functions are different.

Figure 2 shows excerpts from an Open System Resources (OSR) sample driver for a PCI device, Sealevel PIO-24 digital I/O card. The card has three 8-bit ports (namely, A, B, and C) for input or output. When the interrupt is enabled and Port A has an input, the card fires a data-ready interrupt. The driver reads data when the data-ready interrupt fires and outputs data by writing to the port registers. `DioEvtDeviceControl` is the callback function that handles device control commands and `DioIsr` is the Interrupt Service Routine (ISR).

```
VOID DioEvtDeviceControl( ... ) {
  ...
  switch(IoControlCode) {
    ...
    // Waits for an interrupt to occur, and when it does,
    // ISR/DPC will read the contents of PortA.
    case IOCTL_WDFDIO_READ_PORTA_AFTER_INT:
    ...
    // If PortAInput is true, the interrupt is enabled
    if (devContext->PortAInput == FALSE) {
        status = STATUS_INVALID_DEVICE_STATE;
    } else {
        // Store the I/O request to CurrentRequest
        devContext->CurrentRequest = Request;

        // Tell ISR: we're waiting for an interrupt
P1:     devContext->AwaitingInt = TRUE;
        ...
        return;
    }
    break;
    ...
  }
  ...
}
```

```
BOOLEAN DioIsr( ... ) {
  ...
  // Check if we have an interrupt pending
  data = READ_REGISTER_UCHAR(
      devContext->BaseAddress +
              DIO_INTSTATUS_OFFSET );
  if(data & DIO_INTSTATUS_PENDING) {

    // Are we waiting for this interrupt
P2: if(devContext->AwaitingInt) {
      // Read the contents of PortA
      data = READ_REGISTER_UCHAR(
          devContext->BaseAddress +
              DIO_PORTA_OFFSET );
      // Store it in our device context
      // DPC will send the data to users
      devContext->PortAValueAtInt = data;
      devContext->AwaitingInt = FALSE;
    }

    // Request our DPC
P3: WdfInterruptQueueDpcForIsr( Interrupt );
    // Tell WDF, and hence Windows, this is our interrupt
    return(TRUE);
  }
  return(FALSE);
}
```

Fig. 2. Excerpts from OSR sample driver code for PIO-24 digital I/O card

2.2 Language Features for Co-specification

In our example, the C program of the driver is the software model. Next, we focus on the hardware model and the HW/SW interface.

We specify the hardware model and hardware-related parts of the HW/SW interface using the Verilog hardware description language [8]. There are two major reasons behind using Verilog for models related to hardware. First, Verilog is a popular language for hardware design. Lots of existing hardware has been designed using Verilog. Second, Verilog supports the concurrent semantics of hardware. A key feature is parallel assignments (a.k.a. nonblocking assignments that use the operator "<=" in the examples), which capture the simultaneous updates of register states through state transitions.

The **hardware model** describes behaviors of the device as state transitions. Different from the commonly used clock-driven semantics of Verilog, a state transition of our hardware model represents an arbitrarily long but finite sequence of clock cycles. This preserves hardware design logic that is externally visible to software, but hides details only necessary for synthesizable Register Transfer Level (RTL) design. Figure 3 shows the hardware model of the PIO-24 digital I/O card. The model simply fires an interrupt when it is in an interrupt-enabled state and Port A has an input. We define `rand` as a function that returns a non-deterministic value in the given range. There are three tasks: `reset`, `environment`, and `random`. The `environment` task is executed non-deterministically to simulate inputs from the environment to the device, e.g., the physical *reset* event that clears all registers. Depending on the properties to be verified, hardware models may be extended to exhibit more behaviors. Our model in Figure 3 has been simplified to one aspect of the device to illustrate the device/driver interactions.

```
begin hardware model
  // declare registers
  reg [7:0] PortA, IntConfg;
  ...
  // declare the tasks
  task reset; begin // clears all registers
    PortA <= 8'h0;
    ...
  end endtask
  // model the inputs from the environment
  task environment; begin
    // non-deterministically reset the hardware
    if(rand(0,1)) reset;
    // if the interrupt is enabled but not fired,
    // non-deterministically input to PortA.
    else if((IntConfg & 8'h4) && (IntStatus==0))
        PortA <= rand(8'h0, 8'hFF);
    ...
  end endtask
  ...
  // assign non-deterministic value to registers
  task random; begin
    PortA <= rand(8'h0, 8'hFF);
    ...
    Ctrl <= rand(8'h0, 8'hFF) & 8'h9B;
    IntConfg <= rand(8'h0, 8'h07);
    IntStatus <= rand(8'h0, 8'h01);
  end endtask

  // initial state of the device: non-deterministically initialized
  initial random;
  // non-deterministically execute the environment
  if( rand(0,1) ) begin
    // low level triggers the interrupt
    if( (IntConfg == 8'h04) && ((PortA & 8'h01)==0) )
    begin
      IntStatus <= 1; // set the interrupt status
      INTR <= 1; // set the interrupt pending status to SW
    end
    // high level triggers the interrupt
    if( (IntConfg == 8'h05) && ((PortA & 8'h01)==1) )
    begin IntStatus <= 1;  INTR <= 1; end
    ...
  end
  else environment;
end hardware model
```

Fig. 3. Hardware model for PIO-24 digital I/O card device

The **interface specification** describes the HW/SW interface. Hardware and software run asynchronously and only communicate through their interface. The HW/SW interface includes two parts: shared interface states and interface events. Interface states are state variables provided either by hardware or software and accessible to both; interface events have two types: hardware or software. A hardware interface event happens when hardware updates the software interface states, and vice versa. A typical example of hardware interface events is an interrupt which causes context switches in software. However, it is possible that in a HW/SW system, software provides shared memory for hardware to access. In this case, a hardware interface event (e.g., write to the shared memory) will not cause any context switch in software. In summary, interface events identify the situations when both software and hardware must transit synchronously.

Modern system designs usually have software and hardware aligned as layers in a stack, different layers of software work with their corresponding layers of hardware together to deliver certain functionalities. For example, PCI bus and USB bus are different HW/SW layers in a PC system. Interface specification needs to describe the HW/SW interface behaviors in order to hide the implementation details of other hardware and software layers that lie in between the hardware and software layers to be verified.

For the PIO-24 digital I/O device/driver, the device provides interface registers for the driver to operate the device. In order for the driver to access the interface registers, the Windows OS maps the device interface registers into virtual memory through a technique called Memory Mapped I/O (MMIO). When the driver writes to/reads from a mapped memory address by calling register operation functions, the corresponding interface register will be updated by the OS. How the register should be updated depends on the HW/SW interface protocol. We need to specify (1) the virtual memory alignment for the mapped interface registers, so a specific memory address is related to the proper interface register; (2) how interface registers should be updated when the driver accesses the registers, i.e., when software interface events happens. On the other side, the

device communicates with the driver through interrupts, i.e., hardware interface events. When hardware fires an interrupt, the Windows OS sets its internal interrupt pending status to be true and schedules the driver-provided ISR to service the interrupt.

Figure 4 shows the HW/SW interface specification for the PIO-24 digital I/O device/driver: (1) resource mappings for the driver. The PIO-24 device is mapped as MMIO. The resource mapping type indicates the set of interface register functions used by the driver; (2) interface declaration, which declares the device interface registers with their sizes and mapped address offsets in virtual memory, software interface events when the driver writes/reads a specific interface register, and hardware interface events when hardware fires interrupts; (3) implementation of software interface events. Each interface register is associated with two software interface event functions: read and write. The functions describe device interface state transitions when read/write events happen on the registers. Hardware interface events are defined by connecting the interrupts to the corresponding ISRs that are implemented in the driver model.

```
begin interface
  // resource mappings: Memory Mapped I/O
  use MMIO;

  // interface declaration
  // syntax: <offset(byte), length(byte)> −−>
  //                  name, read_event, write_event;
  <0x00, 1> −−> PortA, read_PortA(), write_PortA(VAR);
  ...
  <0x04, 1> −−> IntConfg, read_IntConfg(),
                write_IntConfg(VAR);
  <0x05, 1> −−> IntStatus, read_IntStatus(),
                write_IntStatus(VAR);
  interrupt INTR: // interrupt pending status
    void FireISR(); // the ISR connected to this interrupt

  // implementation of software interface events
  write_IntConfg(val) {
    if( ((val==4) && (PortA & 8'h1)!=0) ||
        ((val==5) && (PortA & 8'h1)==0) ||
        (val==6) || (val==7) || (val==0) )
      IntConfg <= val;
  }
  read_IntStatus() { // clear the interrupt status when read
    reg [7:0] retreg;
    retreg <= IntStatus;
    IntStatus <= 0;
    return retreg; // return the register value to software
  }
  ...
end interface
```

Fig. 4. Interface specification for PIO-24 digital I/O card device/driver

3 Co-verification Model: Büchi Pushdown System

We propose a hybrid Büchi Automaton Pushdown System, namely Büchi Pushdown System (BPDS) to represent both hardware and software in co-verification. Before we present BPDS, we first review the fundamentals of Büchi Automata (BA) and Pushdown Systems (PDS).

3.1 Background

Büchi Automata as Hardware Models. A **Büchi Automaton** $\mathcal{B}$, as defined in [9], is a non-deterministic finite state automaton accepting infinite input strings. Formally, $\mathcal{B}$ is a tuple $(\Sigma, Q, \delta, q_0, F)$, where Σ is the input alphabet, Q is the finite set of states, $\delta \subseteq (Q \times \Sigma \times Q)$ is the set of state transitions, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of final states. $\mathcal{B}$ accepts an infinite input string if and only if it has a run over the string to visit at least one of the final states infinitely often. A run of $\mathcal{B}$ on an infinite

string s is a sequence of states visited by $\mathcal{B}$ when taking s as the input. We use $q \xrightarrow{\sigma} q'$ to denote a transition from state q to q' with the input symbol σ.

Pushdown Systems as Software Models. A **Pushdown System**, as defined in [6], is a tuple $\mathcal{P} = (G, \Gamma, \Delta, \langle g_0, \omega_0 \rangle)$ where G is a finite set of global states (a.k.a. control locations), Γ is a finite stack alphabet, and $\Delta \subseteq (G \times \Gamma) \times (G \times \Gamma^*)$ is a finite set of transition rules. $\langle g_0, \omega_0 \rangle$ is the initial configuration. A PDS transition rule is written as $\langle g, \gamma \rangle \hookrightarrow \langle g', \omega \rangle$, where $((g, \gamma), (g', \omega)) \in \Delta$. A configuration of $\mathcal{P}$ is a pair $\langle g, \omega \rangle$, where $g \in G$ is a global state and $w \in \Gamma^*$ is a stack content. The set of all configurations is denoted by $Conf(\mathcal{P})$. If $\langle g, \gamma \rangle \hookrightarrow \langle g', \omega \rangle$, then for every $v \in \Gamma^*$ the configuration $\langle g, \gamma v \rangle$ is an immediate predecessor of $\langle g', \omega v \rangle$ and $\langle g', \omega v \rangle$ is an immediate successor of $\langle g, \gamma v \rangle$. The reachability relation $\Rightarrow$ is the reflexive and transitive closure of the immediate successor relation. Given a set $C \subseteq Conf(\mathcal{P})$, the forward reachability analysis, $post^*(C)$, computes the successors of elements of C. Schwoon [6] has designed algorithms to check both reachability and LTL properties on PDS. For computing $post^*$ on $\mathcal{P}$, the time and space complexities are both $(|G| + |\Delta|)^3$. The Moped tool implements all these algorithms.

3.2 Büchi Pushdown System

We synthesize a BPDS $\mathcal{BP}$ by building the synchronization of a BA $\mathcal{B}$ and a PDS $\mathcal{P}$. Let $\mathcal{B} = (\Sigma, Q, \delta, q_0, F)$ represent hardware, where Σ is the power set of the set of propositions that may hold on a configuration of $\mathcal{P}$ (i.e. a symbol of Σ is a set of propositions). In other words, the state transition of $\mathcal{B}$ is constrained by the current configuration of $\mathcal{P}$. We extend the definition of a pushdown system as $\mathcal{P} = (I, G, \Gamma, \Delta, \langle g_0, \omega_0 \rangle)$ representing software, where I is the power set of the set of propositions that may hold on a state of $\mathcal{B}$, G is a finite set of global states, Γ is a finite stack alphabet, and $\Delta \subseteq (G \times \Gamma) \times I \times (G \times \Gamma^*)$ is a finite set of transition rules. We write $\langle g, \gamma \rangle \xhookrightarrow{\tau} \langle g', w \rangle$ as a rule $((g, \gamma), \tau, (g', w)) \in \Delta$. $\langle g_0, w_0 \rangle$ is the initial configuration. It is important to note that we extend the pushdown system so that the transition rules in Δ are all labeled by $\tau \in I$, i.e., the state transition of $\mathcal{P}$ is constrained by the current state of $\mathcal{B}$.

To define the BPDS, $\mathcal{BP}$, for co-verification, we first define two labeling functions:

- $L_{\mathcal{P}2\mathcal{B}} : (G \times \Gamma) \rightarrow \Sigma$, which associates a configuration of $\mathcal{P}$, $\langle g, \gamma \rangle \in (G \times \Gamma)$, with the set of propositions that hold on it.
- $L_{\mathcal{B}2\mathcal{P}} : Q \rightarrow I$, which associates a state of $\mathcal{B}$ with the set of propositions that hold on it.

$\mathcal{BP} = ((G \times Q), \Gamma, \Delta', \langle (g_0, q_0), \omega_0 \rangle, F')$ is constructed by taking the Cartesian product of $\mathcal{B}$ and $\mathcal{P}$: $\langle (g, q), \gamma \rangle \hookrightarrow_{\mathcal{BP}} \langle (g', q'), \omega \rangle \in \Delta'$, where $q \xrightarrow{\sigma} q' \in \delta$, $\sigma \subseteq L_{\mathcal{P}2\mathcal{B}}(\langle g, \gamma \rangle)$ and $\langle g, \gamma \rangle \xhookrightarrow{\tau} \langle g', w \rangle \in \Delta$, $\tau \subseteq L_{\mathcal{B}2\mathcal{P}}(q)$. A configuration of $\mathcal{BP}$ is referred to as $\langle (g, q), \omega \rangle \in (G \times Q) \times \Gamma^*$. The set of all configurations is denoted as $Conf(\mathcal{BP})$. The labeling functions defines how $\mathcal{B}$ and $\mathcal{P}$ synchronize with each other. $\langle (g_0, q_0), \omega_0 \rangle$ is the initial configuration. $\langle (g, q), \omega \rangle \in F'$ if $q \in F$.

If $\langle (g, q), \gamma \rangle \hookrightarrow_{\mathcal{BP}} \langle (g', q'), \omega \rangle \in \Delta'$, then for every $v \in \Gamma^*$ the configuration $\langle (g, q), \gamma v \rangle$ is an immediate predecessor of $\langle (g', q'), \omega v \rangle$, and $\langle (g', q'), \omega v \rangle$ is an

immediate successor of $\langle (g, q), \gamma v \rangle$. A trace of $\mathcal{BP}$ is a sequence of configurations $\langle (g_0, q_0), \omega_0 \rangle, \langle (g_1, q_1), \omega_1 \rangle, \ldots, \langle (g_i, q_i), \omega_i \rangle, \ldots$ such that $\langle (g_i, q_i), \omega_i \rangle$ is an immediate predecessor of $\langle (g_{i+1}, q_{i+1}), \omega_{i+1} \rangle$, where $i \geq 0$. The reachability relation, $\Rightarrow_{\mathcal{BP}}$, is the reflexive and transitive closure of the immediate successor relation. Given a set $C \subseteq Conf(\mathcal{BP})$, the forward reachability analysis, $post^*(C)$, computes the successors of elements of C. In this paper, we are concerned with the reachability properties of $\mathcal{BP}$, i.e., given a configuration c and the initial configuration $c_0 = \langle (g_0, q_0), \omega_0 \rangle$, we want to check if $c \in post^*(\{c_0\})$.

4 Constructing BPDS from Co-specification

In this section, we discuss how to construct a BPDS model from the co-specification presented in Section 2. We assume that the hardware and software models in the co-specification are amenable to abstraction into BA and PDS. Without loss of generality, we describe the state space of the BPDS model using Boolean variables. Before we discuss how to construct the BPDS model, we introduce two tools for conducting predicate abstractions of hardware and software, respectively. The predicate abstraction tools help scale the verification but only preserve the safety properties of a system design, so we restrict the generated BPDS model for reachability analysis. It is important to note that (1) this approach is only one example on constructing BPDS and (2) the BPDS model proposed in Section 3 is not restricted to safety properties only.

4.1 Background

Predicate Abstraction of RTL Designs. Jain, et al. have presented a predicate abstraction algorithm for verifying RTL designs in Verilog [10]. The algorithm computes the abstraction of a Verilog module given certain predicates. The VCEGAR toolkit based on this algorithm generates hardware abstractions in the form of Boolean expressions (see example in Figure 6). This is one representation of state transition relations.

Predicate Abstraction of C Programs. Ball, et al. have shown Boolean programs to be effective abstractions of C programs in the SLAM project [11]. A Boolean program, conceptually a PDS, is essentially a C program in which the only data type available is Boolean. Given predicates, C2BP, the abstraction tool of SLAM, builds Boolean programs from C programs.

4.2 From Co-specification to BPDS

There are four steps to construct a BPDS model from the co-specification: (1) instrumenting the software model based on the HW/SW interface; (2) predicate abstraction of the instrumented software model using C2BP based on manually provided predicates; (3) instrumenting the hardware model based on the HW/SW interface; (4) predicate abstraction of the instrumented hardware model using VCEGAR based on manually provided predicates. The PDS (as a result of C2BP) and the BA (as a result of VCEGAR) are readily synchronized due to the instrumentation, thus forming a BPDS model.

```
UCHAR READ_REGISTER_UCHAR
            (PUCHAR Register) {
  switch(Register) {
    case BASE_ADDRESS+0x0: return read_PortA();
    ...
    case BASE_ADDRESS+0x4: return read_IntConfg();
    case BASE_ADDRESS+0x5: return read_IntStatus();
    default: abort "Register address error."; return 0;
  }
}
```

```
VOID WRITE_REGISTER_UCHAR
      (PUCHAR Register, UCHAR Value) {
  switch(Register) {
    case BASE_ADDRESS+0x0: write_PortA(Value); return;
    ...
    case BASE_ADDRESS+0x4: write_IntConfg(Value); return;
    case BASE_ADDRESS+0x5: write_IntStatus(Value); return;
    default: abort "Register address error."; return;
  }
}
```

Fig. 5. Redirecting read/write register calls to software interface events

At the software side, the instrumentation has three steps. First, we add the signatures of the software interface events into the driver program. Since the header of a software interface event is declared the same way as a C function, the signature of the interface event is simply its type signature. Second, we instrument the driver program to redirect the calls to the register read/write functions to the corresponding software interface events. Third, we instrument the driver to respond to the hardware interface events. Figure 5 shows an example instrumenting the PIO-24 digital I/O card driver, where the calls to two register read/write functions are replaced by calls to software interface events. As discussed in Section 2.2, the OS maintains a variable (*INTR* in the interface specification example) to indicate the interrupt pending status. When hardware fires an interrupt, i.e., a hardware interface event happens, the interrupt pending status is set true, so the OS schedules the ISR. We instrument the driver with a guarded expression at each program statement so that "if the interrupt pending status is true, non-deterministically call ISR". As a result, the context switch to ISR is simulated in the sequential software model. In a uni-processor system, the completeness of this approach is based on the assumption that the ISR cannot be switched out during execution. This is true for most Windows device drivers such as the PIO-24 digital I/O card driver. It is easy and theoretically sound to extend the instrumentation to support multiple ISRs with different priorities, because the number of ISRs in a system are finite. In the last step of software abstraction, we use C2BP to generate Boolean programs from the instrumented C programs. We convert the Boolean programs to PDS using Moped [6].

At the hardware side, we first convert the hardware model and the implementation of software interface events into Verilog modules. The non-deterministic function (`rand`) used in Figure 3 is not directly supported by Verilog. Since input variables of Verilog modules are treated as non-deterministic by VCEGAR, we construct non-deterministic functions using input variables. Second, we utilize VCEGAR to generate the predicate abstraction of the state transition relation for the hardware design (in the form of Verilog modules). Third, we then construct the BA as follows: (1) the alphabet Σ is the power set of the set of propositions induced by the software interface events; (2) the set of states Q are defined by the Boolean variables from the predicate abstraction; (3) the transition relation δ is the predicate abstraction, whose transitions are labeled with input symbols from Σ; (4) the set of final states F is set to Q, since we are interested in reachability only. As an example, Figure 6 shows the abstraction of the software interface events `read_IntStatus` and `write_IntConfg`, as hardware state transitions.

```
// predicates for read_IntStatus
decl b0; // stands for {IntStatus == 1}
decl b1; // stands for {retreg==1}

read_IntStatus
begin
  TRANS ( !next(b0) )
  TRANS ( (!b0 & !next(b1))
        | (b0 & next(b1)) )
end
```

```
// predicates for write_IntConfg
decl b0; // stands for {(4 & IntConfg) == 0}
decl b1; // stands for {(1 & PortA) == 0}
decl b2; // stands for {val == 5}

write_IntConfg
begin
  TRANS ( (!b1 & b0 & b2 & next(b0)) | (b1 & b0 & b2 & !next(b0))
        | (!b0 & b2 & !next(b0)) | (!b2) )
end
```

Fig. 6. Abstraction of software interface events as state transitions in the form of Boolean expressions (TRANS). The transitions are labeled corresponding to their software events respectively.

The constructed BPDS $\mathcal{BP}$ contains a PDS $\mathcal{P}$ representing software, a BA $\mathcal{B}$ representing hardware, and their synchronization, where $\mathcal{P}$ is from software abstraction, $\mathcal{B}$ is from hardware abstraction, and the synchronization by interface events is from the abstraction of the HW/SW interface (through instrumentation). The interface events have two directions, from $\mathcal{P}$ to $\mathcal{B}$ (referred to as software interface events) and from $\mathcal{B}$ to $\mathcal{P}$ (referred to as hardware interface events). In the formal model, the transitions of $\mathcal{B}$ are labeled corresponding to the software interface events and the transitions of $\mathcal{P}$ are labeled corresponding to the hardware interface events. Thus, we are able to synchronize $\mathcal{B}$ and $\mathcal{P}$. Before software abstraction, the signatures of software interface events are merged into the program, so $\mathcal{P}$ already contains the signatures. During the state transitions of $\mathcal{P}$, a software interface event happens when its dedicated stack symbol is reached. The BA transition that is enabled will be executed with the next PDS transition. The PDS transition also needs to be enabled by the current state of $\mathcal{B}$. For example, when $\mathcal{P}$ inputs from $\mathcal{B}$, the transition will depend on the state of $\mathcal{B}$. A hardware interface event happens when $\mathcal{B}$ transits to a state, which may cause a context switch in $\mathcal{P}$. Because $\mathcal{P}$ is a sequential PDS, we model the context switch by calling the function that services the hardware event, which is done during the step of software instrumentation.

Because the transitions of hardware and software are normally asynchronous except at their synchronization points, non-deterministic delays of either $\mathcal{B}$ or $\mathcal{P}$ should be allowed in the BPDS. Conceptually, the delays are introduced as self-loop transitions on the states of $\mathcal{B}$ or $\mathcal{P}$ where no interface event happens. When an interface event happens, both hardware and software have to transit synchronously.

5 Reachability Checking of BPDS

We have developed a tool, CoVer, for checking reachability properties of BPDS. As shown in Figure 7, CoVer has two components: (1) BPDS2PDS, which converts a BPDS model $\mathcal{BP}$ into a PDS model $\mathcal{P}'$; and (2) Moped [6], which checks reachability properties of $\mathcal{P}'$. Different from the PDS $\mathcal{P}$ in $\mathcal{BP}$, the new PDS model $\mathcal{P}'$ is a standard PDS in the sense that $\mathcal{P}'$ does not have inputs. The properties to be checked are provided to Moped through labeling states in the software PDS $\mathcal{P}$ and/or the hardware BA $\mathcal{B}$.

First, we present the conversion algorithm, BPDS2PDS, and argue that the conversion preserves the reachability properties of $\mathcal{BP}$. Second, we analyze the complexity of the conversion, the size of $\mathcal{P}'$ compared to $\mathcal{BP}$, and the verification complexity.

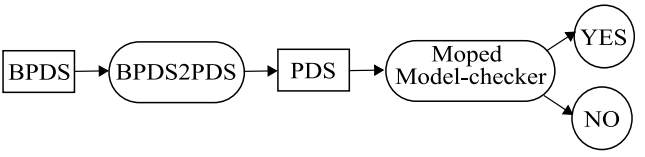

Fig. 7. CoVer: reachability checking of BPDS

5.1 Converting BPDS to PDS

The conversion works in such a way that the transition rules of $\mathcal{B}$ and $\mathcal{P}$ are selected and merged in $\mathcal{P}'$ depending on whether or not an interface event happens. We represent hardware transition rules as the union of two groups as $\delta = R_{model} \cup R_{evt}$. R_{model} is the set of rules that are not associated with software interface events. R_{evt} is the set of rules associated with software interface events. We define two functions: (1) $\text{HW}_{evt}(\tau)$ checks if the transition label τ of a rule in $\mathcal{P}$ is true for a hardware interface event. If yes, this rule services the hardware interface event, for instance, calling the ISR; (2) $\text{SW}_{evt}(\gamma)$ checks if γ is the stack symbol of a software interface event.

Algorithm 1 converts $\mathcal{BP}$ (given δ as rules of $\mathcal{B}$ and Δ as rules of $\mathcal{P}$) into a PDS $\mathcal{P}'$ with its rules as Δ'. Algorithm 1 explores the rules of $\mathcal{P}$ and $\mathcal{B}$ to build new PDS rules of $\mathcal{P}'$ based on the synchronization of $\mathcal{P}$ and $\mathcal{B}$. It terminates when all rules in δ and Δ are processed. For each transition rule in Δ, the algorithm has three choices: (1) if the transition handles a hardware interface event, the transition is merged with its corresponding transition in $\mathcal{B}$ to form a transition of $\mathcal{P}'$; (2) if it is a software interface event, the transition is merged with its corresponding transition in $\mathcal{B}$ as well. Hardware and software should always be synchronous on interface events; (3) when no interface event happens, the loop between lines 15-18 merges the transition of $\mathcal{P}$ with transitions of $\mathcal{B}$. Because hardware and software are asynchronous, the transition labels of $\mathcal{P}$ and

Algorithm 1. BPDS2PDS($\delta = R_{model} \cup R_{evt}$, Δ)

1: $\Delta' \leftarrow \emptyset$
2: **for all** $\langle g, \gamma\rangle \overset{\tau}{\hookrightarrow} \langle g', \omega\rangle \in \Delta$ **do**
3: **if** $\text{HW}_{evt}(\tau)$ **then**
4: {*If this PDS rule handles a hardware interface event*}
5: **for all** $q \overset{\sigma}{\rightarrow} q' \in R_{model}$ **and** $\sigma \subseteq L_{\mathcal{P}2\mathcal{B}}(\langle g, \gamma\rangle)$ **and** $\tau \subseteq L_{\mathcal{B}2\mathcal{P}}(q)$ **do**
6: $\Delta' \leftarrow \Delta' \bigcup \{\langle (g, q), \gamma\rangle \hookrightarrow \langle (g', q'), \omega\rangle\}$
7: **end for**
8: **else if** $\text{SW}_{evt}(\gamma)$ **then**
9: {*Else if this is a software interface event*}
10: **for all** $q \overset{\sigma}{\rightarrow} q' \in R_{evt}$ **and** $\sigma \subseteq L_{\mathcal{P}2\mathcal{B}}(\langle g, \gamma\rangle)$ **and** $\tau \subseteq L_{\mathcal{B}2\mathcal{P}}(q)$ **do**
11: $\Delta' \leftarrow \Delta' \bigcup \{\langle (g, q), \gamma\rangle \hookrightarrow \langle (g', q'), \omega\rangle\}$
12: **end for**
13: **else**
14: {*For transitions with no interface event*}
15: **for all** $q \overset{\sigma}{\rightarrow} q' \in R_{model}$ **do**
16: $\Delta' \leftarrow \Delta' \bigcup \{\langle (g, q), \gamma\rangle \hookrightarrow \langle (g, q'), \gamma\rangle\}$
17: $\Delta' \leftarrow \Delta' \bigcup \{\langle (g, q), \gamma\rangle \hookrightarrow \langle (g', q), \omega\rangle\}$
18: **end for**
19: **end if**
20: **end for**
21: **return** Δ'

$\mathcal{B}$ trivially hold on each other. There are four types of rules that can be generated for $\mathcal{P}'$ in the third condition: (1) $\mathcal{P}$ self-loops on its current state while $\mathcal{B}$ transits, which occurs in line 16; (2) $\mathcal{B}$ self-loops on its current state while $\mathcal{P}$ transits, which occurs in line 17; (3) Both $\mathcal{B}$ and $\mathcal{P}$ self-loop; (4) Both $\mathcal{B}$ and $\mathcal{P}$ transit. Rule (3) is trivial and can be eliminated. Rule (4) equals to consecutive transitions by Rules (1) and (2), because hardware and software execute asynchronously when no interface event happens.

$\mathcal{P}'$ preserves the reachability property of $\mathcal{BP}$. (1) The state space of $\mathcal{P}'$ equals to that of $\mathcal{BP}$; (2) The initial state of $\mathcal{P}'$ is the initial state of $\mathcal{BP}$; (3) We do not utilize the final states F' (the BA constraints) of $\mathcal{BP}$ in the reachability checking; (4) In Algorithm 1, it is clear that $\mathcal{P}'$ preserves all the transitions of both $\mathcal{B}$ and $\mathcal{P}$. Self-loop transitions are introduced for states of both $\mathcal{B}$ and $\mathcal{P}$ to model the asynchronous transitions between hardware and software. They do not affect the correctness of reachability checking.

5.2 Complexity Analysis

Algorithm 1 generates $O(|\Delta| \times |\delta|)$ PDS rules and has a time complexity of $O(|\Delta| \times |\delta|)$. The number of rules in $\mathcal{P}'$ is equal to the number of rules of $\mathcal{BP}$, because we add a rule to $\mathcal{P}'$ only if there is a corresponding rule in $\mathcal{BP}$. $\mathcal{P}'$ and $\mathcal{BP}$ have the same configurations because their state space is identical. We use Schwoon's $post^*$ algorithm [6] (implemented in Moped) to solve the reachability problems of $\mathcal{P}'$, so the time and space model-checking complexities on $\mathcal{P}'$ are $O((|G \times Q| + |\Delta \times \delta|)^3)$.

6 Evaluation

We first show an overall evaluation of our co-verification framework, where we succeeded in verifying the Windows driver and the hardware model of the PIO-24 digital I/O card, finding a previously undiscovered software bug – an "invalid read" bug. Then we discuss our experiments on evaluating the model-checking performance of our BPDS model. All experiments were run on a workstation with Intel Xeon 3GHz dual core CPU and 2GB physical memory.

For PIO-24, we abstract the hardware model (269 lines), the driver program (1724 lines), and the interface specification into a BPDS model. The verification detects a bug using 12 predicates and 4165 peak live Binary Decision Diagram (BDD) nodes in 0.02 seconds. The falsifying path that combines the execution of both hardware and software leads to a violation where a Deferred Procedure Call[1] (DPC) finishes the input request in success without actually reading the data from the device. As shown in Figure 2, the "invalid read" bug occurs when `DioIsr` interrupts `DioEvtDeviceControl` at `P1`, where `CurrentRequest` and `AwaitingInt` become inconsistent. `DioIsr` will not execute the `if` block at `P2` because `AwaitingInt` is `FALSE`. Later the DPC is requested at `P3`. The DPC sends data back to the application that generated the I/O request if `CurrentRequest` is not null, but the data is never actually read from the device. It is important to note that this bug cannot be detected when using a sequential model because the inconsistency of variable states only happens in HW/SW concurrent

[1] We omit the DPC implementation due to page limitation.

executions, as represented in our approach. Furthermore, because our approach to co-verification includes both hardware and software models, certain kinds of false bugs will not appear. For example, if the device's status is always interrupt-disabled as the driver reaches `P1`, the above described "invalid read" bug cannot happen. On the other hand, the verification of this property costs one person about 6 hours' manual efforts to construct the BPDS model. In order to avoid this overhead, the abstraction/refinement process needs to be fully automated (see Section 7).

We present an evaluation of the BPDS model based on synthetic programs derived from the template T shown in Figure 8 and hardware models derived from the template H shown in Figure 9. T is similar to the evaluation template used in [12] and later in [6]. The difference is that T operates a hardware counter which has global state. The templates allow us to generate a Boolean program $T(N)$ and its corresponding hardware model $H(N)$ for $N > 0$. $T(N)$ and $H(N)$ together have four global variables[2], where there are three variables (`a`, `b`, and `c`) representing the states of the hardware counter. $T(N)$ has $2N + 2$ procedures including software interface events: `main` as program entry point, `rd_reg` as a software interface event that returns the value of the most significant bit of the counter, N software interface events of the form `inc_reg<i>`, and N procedures of the form `level<i>` that call `rd_reg` and `inc_reg<i>`, where $0 < i \leq N$. For $0 < j < N$, the instances of `<stmt>` in the body of procedure `level<j>` are replaced by a call to procedure `level<j+1>`. The instances of `<stmt>` in the body of procedure `level<N>` are replaced by `skip`. $H(N)$ provides hardware transitions corresponding to the N software interface events `inc_reg<i>`, where the transitions increase the hardware counter by one. To further increase the complexity of the model for purposes of testing, we define an environment model that non-deterministically left-shifts the hardware counter by one bit. Templates T and H cover common scenarios where software operates hardware via interface events.

We compare the two approaches that model hardware using BA or using PDS. When using PDS, we model both hardware and software using a sequential program, similar to Monniaux's approach [4]. We model the environment and interface events as procedures such as `environment()` and `inc_reg<i>()`. The procedure `environment` is called after each software interface event to simulate the input from environment.

Table 1 shows the statistics on the co-verification models generated from Figure 8 and Figure 9, where the counter's size varies from 3 to 5 bits (i.e., the number of Boolean variables used by the counter). We force the reachability checking to be exhaustive, so the results represent the worst case performance. Statistics show that the PDS hardware model adds significant overhead to co-verification compared to the BA hardware model. For example, when $N = 4000$ and the counter has 5-bit size, the PDS hardware model generates 148k transition rules and has 22383 peak live BDD nodes, compared to 76k transition rules and 2115 peak live BDD nodes for the BA hardware model.

Table 2 shows the statistics on the model when interrupt checking is enabled. Compared to the result in Table 1, we use one more global variable to track whether an interrupt has fired. Similar to the procedure `environment()`, the ISR (not shown) calls interface events to left-shift the counter by one bit. The statistics show that although the

[2] Actually, we use three groups of templates. They differ in the size of the counter. For a counter with 3, 4, or 5 bits, we have 4, 5, or 6 global variables.

```
// a, b, c represent
// the hardware counter
decl g, a, b, c;
void main()
begin
   level1();
   level1();
   if (!g) then
      reach: skip;
   else
     skip;
   fi
end
```

```
void level<i>()
begin
   decl t;
   t := 0;
   if(g) then
      while(!t) do
         inc_reg<i>();
         t := rd_reg();
      od
   else
      <stmt>; <stmt>;
   fi
   g := !g;
end
```

```
bool rd_reg()
begin
  return c;
end

// Büchi automaton
// as hardware model
void inc_reg<i>()
begin
   skip;
end
```

```
// Using PDS as hardware model
void inc_reg<i>()
begin
   if (!a) then
      a := 1;
   elsif (!b) then
      a,b := 0,1;
   elsif (!c) then
      a,b,c := 0,0,1;
   else
      a,b,c := 0,0,0;
   fi
end
```

Fig. 8. Boolean program template T for evaluating the BPDS model

```
decl a,b,c;

inc_reg<i>
begin
  TRANS ( (!a & !b & !c & next(a) & !next(b) & !next(c))
        | (!a & !b & c & next(a) & !next(b) & next(c))
        | (!a & b & !c & next(a) & next(b) & !next(c))
        | (!a & b & c & next(a) & next(b) & next(c))
        | (a & !b & !c & !next(a) & next(b) & !next(c))
        | (a & !b & c & !next(a) & next(b) & next(c))
        | (a & b & !c & !next(a) & !next(b) & next(c))
        | (a & b & c & !next(a) & !next(b) & !next(c)) )
end
```

```
// Run non-deterministically
environment
begin
  TRANS( (b & next(a)) |
         (!b & !next(a)) )
  TRANS( (c & next(b)) |
         (!c & !next(b)) )
  TRANS( !next(c) )
end
```

```
// Using PDS as
// hardware model
bool environment()
begin
   l0: if(*) then
      a,b,c := b,c,0;
      goto l0;
   fi
end
```

Fig. 9. Hardware model template H for evaluating the BPDS model

Table 1. Comparison of co-verification statistics with BA and PDS hardware models (hardware does not interrupt software, and the size of global counter varies from 3 to 5 bits)

N	Time usage with BA HW model (Sec)			Time usage with PDS HW model (Sec)		
	3 bits	4 bits	5 bits	3 bits	4 bits	5 bits
1k	0.42	0.72	1.28	2.47	8.08	35.62
2k	0.91	1.47	2.64	5.09	16.31	71.83
3k	1.42	2.33	4.08	8.14	25.42	109.33
4k	1.91	3.11	5.50	10.75	33.70	144.22

Table 2. Statistics when interrupt checking is enabled (one more Boolean variable is used to track the interrupt status)

N	Time usage (Sec)			Peak live BDD nodes		
	3 bits	4 bits	5 bits	3 bits	4 bits	5 bits
1k	1.67	3.31	6.98	2335	5129	10325
2k	3.70	6.97	14.42	2335	5129	10337
3k	6.12	11.19	22.76	2335	5129	10325
4k	8.30	15.61	30.41	2335	5129	10337

full HW/SW concurrency is checked, as expected the verification complexities grow in the same order of magnitude as Table 1, where the complexities depend on the numbers of both global states and rules (the sizes of the program and hardware design). It can be inferred from Table 1 and Table 2 that using PDS as the hardware model without interrupt checking performs even worse than when using BA as the hardware model with interrupt checking. The time usage of Algorithm 1 that converts a BPDS to a PDS representation is very low. In our experiments, the maximum time usage is 0.45 seconds.

7 Conclusion and Future Work

In this paper, we have presented an automata-theoretic approach to co-verification. The core of this approach is a formal model for co-verification, the Büchi Pushdown System (BPDS). We have designed a co-specification framework for HW/SW interfaces and demonstrated a process of constructing a BPDS from the abstraction of hardware, software, and their interface specification. A BPDS can be converted to a PDS with the same complexities, so reachability analysis algorithms for PDS can be readily utilized to analyze BPDS. The evaluation has shown that BPDS is an effective model for co-verification. For the next step, we plan to automate the abstraction/refinement process of co-verification by integrating the abstraction/refinement engine of SLAM (C2BP for abstraction and Newton [11] for refinement) and the VCEGAR engine. One challenge for this integration is how to automatically propagate the predicates discovered by one engine across the HW/SW boundary to the other engine.

Acknowledgement. This research received financial support from National Science Foundation of the United States (Grant #: 0916968).

References

1. Clarke, E.M., Grumberg, O., Peled, D.: Model checking. MIT Press, Cambridge (1999)
2. Kurshan, R.P., Levin, V., Minea, M., Peled, D., Yenigün, H.: Combining software and hardware verification techniques. FMSD 21(3), 251–280 (2002)
3. Xie, F., Yang, G., Song, X.: Component-based hardware/software co-verification for building trustworthy embedded systems. JSS 80(5), 643–654 (2007)
4. Monniaux, D.: Verification of device drivers and intelligent controllers: a case study. In: Proc. of EMSOFT, pp. 30–36 (2007)
5. Ramalingam, G.: Context-sensitive synchronization-sensitive analysis is undecidable. ACM Trans. Program. Lang. Syst. 22(2), 416–430 (2000)
6. Schwoon, S.: Model-Checking Pushdown Systems. PhD thesis (2002)
7. Solomon, D.A.: Inside Windows NT, 2nd edn. Microsoft Press, Redmond (1998)
8. IEEE: IEEE Standard for Verilog (IEEE Std 1364-2005). IEEE (2005)
9. Kurshan, R.P.: Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach. Princeton University Press, Princeton (1994)
10. Jain, H., Kroening, D., Sharygina, N., Clarke, E.M.: Word-level predicate-abstraction and refinement techniques for verifying RTL Verilog. IEEE TCAD 27(2), 366–379 (2008)
11. Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S.K., Ustuner, A.: Thorough static analysis of device drivers. In: Proc. of EuroSys, pp. 73–85 (2006)
12. Ball, T., Rajamani, S.K.: Bebop: A symbolic model checker for boolean programs. In: Proc. of SPIN, pp. 113–130 (2000)