Efficient Reachability Analysis of Büchi Pushdown Systems for Hardware/Software Co-verification

Juncao Li[1], Fei Xie[1], Thomas Ball[2], and Vladimir Levin[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}@microsoft.com

Abstract. We present an efficient approach to reachability analysis of Büchi Pushdown System (BPDS) models for Hardware/Software (HW/SW) co-verification. This approach utilizes the asynchronous nature of the HW/SW interactions to reduce unnecessary HW/SW state transition orders being explored in co-verification. The reduction is applied when the verification model is constructed. We have realized this approach in our co-verification tool, CoVer, and applied it to the co-verification of two fully functional Windows device drivers with their device models respectively. Both of the drivers are open source and their original C code has been used. CoVer has proven seven safety properties and detected seven previously undiscovered software bugs. Evaluation shows that the reduction can significantly scale co-verification.

1 Introduction

Hardware/Software (HW/SW) co-verification, verifying hardware and software together, is essential to establishing the correctness of complex computer systems. In previous work, we proposed a Büchi Pushdown System (BPDS) as a formal representation for co-verification [1], a Büchi Automaton (BA) represents a hardware device model and a Labeled Pushdown System (LPDS) represents a model of the system software. The interactions between hardware and software take place through the synchronization of the BA and LPDS. The BPDS is amenable to standard symbolic model checking algorithms [2].

In this paper, we exploit the fact that hardware and software are mostly asynchronous in a system to reduce the cost of model checking. Intuitively, when hardware and software transition asynchronously (i.e. there are no HW/SW interactions), it is unnecessary to explore all the possible state transition orders. Furthermore, we prove that special cases of the transition orders preserve the reachability properties in question. Partial order reduction identifies such special transition orders, so there are fewer interleaving possibilities to be explored during model checking. We base our approach on the concept of static partial order reduction [3], where unnecessary transition orders are pruned during the construction of the verification model. During the model construction, unnecessary transition orders are largely reduced when hardware and software are asynchronous. On the other hand, all the synchronous transitions are preserved.

We implemented our approach in the co-verification tool CoVer and applied it to the co-verification of two fully functional Windows device drivers (C programs for which source code is publically available) with their device models. We specify the device models based on the HW/SW interface documents that are openly available. Conceptually, a driver and its device model together form a BPDS model. CoVer converts the driver and the device model into a C program and utilizes the SLAM engine [4] to check reachability properties of the program. The abstraction/refinement process is carried out by SLAM. CoVer proved seven properties and detected seven real defects in the two drivers. All of these defects can cause serious system failures including data loss, interrupt storm, device hang, etc.

The rest of this paper is organized as follows. Section 2 reviews related work. Section 3 introduces the background of this paper. Section 4 presents our reachability analysis algorithm for BPDS models. Section 5 discusses how we specify a device model as well as the implementation details of CoVer. Section 6 presents the evaluation results. Section 7 concludes and discusses future work.

2 Related Work

Kurshan, et al. presented a co-verification framework that models hardware and software designs using finite state machines [5]. Xie, et al. extended this framework to hardware and software implementations and improved its scalability via component-based co-verification [6]. 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 [7]. He modeled a USB host controller device using a C program and instrumented the device driver, another C program, in such a way as to verify that the USB host controller driver correctly interacts with the device. The hardware and software were both modeled by C programs and thus are formally PDSs. 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 [8].

Bouajjani et al. [9] presented a procedure to compute predecessor reachability of PDS and apply this procedure to linear/branching-time property verification. This approach was improved by Schwoon [2], which results in a tool, Moped, for checking Linear Temporal Logic (LTL) properties of PDS. A LTL formula is first negated and then represented as a BA. The BA is combined with the PDS to monitor its state transitions; therefore the model checking problem is to compute if the BA has an accepting run. The goal of the previous research was to verify software only; however, our goal is to co-verify HW/SW systems.

Our previous work [1] did not exploit the fact that hardware and software are mostly asynchronous in a system. Techniques such as partial order reduction [10] can be applied to reduce the verification complexities via the composition (Cartesian product) of the BA and LPDS. Furthermore, our co-verification implementation in our earlier work was not automatic since it depends on two abstraction/refinement engines (for hardware and software specifications respectively) that were not completely integrated.

3 Background

3.1 Büchi Automaton (BA)

A *BA* $\mathcal{B}$, as defined in [11], is a non-deterministic finite state automaton accepting infinite input strings. Formally, $\mathcal{B} = (\Sigma, Q, \delta, q_0, F)$, 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 iff it has a run over the string that visits 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.

3.2 Labeled Pushdown System (LPDS)

A *LPDS* $\mathcal{P}$, as defined in [1], is a tuple $(I, G, \Gamma, \Delta, \langle g_0, \omega_0 \rangle)$, where I is the input alphabet, G is a finite set of global states, Γ is a finite stack alphabet, $\Delta \subseteq (G \times \Gamma) \times I \times (G \times \Gamma^*)$ is a finite set of transition rules, and $\langle g_0, \omega_0 \rangle$ is the initial configuration. LPDS is an extension of PDS [2] in such a way that a LPDS can take inputs. A LPDS transition rule is written as $\langle g, \gamma \rangle \overset{\tau}{\hookrightarrow} \langle g', w \rangle$, where $\tau \in I$ and $((g, \gamma), \tau, (g', w)) \in \Delta$. 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 $Conf(\mathcal{P})$. The head of a configuration $c = \langle g, \gamma v \rangle$ is $\langle g, \gamma \rangle$ and denoted as $head(c)$, where $\gamma \in \Gamma, v \in \Gamma^*$; the head of a rule $r : \langle g, \gamma \rangle \overset{\tau}{\hookrightarrow} \langle g', \omega \rangle$ is $\langle g, \gamma \rangle$ and denoted as $head(r)$. The head of a configuration decides the transition rules that are applicable to this configuration, where the deciding factors are the global state and the top stack symbol. Given a rule $r : \langle g, \gamma \rangle \overset{\tau}{\hookrightarrow} \langle g', \omega \rangle$, 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 $\langle g, \gamma v \rangle$. We denote the immediate successor relation in PDS as $\langle g, \gamma v \rangle \overset{\tau}{\Rightarrow} \langle g', \omega v \rangle$, where we say this state transition follows the PDS rule r. The reachability relation, $\Rightarrow^*$, is the reflexive and transitive closure of the immediate successor relation. A path of $\mathcal{P}$ on an infinite input string, $\tau_0 \tau_1 \ldots \tau_i \ldots$, is written as $c_0 \overset{\tau_0}{\Rightarrow} c_1 \overset{\tau_1}{\Rightarrow} \ldots c_i \overset{\tau_i}{\Rightarrow} \ldots$, where $c_i \in Conf(\mathcal{P}), i \geq 0$. The path is also referred to as a trace of $\mathcal{P}$ if $c_0 = \langle g_0, \omega_0 \rangle$ is the initial configuration.

3.3 Büchi Pushdown System (BPDS)

A *BPDS* $\mathcal{BP}$, as defined in [1], is the Cartesian product of a BA $\mathcal{B}$ and a LPDS $\mathcal{P}$. To construct $\mathcal{BP}$, we first define (1) the input alphabet of $\mathcal{B}$ as the power set of the set of propositions that may hold on a configuration of $\mathcal{P}$ (i.e. a symbol in Σ is a set of propositions); (2) the input alphabet of $\mathcal{P}$ as the power set of the set of propositions that may hold on a state of $\mathcal{B}$ (i.e. a symbol in I is a set of propositions); and (3) two labeling functions as follows:

- $L_{\mathcal{P}2\mathcal{B}} : (G \times \Gamma) \to \Sigma$, associates the head of a LPDS configuration with the set of propositions that hold on it. Given a configuration $c \in Conf(\mathcal{P})$, we write $L_{\mathcal{P}2\mathcal{B}}(c)$ instead of $L_{\mathcal{P}2\mathcal{B}}(head(c))$ for simplicity in the rest of this paper.
- $L_{\mathcal{B}2\mathcal{P}} : Q \to I$, 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 $\mathcal{P}$. A BPDS rule $\langle(g, q), \gamma\rangle \hookrightarrow_{\mathcal{BP}} \langle(g', q'), \omega\rangle \in \Delta'$ iff $q \overset{\sigma}{\to} q' \in \delta$, $\sigma \subseteq L_{\mathcal{P}2\mathcal{B}}(\langle g, \gamma\rangle)$ and $\langle g, \gamma\rangle \overset{\tau}{\hookrightarrow} \langle g', w\rangle \in \Delta$, $\tau \subseteq L_{\mathcal{B}2\mathcal{P}}(q)$. 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 define 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 $q \in F$.

Given a BPDS rule $r : \langle(g, q), \gamma\rangle \hookrightarrow_{\mathcal{BP}} \langle(g', q'), \omega\rangle \in \Delta'$, 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 $\langle(g, q), \gamma v\rangle$. We denote the immediate successor relation in BPDS as $\langle(g, q), \gamma v\rangle \Rightarrow_{\mathcal{BP}} \langle(g', q'), \omega v\rangle$, where we say this state transition follows the BPDS rule r. The reachability relation, $\Rightarrow^*_{\mathcal{BP}}$, is the reflexive and transitive closure of the immediate successor relation. A path of $\mathcal{BP}$ is a sequence of BPDS configurations, $c_0 \Rightarrow_{\mathcal{BP}} c_1 \ldots \Rightarrow_{\mathcal{BP}} c_i \Rightarrow_{\mathcal{BP}} \ldots$, where $c_i \in Conf(\mathcal{BP}), i \geq 0$. The path is also referred to as a trace of $\mathcal{BP}$ if $c_0 = \langle(g_0, q_0), \omega_0\rangle$ is the initial configuration.

We define four concepts to assist us in analyzing the Cartesian product of $\mathcal{B}$ and $\mathcal{P}$:

Enabledness. A BPDS $\mathcal{BP}$ is constructed by synchronizing a BA $\mathcal{B}$ and a LPDS $\mathcal{P}$ through the labels on their state transitions. A Büchi transition $t_{\mathcal{B}} : q \overset{\sigma}{\to} q'$ is enabled by a LPDS configuration c (resp. a LPDS rule $r : c \overset{\tau}{\hookrightarrow} c'$) iff $\sigma \subseteq L_{\mathcal{P}2\mathcal{B}}(c)$; otherwise $t_{\mathcal{B}}$ is disabled by c (resp. r). The LPDS rule r is enabled/disabled by the Büchi state q (resp. the Büchi transition $t_{\mathcal{B}}$) in a similar way.

Indistinguishability. Given a Büchi transition $t_{\mathcal{B}} : q \overset{\sigma}{\to} q' \in \delta$, two LPDS configurations $c, c' \in Conf(\mathcal{P})$ are (resp. a LPDS rule $r : c \overset{\tau}{\hookrightarrow} c'$ is) indistinguishable to $t_{\mathcal{B}}$ iff $\sigma \subseteq L_{\mathcal{P}2\mathcal{B}}(c) \cap L_{\mathcal{P}2\mathcal{B}}(c')$, i.e. $t_{\mathcal{B}}$ is enabled by both c and c'. On the other hand, given a LPDS rule $r : c \overset{\tau}{\hookrightarrow} c' \in \Delta$, two Büchi states $q, q' \in Q$ are (resp. a Büchi transition $t_{\mathcal{B}} : q \overset{\sigma}{\to} q'$ is) indistinguishable to r iff $\tau \subseteq L_{\mathcal{B}2\mathcal{P}}(q) \cap L_{\mathcal{B}2\mathcal{P}}(q')$, i.e. r is enabled by both q and q'.

Consider a BPDS state transition, $t_{\mathcal{BP}} : \langle(g, q), \gamma v\rangle \Rightarrow_{\mathcal{BP}} \langle(g', q'), \omega v\rangle$ ($v \in \Gamma^*$), which is the combination of $t_{\mathcal{B}} : q \overset{\sigma}{\to} q' \in \delta$ and $t_{\mathcal{P}} : \langle g, \gamma v\rangle \overset{\tau}{\Rightarrow} \langle g', \omega v\rangle$ that follows a LPDS rule $r \in \Delta$. If the Büchi states q and q' (resp. LPDS configurations $\langle g, \gamma v\rangle$ and $\langle g', \omega v\rangle$) are both indistinguishable to r (resp. $t_{\mathcal{B}}$), $t_{\mathcal{BP}}$ can be rewritten as a BPDS path $\langle(g, q), \gamma v\rangle \Rightarrow_{\mathcal{BP}} \langle(g, q'), \gamma v\rangle \Rightarrow_{\mathcal{BP}} \langle(g', q'), \omega v\rangle$ (resp. $\langle(g, q), \gamma v\rangle \Rightarrow_{\mathcal{BP}} \langle(g', q), \omega v\rangle \Rightarrow_{\mathcal{BP}} \langle(g', q'), \omega v\rangle$), where the concurrent state transitions of $\mathcal{B}$ and $\mathcal{P}$ are represented in an interleaved fashion with one intermediate state used.

Independence. Given a Büchi transition $t_{\mathcal{B}}$ and a LPDS rule r, if they are indistinguishable to each other, $t_{\mathcal{B}}$ and r are called independent; otherwise if either $t_{\mathcal{B}}$ or r is not indistinguishable to the other but they still enable each other, $t_{\mathcal{B}}$ and r are called dependent. The independence relation is symmetric. Furthermore, if $t_{\mathcal{B}}$ and r are dependent, (1) the BA $\mathcal{B}$ and LPDS $\mathcal{P}$ are called synchronous on them; and (2) the corresponding BPDS transitions are called synchronous transitions; otherwise if $t_{\mathcal{B}}$ and r are independent, (1) $\mathcal{B}$ and $\mathcal{P}$ are called asynchronous on them; and (2) the corresponding BPDS transitions are called asynchronous transitions.

Commutativity. Without affecting the reachability property, if a BPDS state transition, $t_{\mathcal{BP}} : \langle(g, q), \gamma v\rangle \Rightarrow_{\mathcal{BP}} \langle(g', q'), \omega v\rangle$ can be rewritten respectively as two BPDS paths

such that $\langle (g,q),\gamma v\rangle \Rightarrow_{\mathcal{BP}} \langle (g,q'),\gamma v\rangle \Rightarrow_{\mathcal{BP}} \langle (g',q'),\omega v\rangle$ and $\langle (g,q),\gamma v\rangle \Rightarrow_{\mathcal{BP}} \langle (g',q),\omega v\rangle \Rightarrow_{\mathcal{BP}} \langle (g',q'),\omega v\rangle$, the corresponding Büchi transition $t_{\mathcal{B}}$ and LPDS rule r are called commutative. By definition, commutativity is equivalent to independence but seen under a different light, which will help the presentation of the paper.

3.4 Static Partial Order Reduction

One common method for reducing the complexity of model checking asynchronous systems is partial order reduction [10], which is based on the observation that properties often do not distinguish among the state transition orders. Traditional partial order reduction algorithms use an explicit state representation and depth first search, where both the states and transitions to be explored are selected during the model checking process. Kurshan et al. [3] developed an alternative approach called static partial order reduction, where the key idea is to apply partial order reduction when a model is generated from the system specification. Thus, no modification to the model checker is necessary. The model is reduced during the compilation phase by exploring the structure of the system specification. Any model checker that accepts this kind of model can then be applied to solve the verification problem.

4 Reachability Analysis of BPDS

4.1 Reachability Analysis of BPDS without Reduction

For reachability analysis, we have demonstrated [1] that a BPDS $\mathcal{BP}$ can be converted into a PDS $\mathcal{P}'$, which we refer to as the verification model, so that model checkers for PDS (or PDS-equivalent models) can be readily utilized. It is important to note that $\mathcal{P}'$ is a standard PDS in the sense that $\mathcal{P}'$ does not have inputs. Given $\mathcal{BP} = ((G \times Q), \Gamma, \Delta', \langle (g_0,q_0),\omega_0\rangle, F')$, we construct $\mathcal{P}' = (G_{\mathcal{P}'}, \Gamma_{\mathcal{P}'}, \Delta_{\mathcal{P}'}, c_0)$ such that $G_{\mathcal{P}'} = (G \times Q)$, $\Gamma_{\mathcal{P}'} = \Gamma$, $c_0 = \langle (g_0,q_0),\omega_0\rangle$, and $\Delta_{\mathcal{P}'}$ is converted from $\Delta' = \delta \times \Delta$, where $\forall t = q \xrightarrow{\sigma} q' \in \delta$ and $\forall r = \langle g,\gamma\rangle \overset{\tau}{\hookrightarrow} \langle g',\omega\rangle \in \Delta$, if t and r are dependent, add a rule $\langle (g,q),\gamma\rangle \hookrightarrow \langle (g',q'),\omega\rangle$ to $\Delta_{\mathcal{P}'}$, i.e. $\mathcal{B}$ and $\mathcal{P}$ must transition synchronously; else if t and r are independent, add three rules to $\Delta_{\mathcal{P}'}$: (1) $\langle (g,q),\gamma\rangle \hookrightarrow \langle (g,q'),\gamma\rangle$, i.e. $\mathcal{B}$ transitions and $\mathcal{P}$ loops; (2) $\langle (g,q),\gamma\rangle \hookrightarrow \langle (g',q),\omega\rangle$, i.e. $\mathcal{P}$ transitions and $\mathcal{B}$ loops; and (3) $\langle (g,q),\gamma\rangle \hookrightarrow \langle (g',q'),\omega\rangle$, i.e. $\mathcal{B}$ and $\mathcal{P}$ transition together. Rules (1) and (2) represent the non-deterministic delays that may occur between $\mathcal{B}$ and $\mathcal{P}$. Non-deterministic delays do not affect reachability properties. Rule (3) can be represented by Rules (1) and (2) together because $\mathcal{B}$ and $\mathcal{P}$ are asynchronous; however we include Rule (3) here to help the presentation of Section 4.2. The correctness of the conversion that $\mathcal{P}'$ preserves the reachability property of $\mathcal{BP}$ is proved in [1].

4.2 Efficient Reachability Analysis based on Static Partial Order Reduction

As discussed above, when a BPDS $\mathcal{BP}$ is converted to a PDS $\mathcal{P}'$ by the naïve approach, both the size of the state space and the number of the transition rules remain the same. For example, the set of transition rules is the product of δ that belongs to $\mathcal{B}$ and Δ that

belongs to $\mathcal{P}$. However, a complete product is not necessary when $\mathcal{B}$ and $\mathcal{P}$ are asynchronous. Without affecting the verification result, static partial order reduction can be applied to reduce the transition rules generated by the product. The reduced PDS model $\mathcal{P}'_r$ will have a smaller set of transition rules $\Delta_{\mathcal{P}'_r} \subseteq \Delta_{\mathcal{P}'}$ and fewer state transition traces while still preserving the reachability properties of $\mathcal{P}'$. Figure 1 illustrates the verification process that supports the reduction.

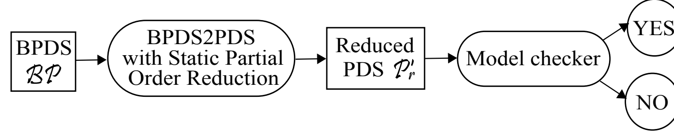

Fig. 1. Reachability analysis of BPDS with static partial order reduction.

Our reduction is based on the observation that when $\mathcal{B}$ and $\mathcal{P}$ transition asynchronously, one can run continuously while the other one loops. Figure 2 illustrates the idea of reducing a BPDS state transition graph that starts from the configuration $c_{0,0}$. Figure 2(a) is a complete state transition graph. There are three types of transition

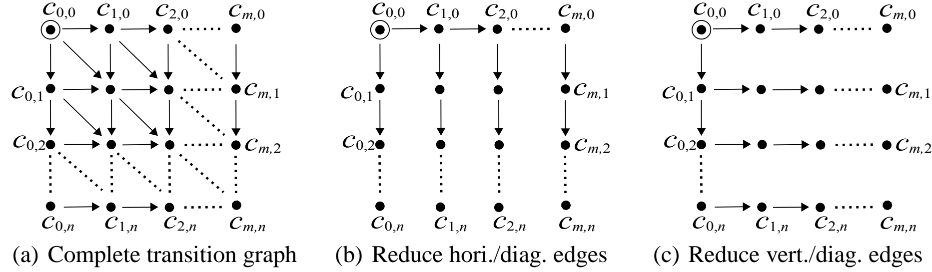

Fig. 2. Reducing state transition edges without affecting the reachability from $c_{0,0}$ when BA and LPDS are asynchronous.

edges: (1) a horizontal edge represents a transition when $\mathcal{B}$ transitions and $\mathcal{P}$ loops, which follows a BPDS rule in the form of $\langle (g, q), \gamma\rangle \hookrightarrow_{\mathcal{BP}} \langle (g, q'), \gamma\rangle$; (2) a vertical edge represents a transition when $\mathcal{P}$ transitions and $\mathcal{B}$ loops, which follows a BPDS rule in the form of $\langle (g, q), \gamma\rangle \hookrightarrow_{\mathcal{BP}} \langle (g', q), w\rangle$; and (3) a diagonal edge represents a transition when $\mathcal{B}$ and $\mathcal{P}$ transition together, which follows a BPDS rule in the form of $\langle (g, q), \gamma\rangle \hookrightarrow_{\mathcal{BP}} \langle (g', q'), w\rangle$. For every configuration $c_{i,j} = \langle (g, q), \gamma v\rangle$ ($0 \leq i \leq m$ and $0 \leq j \leq n$) as well as the Büchi transition $t_{\mathcal{B}} : q \xrightarrow{\sigma} q'$ and the LPDS rule $r : \langle g, \gamma\rangle \overset{\tau}{\hookrightarrow} \langle g', \omega\rangle$ that are both enabled on $c_{i,j}$, if $t_{\mathcal{B}}$ and r are independent, we can reduce many state transitions in Figure 2(a) without affecting the reachability from $c_{0,0}$ to other configurations in the graph. Figure 2(b) and Figure 2(c) illustrate two reductions that reduce horizontal/diagonal transition edges and vertical/diagonal transition

edges respectively. This kind of reduction can significantly reduce the transition rules of $\mathcal{BP}$, where Büchi transitions and LPDS rules are independent.

Now we present an optimization of our previous approach, where the reduction is applied during the rule generation phase of constructing the verification model $\mathcal{P}'_r$. We define a set of heads, $SensitiveSet$, on $Conf(\mathcal{P})$ as follows:

Definition 1. *$SensitiveSet = \{\ head(\langle g_0, \omega_0\rangle)\ \} \bigcup \{\ head(c') \mid \exists r = c \overset{\tau}{\hookrightarrow} c' \in \Delta$, $\exists t_{\mathcal{B}} \in \delta$, r and $t_{\mathcal{B}}$ are dependent $\}$, where $\langle g_0, \omega_0\rangle$ is the initial configuration of $\mathcal{P}$.*

The concept of $SensitiveSet$ is similar to that of sleep set [10]. However, instead of identifying transitions that are not necessary to be executed (i.e. reducible) at a state, $SensitiveSet$ identifies transitions that should be kept (i.e. irreducible). Algorithm 1 applies the reduction following the idea illustrated in Figure 2(b), where the horizontal/diagonal edges are reduced. If the LPDS rule r and the Büchi transition $t_{\mathcal{B}}$ are dependent, $\mathcal{B}$ and $\mathcal{P}$ must transition synchronously as the set of rules, Δ_{sync}, generated in line 6; otherwise, asynchronous transitions are generated. The set of rules, Δ_{vert}, generated in line 9 represent the vertical edges, i.e. when $\mathcal{P}$ transitions and $\mathcal{B}$ loops. The set of rules, Δ_{hori}, representing the horizontal edges, i.e. when $\mathcal{B}$ transitions and $\mathcal{P}$ loops, are generated in line 12 only if $head(r)$ belongs to $SensitiveSet$.

Algorithm 1 BPDS2PDS_SPOR($\delta \times \Delta$)

1: $\Delta_{sync} \leftarrow \emptyset$, $\Delta_{vert} \leftarrow \emptyset$, $\Delta_{hori} \leftarrow \emptyset$
2: **for all** $r : \langle g, \gamma\rangle \overset{\tau}{\hookrightarrow} \langle g', \omega\rangle \in \Delta$ **do**
3:   **for all** $t_{\mathcal{B}} : q \overset{\sigma}{\to} q' \in \delta$ **and** $\sigma \subseteq L_{\mathcal{P}2\mathcal{B}}(\langle g, \gamma\rangle)$ **and** $\tau \subseteq L_{\mathcal{B}2\mathcal{P}}(q)$ **do**
4:     **if** r and $t_{\mathcal{B}}$ are dependent **then**
5:       {*When $\mathcal{B}$ and $\mathcal{P}$ are synchronous on r and $t_{\mathcal{B}}$*}
6:       $\Delta_{sync} \leftarrow \Delta_{sync} \bigcup \{\langle (g, q), \gamma\rangle \hookrightarrow \langle (g', q'), \omega\rangle\}$
7:     **else**
8:       {*For vertical edges (see Figure 2(b)), when $\mathcal{P}$ transitions and $\mathcal{B}$ loops*}
9:       $\Delta_{vert} \leftarrow \Delta_{vert} \bigcup \{\langle (g, q), \gamma\rangle \hookrightarrow \langle (g', q), \omega\rangle\}$
10:       **if** $\langle g, \gamma\rangle \in SensitiveSet$ **then**
11:         {*For horizontal edges (see Figure 2(b)), when $\mathcal{B}$ transitions and $\mathcal{P}$ loops*}
12:         $\Delta_{hori} \leftarrow \Delta_{hori} \bigcup \{\langle (g, q), \gamma\rangle \hookrightarrow \langle (g, q'), \gamma\rangle\}$
13:       **end if**
14:     **end if**
15:   **end for**
16: **end for**
17: $\Delta_{\mathcal{P}'_r} \leftarrow \Delta_{sync} \bigcup \Delta_{vert} \bigcup \Delta_{hori}$
18: **return** $\Delta_{\mathcal{P}'_r}$

In Algorithm 1, a diagonal rule is reduced if $\mathcal{B}$ and $\mathcal{P}$ are asynchronous on the corresponding Büchi transition and LPDS rule. This kind of reduction does not affect any reachability property, because the diagonal rule can be represented by one horizontal rule and one vertical rule respectively. A horizontal rule is reduced if the head of the corresponding LPDS rule in $\mathcal{P}$ does not belong to $SensitiveSet$. There is a special set

of heads, $DivideSet = \{ h \mid h \in SensitiveSet, \forall r = c \overset{\tau}{\hookrightarrow} c' \in \Delta$ and $\forall t_{\mathcal{B}} \in \delta$, if $head(c) = h$ then r and $t_{\mathcal{B}}$ are not dependent $\}$. Informally, $DivideSet$ describes a set of configurations that can be considered as divide-lines (in the traces of $\mathcal{P}$ projected from the traces of $\mathcal{BP}$) for two adjacent LPDS transitions that are respectively synchronous and asynchronous with the state transitions of $\mathcal{B}$. Given a trace of $\mathcal{P}'_r$ in the form of $\langle (g_0, q_0), \omega_0 \rangle \Rightarrow \ldots \Rightarrow \langle (g_j, q_j), \omega_j \rangle \Rightarrow \ldots \Rightarrow \langle (g_k, q_k), \omega_k \rangle \Rightarrow \ldots$ $(0 \leq j < k)$, if $head(\langle g_j, \omega_j \rangle) \in DivideSet$ and $\langle (g_k, q_k), \omega_k \rangle$ is the first configuration satisfying $head(\langle g_k, \omega_k \rangle) \in SensitiveSet$ after $\langle (g_j, q_j), \omega_j \rangle$, we can infer that no horizontal transition occurs between $\langle (g_{j+1}, q_{j+1}), \omega_{j+1} \rangle$ and $\langle (g_k, q_k), \omega_k \rangle$ in the trace (i.e. $q_{j+1} = q_k$), because the horizontal transitions have been reduced.

Theorem 1. *$\mathcal{P}'_r$ preserves the reachability of $\mathcal{P}'$ from the initial configuration.*

Proof. It is easy to observe that $\mathcal{P}'_r$ and $\mathcal{P}'$ have the same state space and initial configuration, so the question is to prove that (1) given a configuration c and a trace of $\mathcal{P}'$ in the form of $T : c_0 \Rightarrow c_1 \ldots \Rightarrow c_i \Rightarrow c$, there is a corresponding trace of $\mathcal{P}'_r$ such that $T' : c_0 \Rightarrow c'_1 \ldots \Rightarrow c'_j \Rightarrow c$; and (2) vice versa.

Two types of transitions are reduced in $\mathcal{P}'_r$, compared to $\mathcal{P}'$. As explained above, the reduction of diagonal transitions does not affect any reachability property. We prove that the reduction of horizontal transitions does not affect the correctness of (1) by induction. If $|T| = 0$, i.e. $c = c_0$, the reachability trivially holds on $\mathcal{P}'_r$. If $|T| = 1$, because there is no horizontal transition reduced on the initial configuration, for any transition $c_0 \Rightarrow c$ of $\mathcal{P}'$ there must be a corresponding trace of $\mathcal{P}'_r$ that preserves the reachability. Given a trace $T : c_0 \Rightarrow c_1 \ldots \Rightarrow c_i \Rightarrow c'$ $(i \geq 0)$ of $\mathcal{P}'$ where $|T| = i+1$, if there exists a trace $T' : c_0 \Rightarrow c'_1 \ldots \Rightarrow c'_j \Rightarrow c'$ $(j \geq 0)$ of $\mathcal{P}'_r$ where $|T'| = j + 1$, we show that for all $c \in Conf(\mathcal{P}')$ and $t_{\mathcal{P}'} : c' \Rightarrow c$ of $\mathcal{P}'$, there is a trace of $\mathcal{P}'_r$ such that $c_0 \Rightarrow^* c$. Recall that the horizontal transitions are reduced in $\mathcal{P}'_r$ except at configurations whose heads belong to $SensitiveSet$, so we need to prove that this reduction does not affect the reachability if $t_{\mathcal{P}'}$ involves a horizontal transition that is reduced in $\mathcal{P}'_r$. In the trace T', we can always find a configuration $c'_k = \langle (g_k, q_k), \omega_k \rangle$ $(0 \leq k \leq j)$ such that c'_k is the last configuration satisfying $head(\langle g_k, \omega_k \rangle) \in SensitiveSet$. Thus, the path from c'_k to c' has the form of $(c'_k : \langle (g_k, q_k), \omega_k \rangle) \Rightarrow \langle (g_{k+1}, q_k), \omega_{k+1} \rangle \Rightarrow \ldots \Rightarrow (c' : \langle (g_{j+1}, q_k), \omega_{j+1} \rangle)$, where $\mathcal{B}$ always loops at the state q_k after c'_k. Because the horizontal transitions are reduced on the configurations after c'_k, $\mathcal{P}'_r$ cannot directly have the transition $(c' : \langle (g_{j+1}, q_k), \omega_{j+1} \rangle) \Rightarrow (c : \langle (g_{j+1}, q_{k+1}), \omega_{j+1} \rangle)$, i.e. the corresponding BPDS rule $\langle (g_{j+1}, q_k), \gamma_{j+1} \rangle \hookrightarrow_{\mathcal{BP}} \langle (g_{j+1}, q_{k+1}), \gamma_{j+1} \rangle$ (γ_{j+1} is the top stack symbol of ω_{j+1}) does not exist after the reduction. According to the commutativity between independent Büchi transitions and LPDS rules, we can shift this transition backward to the position right after c'_k where the horizontal transitions are not reduced. In this case, the path is $\underline{(c'_k : \langle (g_k, q_k), \omega_k \rangle) \Rightarrow \langle (g_k, q_{k+1}), \omega_k \rangle} \Rightarrow \langle (g_{k+1}, q_{k+1}), \omega_{k+1} \rangle \Rightarrow \ldots \Rightarrow$ $(c : \langle (g_{j+1}, q_{k+1}), \omega_{j+1} \rangle)$, so we proved that there is a trace $c_0 \Rightarrow^* c$ of $\mathcal{P}'_r$.

On the other direction, (2) always holds because $\Delta_{\mathcal{P}'_r} \subseteq \Delta_{\mathcal{P}'}$. For every rule of $\mathcal{P}'_r$, $\mathcal{P}'$ has the same rule. Thus for every trace of $\mathcal{P}'_r$, $\mathcal{P}'$ has the same trace. □

Complexity analysis. Let n_{SR} be the number of LPDS rules (in Δ) whose heads belong to $SensitiveSet$, and n_{sync} be the number of PDS rules (in $\Delta_{\mathcal{P}'_r}$) where $\mathcal{B}$ and $\mathcal{P}$ transition synchronously on the corresponding Büchi transitions and LPDS rules. We

have $|\Delta_{hori}| = n_{SR} \times |\delta|$ and $|\Delta_{sync}| = n_{sync}$. As illustrated in Figure 2, asynchronous transitions can be organized as triples where each one includes a vertical transition, a horizontal transition, and a diagonal transition, so we have $|\Delta_{vert}| = \frac{|\delta \times \Delta| - n_{sync}}{3}$. The number of rules generated in Algorithm 1 is $|\Delta_{\mathcal{P}'_r}| = n_{sync} + \frac{|\delta \times \Delta| - n_{sync}}{3} + n_{SR} \times |\delta| = \frac{2}{3} n_{sync} + \frac{|\delta \times \Delta|}{3} + n_{SR} \times |\delta|$. The size of transition rules reduced is $|\Delta'| - |\Delta_{\mathcal{P}'_r}| = \frac{2}{3}|\delta \times \Delta| - \frac{2}{3} n_{sync} - n_{SR} \times |\delta|$. We can infer from this expression that the fewer places that $\mathcal{B}$ and $\mathcal{P}$ transition synchronously the more transition rules Algorithm 1 can reduce.

Discussions. Algorithm 1 makes a product of the transition relations respectively from the BA and LPDS, where all the transition rules are explored. Obviously, this process could be inefficient if the BA and LPDS are represented in a flattened approach, since the sizes of the transition relations can be exponentially large. Symbolic representations are efficient to model transition relations; therefore the cost of Algorithm 1 can be exponentially smaller on symbolic representations than that on flattened representations. However, the symbolic rules should be properly separated for the reduction to be effective. For example, if there is only one giant symbolic transition rule for each transition relation, Algorithm 1 will have no reduction effect. Symbolic rules are commonly differentiated by their control locations. This explains why the idea in Figure 2(b) is used instead of that in Figure 2(c), because LPDS usually has a better control-flow structure than BA.

5 Implementation

We apply the BPDS model in the verification of Windows device drivers with their formal hardware interface models as illustrated in Figure 3, where software is represented as LPDS and hardware is represented as BA. From the view of software, we

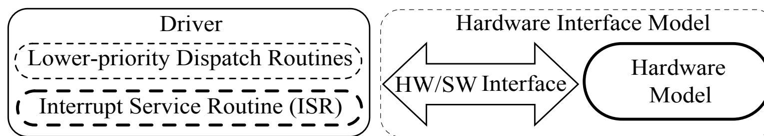

Fig. 3. Driver-centric co-verification.

specify both the HW/SW interface and the hardware model, which together we refer to as a hardware interface model. The HW/SW interface describes how hardware and software should transition synchronously when they interact through their interfaces. The hardware model describes the desired hardware behaviors when hardware and software transition asynchronously, i.e. when there is no HW/SW interaction.

First, we present several preliminary definitions for our implementation. Second, we elaborate on the specification of the HW/SW interface and the hardware model respectively by examples. Third, we illustrate our automatic co-verification tool, CoVer.

5.1 Preliminary Definitions

We use Transaction Level Modeling (TLM) to specify the hardware interface model. TLM is a commonly used approach to hardware system-level specification, and we have designed a specification language, modelC, for our TLM specification. The modelC language uses C semantics with two extensions to support non-determinism and relative atomicity (see definitions below). In modelC, (1) we treat numbers as bounded integers, so hardware registers can be properly modeled; and (2) the global hardware state space is static, i.e. there is no dynamic memory allocation.

Hardware transaction. In co-verification, the interaction between hardware and software is relevant rather than the implementation details of a hardware device; therefore it is unnecessary to preserve the clock-driven semantic feature. We define a hardware transaction to represent a hardware state transition in an arbitrarily long but finite sequence of clock cycles. Hardware transactions are atomic to software. The concept of hardware transaction preserves hardware design logic that is visible to software, but hides details only necessary for synthesizable Register Transfer Level (RTL) design.

Hardware transaction function. We define a transaction function as a C function that describes a set of hardware transactions (i.e. state transitions). Because transactions are atomic, the intermediate states of hardware during a transaction is not visible to software. We define the current-states and next-states of a transaction function respectively as $\rho \subseteq Q$ representing the hardware states when entering the function and $\rho' \subseteq Q$ representing the hardware states when exiting the function. Formally, a transaction function $\mathcal{F} : Q \times Q$ describes a set of state transitions. Following this definition, any terminating C function can be treated as a transaction function.

Relative atomicity. Relative atomicity has two key ideas: (1) hardware transactions are atomic from the view of software; and (2) Interrupt Service Routines (ISRs) are atomic to other lower-priority software routines. In device/driver applications, when hardware fires an interrupt, the Operating System (OS) calls the ISRs that are registered in the interrupt vector table sequentially until an ISR acknowledges its ownership of the interrupt. During this process, only one ISR can run at a time and other hardware interrupts are suppressed [12]. The interrupted thread can continue its execution only after the interrupting ISR terminates.

Software synchronization points. As the concrete counterpart of the $SensitiveSet$ concept, we define software synchronization points as a set of program locations[1] where the program statements right before these locations may be dependent with some of the hardware state transitions. In general, there are three types of software synchronization points: (1) the point where the program is initialized; (2) those points right after software reads/writes hardware interface registers; and (3) those points where hardware interrupts may affect the verification results. The first and second types are straightforward for hardware and software to transition synchronously. We may understand the third type in such a way that the effect of interrupts (by executing ISRs) may influence certain program statements, e.g. the statements that access global variables.

[1] Assuming the program is preprocessed to ensure that every statement is atomic from the view of hardware.

5.2 Specifying Hardware Interface Model

In the specification of the hardware BA model, $\mathcal{B} = (\Sigma, Q, \delta, q_0, F)$, the alphabet Σ is the power set of the set of propositions induced by software interface events (see definition below); the set of states Q is defined by global variables; the initial state q_0 is given by an initialization function; and the transition relation $R = R_{evt} \cup R_{model}$ has two parts: R_{evt}, is a set of transitions that are dependent with at least one of the software LPDS transition rules; R_{model}, is a set of transitions that are not dependent with any of the LPDS transition rules. Informally, R_{evt} is described by the HW/SW interface and R_{model} is described by the hardware model. In this paper, we are interested only in safety properties; therefore the Büchi constraint F is not necessary to be specified.

Specifying the HW/SW interface. The HW/SW interface, as the abstraction of the HW/SW layers between the target device and driver, propagates the hardware (resp. software) interface events to software (resp. hardware).

Figure 4 illustrates an example of a software interface event function in response to a register write operation. The keyword `__atomic` indicates that `WritePortA` is a transaction function atomic from the view of software. This transaction function describes a set of state transitions, $R'_{evt} \subseteq R_{evt}$, when the driver writes to the interface register, PortA, of the Sealevel PIO-24 digital I/O device (see Section 6). Figure 5 illustrates an example how function calls to a software write-register function (originally provided by the OS) are related to interface event functions. A software interface event happens when the entry stack symbol of the interface event function is reached.

```
__atomic VOID WritePortA(UCHAR ucRegData) {
  // If Port A is configured as an "input" port
  if ( g_DIORegs.CW.CWD4 == 1 ) {
    // Write to the output register instead of the port
    g_DIOState.OutputRegA.ucValue = ucRegData;

  } else { // Otherwise, configured as an "output" port
    // Update both the port and the output register
    g_DIORegs.A.ucValue = ucRegData;
    g_DIOState.OutputRegA.ucValue = ucRegData;
  }
}
```

Fig. 4. An implementation of a software interface event.

```
VOID WRITE_REGISTER_UCHAR
    (PUCHAR Register, UCHAR ucData) {
  switch ( Register ) {
    case REG_PORTA:  WritePortA(ucData); return;
    case REG_PORTB:  WritePortB(ucData); return;
    . . .
    case REG_CONFIG:  WriteConfig(ucData); return;
    case REG_STATUS:  WriteStatus(ucData); return;
    default: abort "Register address error."; return;
  }
}
```

Fig. 5. Relating register calls to software interface events.

When hardware fires an interrupt, the ISR should be invoked to service this interrupt. The HW/SW interface simulates this process as shown in Figure 6. The variable `IsrRunning` represents the software status and the variable `InterruptPending` represents the hardware status. The function `RunIsr` has three parts, (1) check/prepare the precondition before invoking the ISR; (2) invoke the ISR; and (3) set both the hardware and software to proper status after ISR. The first and third parts describe synchronous state transitions of both hardware and software. Formally, when hardware (the BA) fires an interrupt, i.e. the interrupt pending status is set to be true, the corresponding state transitions in software (the LPDS) will be enabled, so the BA and the LPDS will transition synchronously in the next state transition.

```
VOID RunIsr() {
  __atomic {
    // Make sure only one ISR is invoked
    if ( (IsrRunning == TRUE) ||
         (InterruptPending == FALSE) )
      return;
    IsrRunning = TRUE;
  }

  // Invoke the ISR
  IsrFoo();

  __atomic {
    IsrRunning = FALSE;
    InterruptPending = FALSE;
  }
}
```

Fig. 6. Interrupt monitoring function.

```
__atomic VOID Run_DIO() {

  // non-deterministic choices
  switch ( choice() ) {

    // Port I/O Management
    case 0: RunPorts(); break;

    // Interrupt Management
    case 1: RunInterrupt(); break;
    ...
}
```

Fig. 7. The transaction function of the Sealevel PIO-24 card.

```
VOID HWInstr() {

  // non-deterministic choices
  while( choice() ) {

    // Run hardware transaction
    Run_DIO();

    // If interrupt has been fired
    RunIsr();
  }
}
```

Fig. 8. The hardware instrumentation function.

Specifying the hardware model. The hardware model describes the desired hardware behaviors when hardware works asynchronously with software to realize system functionalities. Conceptually, the behavior of the hardware model is represented as a set of state transitions, R_{model}, where all the transitions are labeled by a set of propositions that hold when no software interface event happens. Figure 7 illustrates an example of a transaction function, `Run_DIO`, that specifies the set of state transitions, R_{model}, for the digital I/O device. When `Run_DIO` is executed multiple times, the stub-functions such as `RunPorts` and `RunInterrupt` are non-deterministically invoked to simulate the concurrent sub-modules of the hardware device.

Hardware instrumentation function. We define a C function to invoke independent hardware transaction functions (for the hardware model) and ISRs. Figure 8 illustrates such an example, where `RunIsr` is invoked right after every hardware transaction, `Run_DIO`. If an interrupt is fired due to a hardware state transition by executing `Run_DIO`, the context-switch to the ISR is modeled as a function call, where the execution privilege switches back to the interrupted thread only after the ISR returns. This approach is correct to simulate the context-switches because ISRs are relatively atomic to other driver routines. The non-deterministic while-loop simulates the delays of either software or hardware. This is correct when only safety properties are verified.

5.3 Co-verification Tool, CoVer

Our co-verification tool, CoVer, has two automatic steps. First, the frontend instruments (i.e. make the product of) the driver with the hardware interface model to generate a C program, which conceptually is the reduced verification model $\mathcal{P}'_r$ discussed in Section 4.2. Second, the SLAM engine checks the reachability property (in the form of a SLIC rule [4]) of the C program.

The instrumentation step has two parts. First, the dependent HW/SW transitions when driver writes hardware registers are modeled by replacing the implementation of the driver programming interfaces (see Figure 5), which is provided in the harness of

Static Driver Verifier [4]. Second, CoVer inserts function calls to the hardware instrumentation function `HWInstr` into the C code of the driver, between the driver statements. Without reductions, the function calls need to be inserted after every driver statement. Using our reduction algorithm, CoVer first detects the software synchronization points in the driver code and then inserts the function calls only at those detected points. Conceptually, the instrumentation lets hardware run continuously for all the possibilities after every HW/SW synchronous transition. Compared to the trivial approach that inserts `HWInstr` after every software statement to simulate the HW/SW concurrent state transitions, our approach can significantly reduce the complexity of the verification model, because the number of software synchronization points are usually very small in common applications.

6 Evaluation

We have applied our approach to the verification of two fully functional Windows device drivers: (1) the Sealevel PCI (Peripheral Component Interconnect) PIO-24 Digital I/O card driver from Open Systems Resources (OSR), and (2) the Intel 82557/82558 based PCI Ethernet adapter driver from Microsoft Windows Driver Kit (WDK) samples. We developed hardware interface models respectively for the drivers and verified two kinds of properties: (1) whether a driver callback function[2] accesses the hardware interface registers in correct ways, e.g. a command should not be issued when the hardware is busy; and (2) whether a driver callback function can cause an out-of-synchronization between the driver and the device. In other words, we check if the return value of a driver callback function correctly indicates the current hardware state. Because both of the drivers have been provided to public as samples for years, we did not expect to find many bugs. However, CoVer detected seven real bugs. All these bugs can cause malfunction of the devices/drivers, where the symptoms include data loss, interrupt storm, device hang, etc. Our evaluation runs on a Lenovo ThinkPad notebook with Dual Core 2.66GHz CPU and 4GB memory. We set the timeout and spaceout threshold as 3000 seconds and 2000MB respectively.

Table 1 presents the statistics on the verification of the PIO-24 driver with its hardware interface model. CoVer detected four bugs and proved two properties of the driver. For example, the driver has two global variables to maintain the I/O request status and the device I/O port status respectively. The values of the two variables become inconsistent when the ISR interrupts the callback function `EvtDeviceControl` at a specific program location. This inconsistency will cause the driver to return invalid data to user applications later, which violates the rule `InvalidRead`. Another serious bug (detected by the rule `ProperISR1`) of this driver can cause interrupt storm. The design of the device expects interrupts being repeatedly generated in certain configuration, however the driver does not handle the interrupts correctly which will cause interrupts being fired more frequently than that can be consumed, i.e. interrupt storm. As a comparison, the Ethernet adapter driver disables the interrupt first and re-enables it after the interrupt processing is completed later in DPC (Deferred Procedure Call). This prevents the situation when interrupts overwhelm the PCI bus.

[2] Windows OS invokes the predefined driver callback functions to service the I/O requests from user applications.

Table 1. Statistics on the co-verification of the Sealevel PIO-24 device/driver.

Size of the driver (# of lines)							1724
Size of the hardware interface model (# of lines)							1232
Rule	Description	No Reduction		Reduction		Result	
		Time (Sec)	Mem. (MB)	Time (Sec)	Mem. (MB)		
DevD0Entry	Driver and device will not go out-of-synchronization when starting.	391.3	293	214.3	181	Passed	
DevD0Exit	Driver and device will not go out-of-synchronization when stopping.	71.1	69	38.4	43	Passed	
IsrCallDpc	ISR will not queue DPC without reading specific hardware registers.	Timeout	N/A	700.5	218	Failed	
InvalidRead	Driver will not read any invalid input data.	589.4	132	91.3	66	Failed	
ProperISR1	ISR will clear the device interrupt-pending status before return.	58.9	58	35.2	43	Failed	
ProperISR2	ISR will not acknowledge the interrupt fired by other devices.	74.1	62	28.7	37	Failed	

Table 2 presents the statistics on the verification of the Intel 82557/82558 based PCI Ethernet adapter driver with its hardware interface model. CoVer detected three bugs and proved five properties of the driver. For example, CoVer detects a bug that violates the rule `DevD0Entry` and reports an error trace where the callback function `EvtDeviceD0Entry` returns `TRUE` even if the driver fails to initialize the device correctly. This is a direct violation of Windows device driver programming standards and will cause the device unusable without the OS being notified. The error trace also illustrates that the driver continues its attempts to initialize the device even after the previous device operations have failed. This may cause the device permanently unaccessible. Compared to the PIO-24 device/driver, the Ethernet adapter device/driver have more comprehensive functionalities and implementation, where the static partial order reduction is clearly necessary for most of the rules to be even verified.

Table 2. Statistics on the co-verification of the Intel PCI Ethernet adapter device/driver.

Size of the driver (# of lines)							14406
Size of the hardware interface model (# of lines)							3518
Rule	Description	No Reduction		Reduction		Result	
		Time (Sec)	Mem. (MB)	Time (Sec)	Mem. (MB)		
DevD0Entry	Driver and device will not go out-of-synchronization when starting.	1328.3	758	367.1	182	Failed	
DevD0Exit	Driver and device will not go out-of-synchronization when stopping.	Timeout	N/A	206.6	143	Failed	
IsrCallDpc	ISR will not queue DPC without reading specific hardware registers.	64.1	99	39.9	79	Passed	
ProperISR1	ISR will clear the device interrupt-pending status before return.	48.9	59	32.6	52	Passed	
ProperISR2	ISR will not acknowledge the interrupt fired by other devices.	779.3	291	407.4	199	Passed	
DoubleCUC	Driver will not issue a command while the command unit is busy.	Timeout	N/A	602.4	238	Failed	
DoubleRUC	Driver will not issue a command while the receiving unit is busy.	N/A	Spaceout	1797.3	231	Passed	
ProperReset	Driver uses a correct sequence to reset the device.	Timeout	N/A	86.9	71	Passed	

7 Conclusion and Future Work

We have presented an efficient approach to reachability analysis of BPDS models for HW/SW co-verification. The key idea of this approach is to reduce unnecessary state transition orders between hardware and software, so there are fewer possibilities to be explored in verification. We have implemented this approach in our co-verification tool,

CoVer, and successfully applied it to co-verify two Windows device drivers with their device models. CoVer proved seven properties and detected seven previously undiscovered software bugs which can cause serious system failures. Evaluation shows that the reduction can significantly scale co-verification. These results demonstrate that our approach is very promising in ensuring the correct interactions between hardware and software. For the next step, we plan to apply our approach to more devices and drivers.

Acknowledgement. This research received financial support from National Science Foundation of the United States (Grant #: 0916968). We thank Con McGarvey, Onur Ozyer, and Peter Wieland for evaluating our findings of device driver bugs.

References

1. Li, J., Xie, F., Ball, T., Levin, V., McGarvey, C.: An automata-theoretic approach to hardware/software co-verification. In: Proc. of FASE. (2010)
2. Schwoon, S.: Model-Checking Pushdown Systems. PhD thesis (2002)
3. Kurshan, R.P., Levin, V., Minea, M., Peled, D., Yenigün, H.: Static partial order reduction. In: Proc. of TACAS. (1998)
4. 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. (2006)
5. Kurshan, R.P., Levin, V., Minea, M., Peled, D., Yenigün, H.: Combining software and hardware verification techniques. FMSD (2002)
6. Xie, F., Yang, G., Song, X.: Component-based hardware/software co-verification for building trustworthy embedded systems. JSS **80**(5) (2007)
7. Monniaux, D.: Verification of device drivers and intelligent controllers: a case study. In: Proc. of EMSOFT. (2007)
8. Ramalingam, G.: Context-sensitive synchronization-sensitive analysis is undecidable. ACM Trans. Program. Lang. Syst. (2000)
9. Bouajjani, A., Esparza, J., Maler, O.: Reachability analysis of pushdown automata: Application to model-checking. In: Proc. of CONCUR. (1997)
10. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem. PhD thesis (1994)
11. Kurshan, R.P.: Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach. Princeton University Press (1994)
12. Microsoft: Synchronizing interrupt code. In: MSDN: msdn.microsoft.com/en-us/library/aa490313.aspx. (2009)