Portland State University
PDXScholar

Dissertations and Theses

Dissertations and Theses

1-1-2010

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

Juncao Li
Portland State University

Let us know how access to this document benefits you.

Follow this and additional works at: https://pdxscholar.library.pdx.edu/open_access_etds

Recommended Citation

Li, Juncao, "An Automata-Theoretic Approach to Hardware/Software Co-verification" (2010). *Dissertations and Theses.* Paper 12.

10.15760/etd.12

This Dissertation is brought to you for free and open access. It has been accepted for inclusion in Dissertations and Theses by an authorized administrator of PDXScholar. For more information, please contact pdxscholar@pdx.edu.

An Automata-Theoretic Approach to

Hardware/Software Co-verification

by

Juncao Li

A dissertation submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
in
Computer Science

Dissertation Committee:
Fei Xie, Chair
Thomas Ball
Jingke Li
Suresh Singh
Bryant W. York
Fu Li

Portland State University

© 2010

ABSTRACT

Hardware/Software (HW/SW) interfaces are pervasive in computer systems. However, many HW/SW interface implementations are unreliable due to their intrinsically complicated nature. In industrial settings, there are three major challenges to improving reliability. First, as there is no systematic framework for HW/SW interface specifications, interface protocols cannot be precisely conveyed to engineers. Second, as there is no unifying formal model for representing the implementation semantics of HW/SW interfaces accurately, some critical properties cannot be formally verified on HW/SW interface implementations. Finally, few automatic tools exist to help engineers in HW/SW interface development.

In this dissertation, we present an automata-theoretic approach to HW/SW co-verification that addresses these challenges. We designed a co-specification framework to formally specify HW/SW interface protocols; we synthesized a hybrid Büchi Automaton Pushdown System, namely Büchi Pushdown System (BPDS), as the unifying formal model for HW/SW interfaces; and we created a co-verification tool, CoVer that implements our model checking algorithms and realizes our reduction algorithms for BPDS.

The application of our approach to the Windows device/driver framework has resulted in the detection of fifteen specification issues. Furthermore, utilizing CoVer, we discovered twelve real bugs in five drivers. These non-trivial findings have demonstrated the significance of our approach in industrial applications.

DEDICATION

To the memory of my father, Bochun Li

To my mother, Jinping Cao

To my wife, Xiaojing Liu

ACKNOWLEDGMENTS

This dissertation could not have been accomplished without the help and influence by many generous people. I am sincerely grateful and deeply in debt to them.

First and foremost, thanks to my advisor, Prof. Fei Xie, who brought me on board to software engineering and formal methods. When I first met Fei, various wild ideas jumped out of my head, but I was never able to find the right track to approach the real problems. Fei always listened to my ideas with a patient smile and then pointed out the problems. While Ph.D. study is a long trip with enormous possible outcomes, I often plan for the worst. Fei has always encouraged me and cheered me up when I was frustrated. Fei taught me how to be a student, a researcher, and an educator.

Dr. Thomas Ball and Dr. Vladimir Levin were very generous to share their visions and ideas with me. The key idea of this research comes from a discussion with them. They spent lots of time and effort in helping me with this research. Every discussion with them was fruitful with ideas. They also helped me edit my papers and critiqued my talks. Vladimir hosted me during my two internships at Microsoft. He also helped me in writing the very first prototype of CoVer in order to deal with bitunion operations of SLAM.

Prof. Bryant W. York helped me in many different ways, from computer science to life philosophy. He has broad knowledge and always is ready to help me. He taught me the idea of cognitive science, so that I can understand how knowledge is acquired by people and, most importantly, by myself. He showed me how science

in very different disciplines can be combined to serve each other. He taught me how to write technical papers and gave me feedback on my talks.

Con McGarvey was my mentor during my two internships at Microsoft. Con taught me various skills for working in Microsoft and discussed my research. For a rookie like me, he had to be very patient and responsible. There are many other friends from Microsoft who also gave me great help. They are Randy Aull, Jaivir Aithal, Albert Chen, Alessandro Forin, Nar Ganapathy, David Hargrove, Rahul Kumar, Shuvendu Lahiri, Jakob Lichtenberg, Arvind Murching, Onur Ozyer, Shaz Qadeer, Peter Shier, Peter Wieland, Eliyas Yakub, and Yue Zuo. Among those merits that I learned from them was their passion to make better software. This has inspired me during my dissertation work and is what wakes me up every morning and excites me every day.

My other dissertation committee members, Prof. Fu Li, Prof. Jingke Li, and Prof. Suresh Singh, made invaluable contributions to my dissertation. They offered their perspectives on my research and gave careful feedback on my dissertation.

While I did not study all by my own, it is always my pleasure to take classes from PSU faculties, such as Prof. Sergio Antoy's Programming Language, Prof. Andrew Black's Scholarship Skills, and Prof. David Maier's Algorithm Design and Analysis. I can hardly remember the last time when I blinked my eyes in Prof. Ivan Sutherland's lectures. They are too interesting for me to miss any.

Thanks to my fellow graduate students, Yan Chen, Tim Chevalier, Thanh Dang, Tom Harke, Chuan-kai Lin, Emerson Murphy-Hill, Nicholas T. Pilkington, Xiuli Sun, and Candy Yiu. Interactions with them have enriched my Ph.D. study and sometimes made my life much easier. For example, this dissertation's LaTeX template would have costed me tons of time if it was not for Chuan-kai's work.

Four years of Ph.D. study may be joyful to me; however, it cannot be pleasant to my wife, Xiaojing Liu. Thanks to her love and understanding, which have made me happy every day. At last, certainly, not least, thanks to my parents, Bochun Li and Jinping Cao. Their persistent and passionate characters have been the role model for me. Their early education of me has made me most who I am today.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

1.1 MOTIVATION AND PROBLEM STATEMENT

1.1.1 Motivation

Computer systems such as Personal Computers (PCs) and embedded systems are pervasive. Our everyday life depends on these systems, e.g., accessing data from a local disk or via the Internet using our personal computers and driving to work in our cars equipped with tens and even hundreds of embedded processors. Such dependencies demand high-confidence in these systems. We would be greatly annoyed by blue screens from Microsoft Windows while working on important documents and endangered by malfunctions of the embedded controllers in our cars' braking systems. Many such failures come from HW/SW (Hardware/Software) interface problems that are often hard to test and to debug. High-confidence is traditionally achieved by extensive testing which is becoming cost-prohibiting and, therefore, increasingly supplemented by formal verification such as model checking [21, 74].

HW/SW interfaces are pervasive in modern computer systems. For example, device drivers [19] that operate hardware, constitute about 70% of Linux kernel code (version 2.4.1) and for Windows XP there are over 35,000 device drivers with over 100,000 versions available for various hardware devices [65]. Because drivers usually work in the kernel mode of Operating Systems (OSs), their failures can have severe consequences. Kernel mode drivers cause 85% of reported failures in

Windows XP [83] and there are seven times more failures in drivers than in the rest of the Linux kernel [19]. There have been many efforts [4, 14] to formally verify software properties on drivers without considering the behaviors of hardware devices. A successful example is Microsoft's Static Driver Verifier (SDV) [4], which is a tool for Windows driver verification based on the SLAM model checking engine [9]. Nevertheless, one of the fundamental reasons for computer system failures has been overlooked for years. According to Microsoft's Online Crash Analysis (OCA) [81], at least 52.6% of Windows crashes are related to the interactions between device drivers (software) and their devices (hardware)[1], not to mention those failures that cannot be gathered, e.g., a USB (Universal Serial Bus) mouse cannot be detected after a PC system awakes from sleep. These kinds of system failures are commonly related to HW/SW interface interactions.

1.1.2 Problem Statement

Our thesis is that co-verification of HW/SW interface protocols can be effectively achieved via formal specification and model checking. There are three major challenges:

- *Lack of effective formal specification framework.* Hardware and software are often manufactured separately because their construction requires highly different expertise. Therefore, specifications are necessary to describe HW/SW interface protocols. Such specifications should be self-explanatory and free of ambiguities, since any specification mistake may be encoded in implementations and cause serious failures. In industrial settings, English is commonly used to specify HW/SW interface protocols. The misinterpretations and implementation deviations due to ambiguous or inaccurate English specifications have been a long-outstanding hazard to system reliability.

[1] Not all drivers interact with devices. For example, an antivirus driver intercepts and analyzes I/O between other drivers and devices.

- *Lack of a unifying formal model for HW/SW interfaces.* Hardware and software have different implementation semantics and different formal representations. Hardware designs are finite state and often modeled as some kind of finite state machines such as ω-automata or Büchi automata (BA) [43]. Software designs are infinite state and often modeled as some kind of Pushdown Systems (PDSs) [77]. However, for HW/SW interfaces, it is not desired to model both hardware and software as either pushdown systems or finite state machines (see Section 1.3).

- *Lack of verification tool support.* It is highly desired that both the design and implementation of HW/SW interfaces are supported by automatic verification tools, so that critical properties can be analyzed in a faithful and systematic manner. There are few tools existing for such a purpose and many lack practical performance. Among the various challenges to co-verification tool development, the key is how to efficiently exploit the nature of HW/SW interactions in co-verification, so that the tools can scale up to a practical level of complexity.

Other than these major challenges, there is also a lack of understanding about HW/SW interface failures. Although hardware and software engineers are well aware of the intrinsic complications of HW/SW interface implementations, all have experienced significant difficulties in pinpointing the root causes of relevant failures. One major reason is that hardware implementations are usually unaccessible to software engineers, and vice versa. Therefore, it is impossible for software engineers to look into the states of a hardware device when the driver fails together with the device during runtime testing; and, at the same time, it also is hard for those hardware engineers of the device to look into the driver's state.

1.1.3 The Device/Driver Scenario

In computer systems, HW/SW interfaces are often implemented in devices and drivers. A *device/driver framework* refers to a type of HW/SW interface as well as the devices and drivers that both utilize this interface. We observed a common development process for device/driver frameworks in industrial settings as illustrated in Figure 1.1. The process contains three stages: First, the design stage, where a group of hardware and software companies design the HW/SW interface protocol of a device/driver framework together. The HW/SW interface specification is commonly written in structured English. Second, the development stage, where the English specification is published so that different companies can produced devices and drivers that are compliant with the HW/SW interface protocol. Finally, the post-release stage, where devices and drivers are tested for their conformance to the HW/SW interface protocol. (More discussion is presented in Section 3.3.)

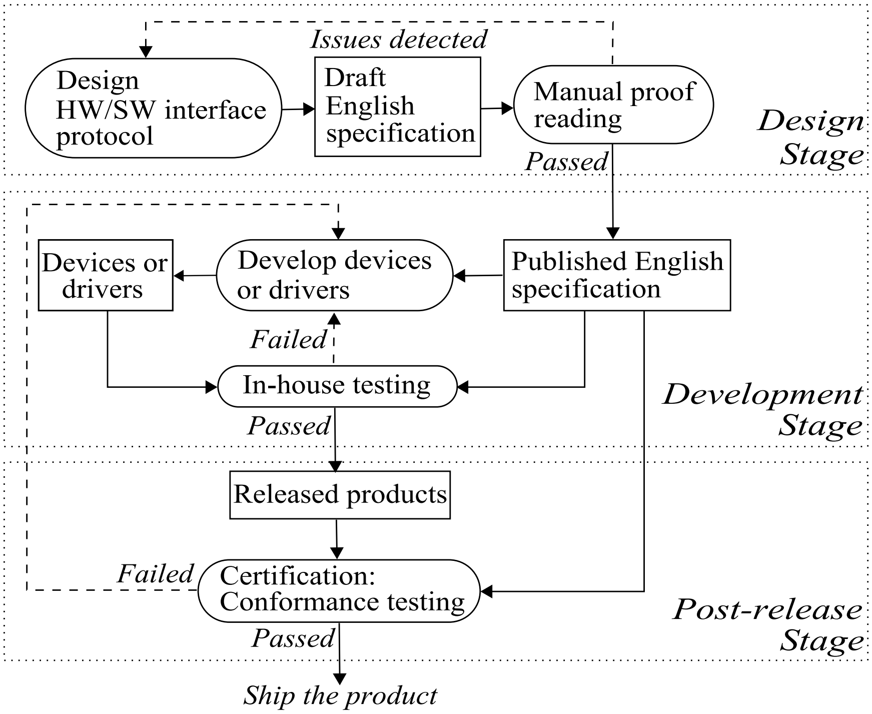


Figure 1.1: Development process of device/driver frameworks.

As we have discussed in the previous sub-section, this development process suffers from three problems: First, since English lacks a formal semantics, we cannot guarantee a unique interpretation from the same English specification. Second, there is not a unifying formal model used to represent the device/driver interactions. Third, verification tools cannot be used to validate the English specifications and are little used to validate the device/driver implementations against the HW/SW interface protocol. It is also very hard for engineers to debug their implementations for an issue related to device/driver interactions.

1.2 CONTRIBUTIONS

1.2.1 Our Approach

We present an automata-theoretic approach to HW/SW co-verification, verifying hardware and software together. The main components of our approach are illustrated in Figure 1.2. In co-specification, we formally specify the HW/SW interface protocol, where the result of the specification can be represented by a unified formal model of HW/SW interface. Such a formal model then can be analyzed automatically using model checking algorithms for correctness assurance. In order to realize our approach, we make the following five contributions in this dissertation:

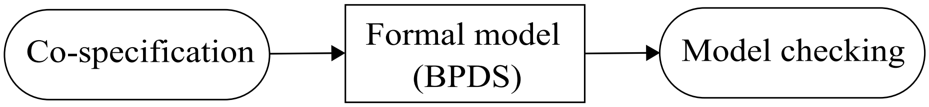


Figure 1.2: Main components of our co-verification approach.

Co-specification. We design a co-specification framework to specify HW/SW interface protocols, where the specification captures the asynchronous hardware behaviors, the asynchronous software behaviors, and the interactions between hardware and software. In our approach, the differences between hardware and software

are not only considered but also exploited. For example, we utilize the concept of Transaction Level Modeling (TLM) [66] to specify hardware; we use a restricted C semantics to specify software. Our specification language, modelC, utilizes the C semantics with three restrictions to achieve a finite state representation and two extensions to support the characteristics of HW/SW interface specifications.

The modelC language has precise semantics; a hardware specification in modelC can be represented as a BA [43]; a software specification in modelC can be represented as a PDS [77]; and the specification of HW/SW interactions in modelC can be represented as the synchronization of the BA and PDS. Therefore, formal models constructed by co-specification can be utilized in the development process of hardware and software, as the formal HW/SW interface specifications. Furthermore, they can also serve as the test harnesses for co-verification, co-simulation, conformance testing, etc. For example, in co-verification, a hardware model constructed by co-specification can be used as the harness for verifying a software implementation.

Co-verification model. We synthesize a hybrid Büchi Automaton Pushdown System (BPDS) as a unifying formal model for HW/SW co-verification. The insight is to synchronize a BA that represents hardware and a PDS (actually a labeled pushdown system as presented in Chapter 4) that represents software. The BPDS closely models the implementation semantics of both hardware and software. For example, BA have been commonly used to model hardware designs in verification practices; the unbounded stack of PDS can represent recursion in software programs. Generally speaking, a BPDS model is a concurrent system with a synchronous execution mode [22], i.e., both the BA and the PDS must transition at the same time in order to make one BPDS transition. In synchronous execution mode, it is straightforward to model the situation when hardware and software transition simultaneously. However, they may also step asynchronously, which can be modeled by introducing self-loop transitions to both BA and PDS.

Co-verification algorithms and optimizations. We design verification algorithms of BPDS models for reachability properties and Linear Temporal Logic (LTL) [71] properties respectively. With respect to reachability analysis, we demonstrate that a BPDS model can be converted into a PDS model; therefore, existing model checkers for PDS can be readily utilized in co-verification. For LTL checking, we employ an automata-theoretic approach. An LTL formula is first negated and then represented as a BA. The BA is combined with BPDS in such a way that the BA monitors the state transitions of the BPDS. As the last step, we only need to compute whether the BA has an accepting run on the BPDS.

In a naïve approach, verification of BPDS needs to explore all interleavings of the concurrent execution between BA and PDS, where some of the interleavings may be unnecessary to explore. We prove that some special interleavings between the BA and PDS are enough to preserve the properties to be checked. We base our reduction algorithms on the concept of static partial order reduction [44], a paradigm of partial order reduction [33, 69] that reduces unnecessary state transitions during the compilation phase instead of the model checking phase. Such reduction algorithms are very useful in practice, since they do not require any modification to the model checker. Therefore, model checkers with industry strength can be readily utilized in our approach.

Co-verification tool. We have created a co-verification tool, CoVer, that supports both reachability analysis and LTL model checking of BPDS. For reachability analysis, CoVer is implemented based on the SLAM verification engine [4]. It accepts HW/SW designs or implementations specified in C/modelC languages. For LTL checking, CoVer is implemented based on the Moped model checker [77]. It accepts HW/SW designs specified in Boolean programs [7]. For every property violation detected, CoVer provides an execution trace of both hardware and software.

This feature is a significant help to hardware and software engineers in exploring, understanding, and validating HW/SW interface designs and implementations.

Evaluation. We use Windows devices/drivers for the case studies of our research. We have applied our approach to four device/driver interfaces, such as the Intel 8255x 10/100Mbps Ethernet controller device/driver interface and the USB 2.0 device/driver interface. All the HW/SW interfaces are industry standards presented in English specifications. Our co-specification process of HW/SW interfaces led to the detection of fifteen issues in the English specifications. Such specification issues can mislead development engineers and cause product failures. Given the fact that some of the English specifications have existed for many years and been revised several times, our approach is rather effective of discovering these issues.

Co-verification is evaluated in reachability analysis and LTL checking respectively. For reachability analysis, we have applied CoVer to five Windows driver implementations, using the formal device models constructed by co-specification. Some of the drivers are fully functional, well tested, and provided as sample drivers in Microsoft Windows Driver Kit (WDK) [59, 61, 63] for many years. Utilizing CoVer, we have discovered real bugs in each of the drivers for a total bug count of twelve. All of these bugs, which could cause serious system failures including data loss, interrupt storm, device hang, etc., involve device/driver interactions and were previously unknown to the driver developers. More specifically, one bug happens when a driver does not initialize its device correctly, i.e., a default device state is not considered during the initialization process; three bugs happen when devices interrupt their drivers, e.g., one of the bugs may cause an interrupt storm; four bugs are due to the out-of-synchronization between devices and drivers, e.g., a driver issues a command while its device is busy; and four bugs happen when drivers mishandle their device failures, e.g., a driver returns SUCCESS when its device actually fails. For LTL checking, we designed a synthetic BPDS template to generate BPDS models with various complexities. Such template can mimic the common scenarios of HW/SW interactions. The co-verification statistics illustrate that our reduction algorithms are very effective in both reachability analysis and LTL checking. The average reduction of the verification cost is 70% in time usage and 30% in memory usage.

1.2.2 Device/Driver Development using Our Approach

Figure 1.3 illustrates how our approach can be integrated into the development process of device/driver frameworks. In the design stage, the HW/SW interface protocol is formally specified as a formal model using our co-specification framework. This formal model can be validated by automatic tools such as CoVer. In the

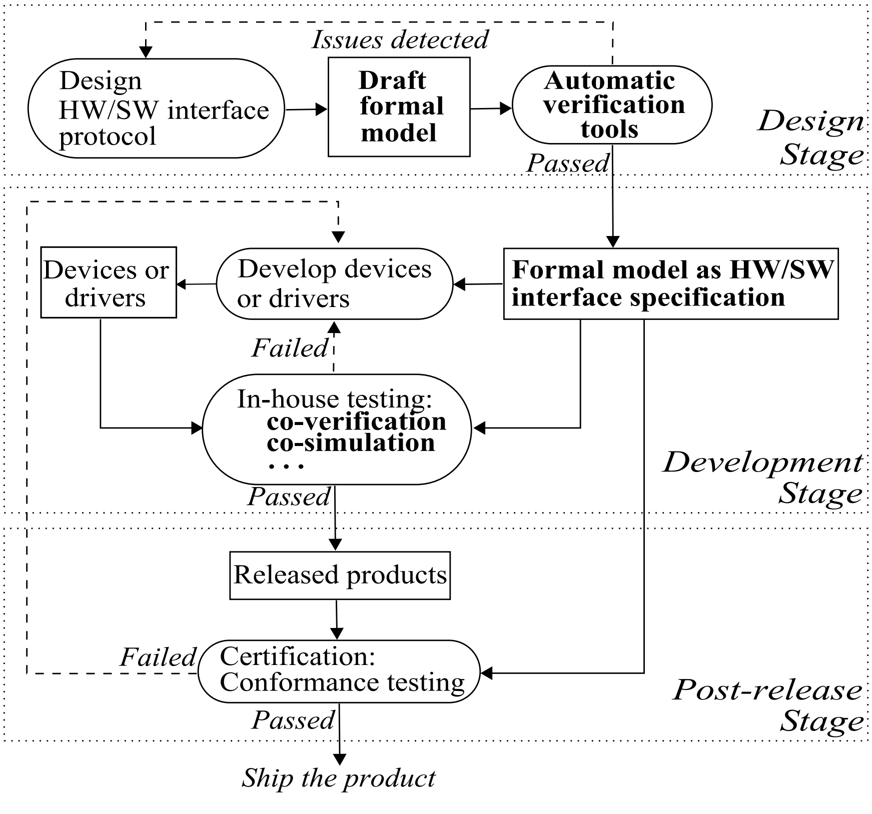

Issues detected
Design HW/SW interface protocol
Draft formal model
Automatic verification tools
Design Stage
Passed
Devices or drivers
Develop devices or drivers
Formal model as HW/SW interface specification
Failed
In-house testing: co-verification co-simulation . . .
Development Stage
Passed
Released products
Failed
Certification: Conformance testing
Post-release Stage
Passed
Ship the product

Figure 1.3: Development process of device/driver frameworks using our approach.

development stage, the formal model is published to guide the HW/SW interface implementations. Furthermore, tools can be utilized to validate the device/driver implementations using the formal model as a test harness. In conformance testing, the formal model can also serve as the golden model, where the device/driver implementations are tested according to the behavior of the formal model.

In this device/driver development process, our BPDS model serves as the formal representation of HW/SW interactions; our co-specification framework is utilized in the formal specification of HW/SW interface protocols; and our co-verification tool, CoVer, is applied to check the formal models and implementations.

1.3 RELATED WORK

Formal verification. Formal verification [41] uses rigorous mathematical reasoning to show that a design meets a property specification. In general, there are two approaches to formal verification: *theorem proving* and *model checking*. Theorem proving [28, 40] checks a property on a system design by proving a theorem in an underlying logic. Since most theorem provers require lots of manual effort, they may not scale to verifying HW/SW implementations. This dissertation research employs the other approach, model checking [21, 74], an automatic formal method that checks whether a model conforms to given properties. Usually, the model is generated automatically from a system design or implementation; the properties are specified manually to assert the desired behaviors of the system; and model checking is the process that explores the state space of the model to check whether a property can be violated.

When the target system is complex, model checking often faces a combinatorial explosion of the state space to be explored. To address this problem, techniques such as predicate abstraction [34] are often applied to reduce the complexity of models. These abstractions should be conservative so that all defects in the original system are preserved. However, due to over-approximation, a defect found in the model may not be a real defect in the system. Kurshan [43] proposed the idea of counterexample-guided abstraction and refinement, where the verification process starts with a highly abstracted model and then asymptotically introduces more details to the abstraction based on infeasible counterexamples given by the model checking engine. Clarke, et al. [20] applied this idea to symbolic model checking and demonstrated its effectiveness in hardware verification. Ball, et al. [6] utilized this idea in SLAM to verify C programs. We implement our co-verification tool, CoVer, based on the SLAM engine; therefore, counterexample-guided abstraction and refinement is also applied in CoVer.

The property specification languages usually are different for various application domains of model checking. In hardware model checking, *temporal logic* [71, 72] often is used. In a temporal logic, the usual operators of propositional logic are augmented with *temporal operators*, which are used to form assertions about changes over time. Depending on the temporal operators, there are different temporal logics such as LTL [71] and Computation Tree Logic (CTL) [21]. The Property Specification Language (PSL) [1], as the industry standard for hardware property specification, is an extension of LTL and CTL. In software model checking, properties are often specified in C-like languages such as SLIC (Specification Language for Interface Checking) [10] and BLAST query language [13], because preserving the source language constructs makes the specification more intuitive than temporal logic based specifications. Our co-verification framework accepts property specifications in either LTL or SLIC.

Formal specification of HW/SW interfaces. Various formal languages have been proposed for specifying the designs of embedded systems, e.g., Hybrid Automata [2], LOTOS [85], Co-design Finite State Machines (CFSMs) [3], and Petri-net based languages such as PRES [26]. Hybrid Automata and CFSMs have been directly model-checked. LOTOS and PRES have been verified via translation to directly model-checkable languages. Furthermore, there have been lots of research on formalizing interface semantics, such as I/O automata by Lynch, et al. [52] and interface automata by De Alfaro, et al. [27]. Kroening, et al. [42] have used SystemC [66] to specify HW/SW interface designs. However, none of the research formally models the stack, an important feature of software implementations. We model the software stack, so that our specification can closely resemble the implementation semantics of HW/SW interfaces. As a significant benefit, our formal specifications can be used, without any modification, as the test harness for software (respectively, hardware) implementations.

Li, et al. [47, 51] modeled component-based embedded systems using ω-automata and specified xPSL [88] (an extension of PSL to support the specification of temporal assertions over both hardware and software events) properties on component interfaces as one representation of HW/SW interface protocols. When the interface protocols are complex, the number of required xPSL assertions becomes quite large, which is inefficient for either engineers or verification engines to use.

Co-verification. Validation techniques for HW/SW interface designs (respectively, implementations) fall into two major categories: co-simulation and (formal) co-verification, which complement each other. Co-simulation is low-cost and efficient in detecting shallow bugs while co-verification provides exhaustive state coverage and is effective in detecting deep bugs.

Research on co-simulation [11, 12, 32, 36, 37, 68, 76, 79] led to industrial tools such as Mentor Graphics' Seamless [55] and Microsoft's Giano [30]. This research focuses on exploring the design boundary between hardware and software rather than the correct implementations of HW/SW interfaces. For example, one important mission of such kind of co-simulation is to decide whether a function unit is best implemented in hardware or software. This is different from our goal: the correctness assurance of HW/SW interface designs and implementations.

Microsoft developed the Device Simulation Framework (DSF) [80] to support the co-simulation of device drivers and their device models. The goal of DSF is to improve the test coverage, increase the test automation, and reduce the test cost of drivers. Using DSF, driver implementation issues can be discovered at an early stage of development even before real hardware devices are available. However, the device models used in DSF are developed in an ad-hoc manner, i.e., only common device functionalities and a small subset of device behaviors are modeled; therefore, the test coverage of DSF is limited. As we discuss in Chapter 8, our formal models from co-specification also can be used in co-simulation via some extensions to DSF.

Device Driver Tester (DDT) [46] is a symbolic simulation engine for testing closed-source binary device drivers against undesired behaviors, such as race conditions, memory errors, resource leaks, etc. Given a driver's binary code, DDT simulates its execution with symbolic hardware, a shallow hardware model that mimics simple device behaviors such as interrupts. In symbolic hardware, most design logic is abstracted away by non-determinism; therefore, false bugs may be reported due to the overapproximation of the hardware behaviors. When simulating the interactions between device and driver, DDT employs a reduction method that allows interrupts only after each kernel API call by the driver. Such reduction is quite ad-hoc, since no formal correctness justification was given. Our static partial reduction algorithms are quite similar to the reduction idea used by DDT and can be considered as a formal foundation for DDT's reduction method.

There has been less research on co-verification than co-simulation. Kurshan, et al. [45] presented a co-verification framework which models hardware and software designs using finite state machines. Xie, et al. [48, 89] extended this framework to hardware and software implementations and improved its scalability via component-based co-verification. 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 [64]. 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, a composition of the two PDSs to model the HW/SW concurrency is problematic, because it is known that, in general, verification of reachability properties on concurrent PDS with unbounded stacks is undecidable [75].

Groβe, et al. [35] applied Bounded Model Checking (BMC) to check whether assembly programs are correctly executed on a RISC CPU. Our approach is different from Groβe's approach in the sense that the properties are at a higher level of abstraction concerning the interactions between devices and drivers. Furthermore, the completeness of BMC is restricted by a predefined verification bound compared to a standard model checking approach.

Bouajjani, et al. [15] presented a procedure to compute backward reachability of PDS and applied this procedure to linear/branching-time property verification. This approach was improved by Schwoon [77] and implemented in Moped, a tool that checks LTL properties of PDS. An 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 whether the BA has an accepting run. The goal of this previous research was to verify software only; our goal is to co-verify hardware and software.

1.4 DISSERTATION OUTLINE

This dissertation is organized as follows. Chapter 2 introduces the background of our research. Chapter 3 presents our co-specification framework and discusses how to apply co-specification to the development process of hardware and software.

Chapter 4 elaborates on the Büchi pushdown system model for co-verification. Chapter 5 presents our co-verification algorithms. Chapter 6 discusses the implementation details of our co-verification approach. Chapter 7 presents the evaluation results. Chapter 8 concludes and discusses future work.

Chapter 2

BACKGROUND

A state transition system is an abstract machine used in the study of computation. The machine consists of a set of states and transitions between states, which may be labeled by symbols chosen from an alphabet; the same label may appear on more than one transition; both the set of states and the set of transitions are not necessarily finite, or even countable. If the alphabet is a singleton, the system is essentially unlabeled; therefore a simpler definition that omits the labels can be used. State transition systems usually have various forms in order to represent different systems. For example, finite state machines, such as Büchi automata and ω-automata [43], are common representations of hardware designs. Pushdown systems [15, 77], pushdown automata without input alphabets and acceptance conditions, are common representations of software programs.

Properties specify the desired behaviors that should be observed on a system. Temporal logics [71, 72] have been widely used in property specification for both hardware and software. For example, they can be used to define a semantics for programs in such a way that includes termination and pre-/post- conditions. However, temporal logics are not intuitive in software property specifications when the source language constructs are desired. Alternatively, Specification Language for Interface Checking (SLIC) [10] is a well known software property specification language designed for safety properties[1] in the SLAM project [4]. SLIC has a C-like syntax, is infinite state, and allows state variables to be read from or written

[1] Often stated as: “bad events never happen.”

to in property specifications.

Model checking [21, 74] is an automatic technique that verifies whether a state transition system meets a property specification. It was first introduced on finite state systems such as hardware designs, and then applied to infinite state systems such as software programs. In this dissertation, we utilize two software model checkers in the implementation of our co-verification algorithms: the SLAM verification engine [4] for C programs and the Moped model checker [77] for pushdown systems.

One major challenge to model checking is the state explosion problem, which is due, among other causes, to the modeling of concurrency by interleaving. However, exploring all interleaving of concurrent executions often is unnecessary. Such observations led to a technique called partial order reduction [33], which is applied on-the-fly during model checking in order to avoid exploring unnecessary interleavings. Kurshan, et al. [44] have demonstrated that partial order reduction can also be applied statically during the compilation phase before model checking; therefore, the reduction can be applied without any modification to the model checking algorithm.

Last, we will introduce the Windows device/driver stack as one of the application domains of our approach.

2.1 STATE TRANSITION SYSTEMS

2.1.1 Büchi Automaton

A *Büchi Automaton* (BA) $\mathcal{B}$, as defined in [43], 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 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. A path of $\mathcal{B}$ is a sequence of states, $q_1 \xrightarrow{\sigma_1} q_2 \xrightarrow{\sigma_1} \ldots q_i \xrightarrow{\sigma_i} \ldots$, where $q_i \in Q$, $\sigma_i \in \Sigma$, $i > 1$.

2.1.2 Pushdown System

A *Pushdown System* (PDS) $\mathcal{P}$, as defined in [77], is a tuple $(G, \Gamma, \Delta, \langle g_0, \omega_0 \rangle)$, where G is a finite set of global states (a.k.a., control locations), Γ is a finite stack alphabet, $\Delta \subseteq (G \times \Gamma) \times (G \times \Gamma^*)$ is a finite set of transition rules, and $\langle g_0, \omega_0 \rangle$ 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 $Conf(\mathcal{P})$. The head of a configuration $c = \langle g, \gamma v \rangle$ $(\gamma \in \Gamma, v \in \Gamma^*)$ is $\langle g, \gamma \rangle$ and denoted by $head(c)$. Similarly the head of a rule $r = \langle g, \gamma \rangle \hookrightarrow \langle g', \omega \rangle$ is $\langle g, \gamma \rangle$ and denoted by $head(r)$. The head of a configuration decides the transition rules that

Denotation	**Comment**
$g \in G$	Global state (a.k.a., control location)
$\gamma \in \Gamma$	Stack symbol
$\omega, v \in \Gamma^*$	Stack
$\langle g, \omega \rangle \in Conf(\mathcal{P})$	A configuration of $\mathcal{P}$
$\langle g, \gamma \rangle \hookrightarrow \langle g', \omega \rangle$	A PDS transition rule, i.e., $((g, \gamma), (g', \omega)) \in \Delta$
$\langle g, \gamma v \rangle \Rightarrow \langle g', \omega v \rangle$	A PDS state transition
$\langle g, \gamma v \rangle \Rightarrow^* \langle g', \omega v \rangle$	Forward reachability relation
$head(c)$, $head(r)$	Head of a PDS configuration/rule

Figure 2.1: Denotations of a pushdown system

are applicable to this configuration, where the deciding factors are the global state and the top stack symbol. By definition, a head can also be considered as a special type of configuration. If two PDS rules have the same head, the rules are said to be non-deterministic, since the execution of either rule is non-deterministically decided.

Given a rule $r = \langle g, \gamma \rangle \hookrightarrow \langle g', \omega \rangle \in \Delta$, 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 $\langle g, \gamma v \rangle \Rightarrow \langle g', \omega v \rangle$, 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 $\mathcal{P}$ is a sequence of configurations, $c_0 \Rightarrow c_1 \Rightarrow \ldots c_i \Rightarrow \ldots$, 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. Figure 2.1 lists some frequently-used PDS denotations for reference convenience.

2.1.3 Concurrent System

A *concurrent system* consists of a set of components that execute together [22]. Normally, there are two modes of execution: *asynchronous* or *interleaved execution*, in which only one component transitions at a time, and *synchronous execution* in which all the components transition at the same time. A concurrent system with an asynchronous execution mode is referred to as an *asynchronous system*; otherwise it is referred to as a *synchronous system*.

In this dissertation, we use two similar phrases with different meanings: the phrase *synchronous* (respectively, *asynchronous*) ***execution*** is concerned with whether concurrent components must transition at the same time to make a system transition; on the other hand, the phrase *synchronous* (respectively, *asynchronous*) ***transition*** is concerned with how transitions of concurrent components may affect each other, i.e., the dependent relation between transitions (see Chapter 4).

2.2 PROPERTY SPECIFICATION LANGUAGES

For reasoning about transition systems, temporal logics have been widely used as formal property specifications. In a temporal logic, the usual operators of propositional logic are augmented by *temporal operators*, which are used to form assertions about changes in time. One can assert, for example, that if proposition p holds in the present, then proposition q holds at some instant in the future, or at some instant in the past.

Temporal logics differ in the temporal operators that they provide and the semantics of those operators. For example, Computation Tree Logic, CTL [21], is a branching-time logic that describes time in a tree-like structure, where temporal operators can be used to quantify over the paths that are possible from a given state. In contrast, Linear Temporal logic (LTL) [71], only provides operators for describing events along a single computation path. This dissertation will discuss the model checking algorithm of co-verification for LTL properties in Chapter 5.

Although temporal logics are powerful to define a semantics for programs, e.g., termination and pre-/post- conditions, they are not intuitive in software property specifications when the source language constructs are desired. Furthermore, since software programs usually have infinite states, it is desired that such feature is also considered in the property specification language. SLIC is a software property specification language designed for SLAM engine. Different from temporal logics such as LTL, SLIC can be infinite, as it can count. However, SLIC is restricted to safety properties. Chapter 6 will utilize SLIC to specify safety properties in co-verification.

2.2.1 Linear Temporal Logic (LTL) Formula

LTL formulae are built up on a set of propositional variables, the common logic connectives, and a set of temporal operators, where the common logic connectives are:

- Negation (not), denoted by $\neg$;
- Conjunction (and), denoted by $\bigwedge$;
- Disjunction (or), denoted by $\bigvee$;
- Material implication (if...then), denoted by $\rightarrow$;
- Biconditional (if and only if), denoted by $\leftrightarrow$;

and the temporal operators are:

- $\boldsymbol{X}$ for next;
- $\boldsymbol{G}$ for always, i.e., globally;
- $\boldsymbol{F}$ for eventually, i.e., in the future;
- $\boldsymbol{U}$ for until;
- $\boldsymbol{R}$ for release.

Let At be a finite set of atomic propositions and $a \in At$ be a propositional variable. An LTL formula can be built according to the following syntax:

$$\varphi ::= a|\neg\varphi|\varphi_1 \bigwedge \varphi_2|\varphi_1 \bigvee \varphi_2|\varphi_1 \rightarrow \varphi_2|\varphi_1 \leftrightarrow \varphi_2|\boldsymbol{X}\varphi|\boldsymbol{G}\varphi|\boldsymbol{F}\varphi|\varphi_1\boldsymbol{U}\varphi_2|\varphi_1\boldsymbol{R}\varphi_2$$

For example, a termination property of a software thread can be expressed as

$$\boldsymbol{F}\ exit,$$

where *exit* is the label on the statement when the thread exits. Such formula states that all runs of the thread will eventually terminate. Another formula

$$\boldsymbol{G}\ (\boldsymbol{F}\ p),$$

states that the propositional variable p will repeatedly become true during the system execution.

2.2.2 Specification Language for Interface Checking (SLIC)

The SLIC language is designed to specify the temporal safety properties of Application Program Interfaces (APIs) implemented in the C language. A SLIC specification, S, defines a state machine that monitors the behavior of the program $P||L$ at APIs' procedural interface, where $P||L$ is the sequential composition of the client P that uses APIs and the library L that provides APIs. An interface state is a triple $(A, \{call, return\}, \Omega\})$, where A is a procedure, the second component indicates that control is being passed to A by a call or that control is returning from A to its caller, and Ω is a valuation to the formal parameters of procedure and the return value of A. The state machine rejects certain finite execution traces (a sequence of interface states) of $P||L$, either because P makes improper use of the API implemented by L or because L does not properly implement the API. SLIC uses a C-like syntax as presented in Figure 2.2. Figure 2.3 illustrates an example of a SLIC rule for a global queue of integers. The rule states that it is an error to have more than four zeroes in the queue.

2.3 MODEL CHECKING

Model checking is an automatic technique that tests whether a model of a system complies with a property specification. It was introduced on finite state systems by Clarke and Emerson [21] and independently by Queille and Sifakis [74]. Since hardware systems are finite state, model checking was most often applied to hardware designs [43, 54]. In the last decade, model checking of software implementations which are usually infinite state has achieved major progress [4, 14]. Such software model checking techniques implement a counterexample-guided abstraction and refinement process, i.e., an iterative process that asymptotically introduces more details to the abstraction based on infeasible counterexamples. This dissertation utilizes two software model checkers for co-verification of reachability properties

Syntax	Comment
S ::= *state* *transFun*$^+$	A SLIC specification consists of a state structure, and a list of transfer functions.
state ::= **state {** $\mathit{fieldDecl}^+$ **}**	A state structure is a list of field declarations.
fieldDecl ::= *ctype id* = *expr*;	A field has a C type, an identifier and an initialization expression.
transFun ::= *pattern stmt*	A transition function consists of a pattern and a statement.
pattern ::= *id* **.** *event* $\mid$ **[** *idList* **]** **.** *event*	
event ::= **entry** $\mid$ **exit**	
stmt ::= $\mathit{id}^+ = \mathit{expr}^+$; $\mid$ **if (** *choose* **)** *stmt* [**else** *stmt*] $\mid$ **abort** string**;** $\mid$ **halt;** $\mid$ **{** *stmt* **}**	Parallel assignment statement.
chose ::= ***** $\mid$ *expr*	Non-deterministic choice
expr ::= *id* $\mid$ *expr op expr* ...	Pure expression sub-language of C
idList ::= *id* $\mid$ *idList* **,** *id*	
id ::= *C_identifier* $\mid$ $\mathbf{\$}\ \mathit{int}$ $\mid$ $\mathbf{\$}$ **return** $\mid$ $\mathbf{\$}\ \mathit{C\_identifier}$	refer to fields of state structure $\$i$ refers to i^{th} formal parameter return value of a function global variable

Figure 2.2: Syntax of the SLIC language.

```
state {
    int zero_cnt = 0;
}

get.exit {
    if ($return == 0)
        zero_cnt = zero_cnt − 1;
}
```

```
put.entry {
    if ($1 == 0) {
        if (zero_cnt == 4)
            abort “Queue has 4 zeroes!”;
        else
            zero_cnt = zero_cnt + 1;
    }
}
```

Figure 2.3: SLIC specification for a simple property of a global queue.

and LTL properties respectively.

2.3.1 SLAM Engine for C Programs

As illustrated in Figure 2.4, the SLAM model checking engine contains three major parts to conduct reachability analysis on an C program instrumented with SLIC rules: (1) *Abstraction.* C2BP [5], a predicate abstraction engine, translates the instrumented C program into a Boolean program. Boolean programs are equivalent in power to pushdown automata, which accept context-free language. (2) *Check.* Bebop [6], a symbolic model checker for Boolean programs, conducts reachability analysis on the Boolean program. If no bug is detected, verification terminates

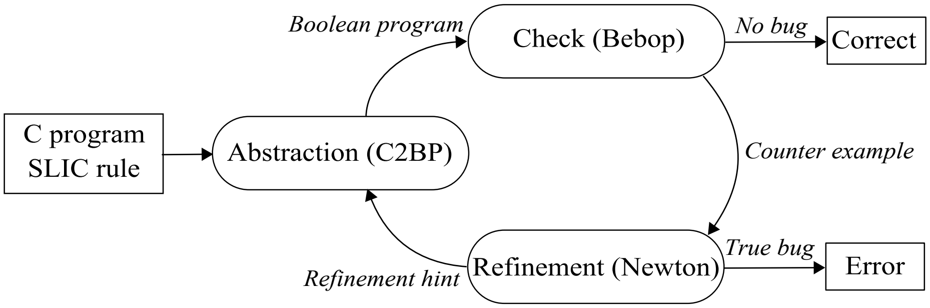


Figure 2.4: The abstraction-check-refinement loop of SLAM

with a SLIC rule pass. (3) *Refinement.* When Bebop finds an error trace, if the trace is confirmed to be feasible, SLAM reports the bug; otherwise, if the trace is infeasible, SLAM uses Newton [8] to generate new predicates that can eliminate the spurious path. This Abstraction-Check-Refinement loop usually ends when the check step cannot find any error trace or an error trace is confirmed to be a bug. SLAM supports reachability analysis of sequential programs. Figure 2.3 shows an example of a SLIC rule.

2.3.2 Moped Engine for Pushdown Systems

Moped, a model checker developed by Schwoon, supports both reachability analysis and LTL checking of pushdown systems [77]. Since PDSs are equivalent in power to Boolean programs, Bebop can be replaced by Moped in SLAM.

For LTL checking, Moped employs an automata-theoretic approach. Given an LTL formula, Moped first converts it to a BA $\mathcal{B} = (\Sigma, Q, \delta, q_0, F)$, and then makes a product of $\mathcal{B}$ with the target PDS $\mathcal{P} = (G, \Gamma, \Delta, \langle g_0, \omega_0 \rangle)$ to construct a *Büchi pushdown system*, $\mathcal{BP} = ((P \times Q), \Gamma, \Delta', \langle (p_0, q_0), w_0 \rangle, G)$. Let $L : (G \times \Gamma) \to \Sigma$ be a labeling function that associates the head of a PDS transition rule with the set of propositions that hold on it. $\mathcal{BP}$ is built such that:

- $\langle (p, q), \gamma \rangle \hookrightarrow \langle (p', q'), w \rangle \in \Delta'$, if $\langle p, \gamma \rangle \hookrightarrow \langle p', w \rangle$, $q \xrightarrow{\sigma} q'$, and $\sigma \subseteq L(\langle p, \gamma \rangle)$.
- $(p, q) \in G$, if $q \in F$.

The model checking problem of an LTL formula on a PDS is then reduced into the problem that computes an accepting run of the BA. It is important to note that Moped constructs a Büchi pushdown system in such a way that the BA monitors the state transitions of the PDS; thus, there is no interaction between the BA and PDS. This is different from the Büchi pushdown system constructed for co-verification in Chapter 4, where interactions between the BA and PDS go in both directions.

2.4 PARTIAL ORDER REDUCTION

One common method for reducing the complexity of model-checking concurrent systems is partial order reduction [33, 69]. This approach is based on an observation that properties in question often do not distinguish among the state-transition orders in concurrent systems. Traditional partial order reduction algorithms use an explicit state representation and depth-first search. The state space and transitions to be searched are selected during the model checking process; therefore, model checkers have to be customized for the reduction.

Normally, there are two types of transitions that help decide the selection process during reachability analysis [33]. *Persistent sets* describe the set of transitions that should be searched at a state, so that the verification result is conservatively preserved. A persistent set is constructed in such a way that no transition of the set can be disabled by any execution sequence of transitions that are not in the set. *Sleep sets* describe the set of transitions that can be avoided at a state without affecting the verification result. The key idea behind a sleep set is that if (1) a transition has been explored, and (2) it is independent with all the transitions on the searching path thereafter, this transition is unnecessary to be explored even though it is enabled in current state.

With respect to partial order reduction for LTL checking, a common approach is the *ample set methods*, which computes a set of state transitions that needs to be explored at each state during model checking. Peled [69] demonstrated that a number of conditions must be enforced on ample sets so that the truth value of the property to be checked is preserved in the reduced model.

Figure 2.5 illustrates the process of model checking with traditional partial order reductions. Because this approach requires modifications in the model checker and is often applied with depth-first search, it is difficult to apply the reduction

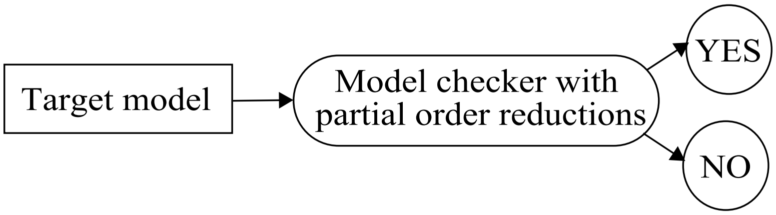


Figure 2.5: Model checking with traditional partial order reduction.

with other techniques that use breadth-first search, e.g., the symbolic model checking based on Binary Decision Diagrams (BDDs) [53].

Kurshan, et al. [44] 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. Therefore, no modification to the model checker is necessary. As illustrated in Figure 2.6, the model is reduced during the compilation phase by exploring the structure of the system specification. Any model checker that accepts the original model can be used to solve the verification problem of the reduced model.

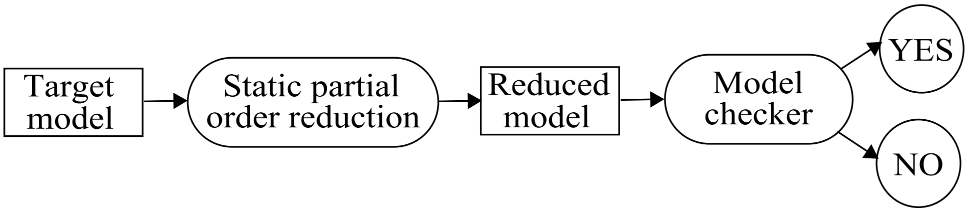


Figure 2.6: Model checking with static partial order reduction.

2.5 WINDOWS DEVICE/DRIVER STACK

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 [82], drivers are organized in stacks as illustrated in Figure 2.7. Each layer of a driver stack services a specific type of device in the corresponding

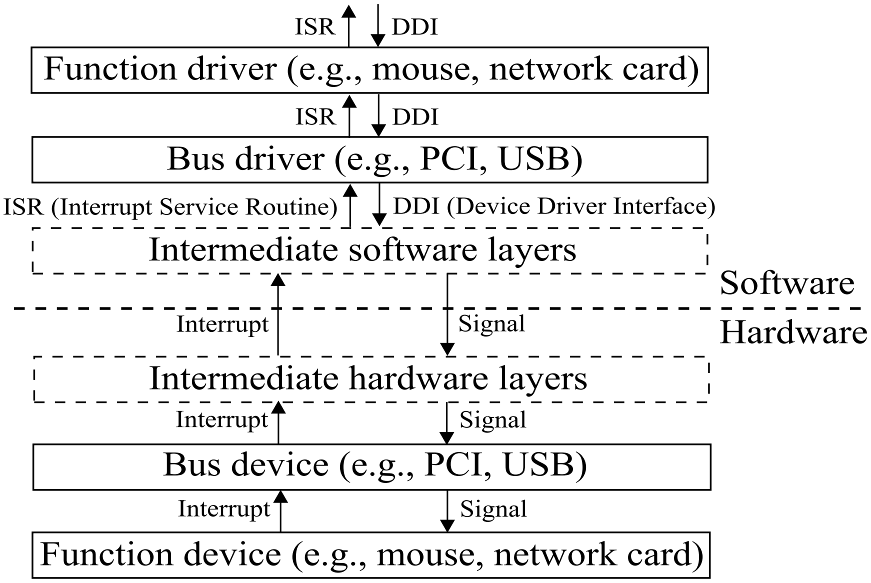

ISR
DDI
Function driver (e.g., mouse, network card)
ISR
DDI
Bus driver (e.g., PCI, USB)
ISR (Interrupt Service Routine)
DDI (Device Driver Interface)
Intermediate software layers
Software
Hardware
Interrupt
Signal
Intermediate hardware layers
Interrupt
Signal
Bus device (e.g., PCI, USB)
Interrupt
Signal
Function device (e.g., mouse, network card)

Figure 2.7: A generic view of Windows device/driver stacks.

hardware stack. One common method to classify device/driver layers is by deciding whether devices of a layer interconnect other devices. If so, this type of device is called as a bus device, e.g., PCI (Peripheral Component Interconnect) bus or USB (Universal Serial Bus) bus; otherwise, this type of device is referred to as a function device, e.g., a network adapter card connected to the PCI bus or a USB mouse connected to the USB bus. We usually refer to function devices (respectively, function drivers) directly as devices (respectively, drivers) for simplicity. For example, a PCI function device is referred to as a PCI device.

In the stack shown in Figure 2.7, in addition to the target device and driver layers whose HW/SW interface protocols we want to specify, there may be other layers in between. We refer to these layers as intermediate layers. Co-specification (see Chapter 3) needs to abstract the intermediate layers in such a way that the interactions between the target device and driver layers are properly modeled.

Different layers of a driver stack usually have different I/O interfaces. For example, USB drivers read USB device registers using Device Driver Interface

(DDI) functions such as WdfUsbRetrieveConfigDescriptor; and PCI drivers read device registers using DDI functions such as READ_REGISTER_UCHAR.

A Windows driver example. Figure 2.8 illustrates the excerpts from an Open System Resources (OSR) sample driver [67] for a PCI device, Sealevel PIO-24 digital I/O card [78]. This driver will be used as an example in the rest of this dissertation. The digital I/O 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 raises a data-ready interrupt. The driver inputs data when the data-ready interrupt is raised 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). For example, when an application sends down an I/O request, IOCTL_WDFDIO_READ_PORTA_AFTER_INT, to read data, the callback function, DioEvtDeviceControl, stores this request and marks the driver's status variable, AwaitingInt, to be true. Therefore, when the device raises an interrupt later, data will be read from the device by the ISR, DioIsr. Because ISRs run at the highest priority in a system, they preempt and block all other system routines [62]. Therefore, ISRs should return as quickly as possible. If necessary, an ISR will schedule a lower-priority routine for post-interrupt processing of the received data, where this kind of routine is commonly referred to as a Deferred Procedure Call (DPC). As illustrated in Figure 2.8c, the DPC routine, DioDpc, returns the data read by ISR to upper applications and completes the I/O request with proper status indicating the result of the operation.

Chapter 7 will discuss that the driver excerpt in Figure 2.8 contains a bug which can cause invalidate data being returned to upper applications. This bug is discovered by our co-verification tool, CoVer.

```
    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;
            ...
        }
        ...
    }
```

(a) Device Driver Control.

Figure 2.8: Excerpts from OSR sample driver code for PIO-24 digital I/O card.

```
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);
}
```

(b) Interrupt Service Routine (ISR).

Figure 2.8: Excerpts from OSR sample driver code for PIO-24 digital I/O card.

```
    BOOLEAN DioDpc( ... ) {
        ...
        // Is there a read-after-interrupt request in progress?
        if (devContext->CurrentRequest) {
            // Get a pointer to the I/O request.
            req = devContext->CurrentRequest;
            devContext->CurrentRequest = NULL;
            // Get the data that was read in ISR
P4:         data = devContext->PortAValueAtInt;
        }
        // Is there a pending request?
        if(req) {
            PUCHAR dataBuffer;
            ...
            // Retrieve the buffer for the input data
            status = WdfRequestRetrieveOutputBuffer(req, 0, (PVOID*)&dataBuffer, &length);
            if(NT_SUCCESS(status)) {
                // Return the data to the user – Just 1 byte, read in the ISR
                *dataBuffer = data;
                // Complete the request with success, pass back 1 in the information field
P5:             WdfRequestCompleteWithInformation(req, STATUS_SUCCESS, 1);
                return; // This request is successfully completed
            } else {
                WdfRequestCompleteWithInformation(req, STATUS_INVALID_REQUEST, 0);
            }
    }
```

(c) Deferred Procedure Call (DPC).

Figure 2.8: Excerpts from OSR sample driver code for PIO-24 digital I/O card.

Chapter 3

CO-SPECIFICATION

As the first step of co-verification, we need to specify HW/SW interface protocols. Such specification should capture the asynchronous hardware behaviors, the asynchronous software behaviors, and the interactions between hardware and software (i.e., synchronous behaviors of hardware and software). For various verification foci, the models constructed by co-specification can be used in different ways. For example, we can combine a driver implementation with its device model to verify whether the driver correctly operates its device; or we can verify the design of a hardware device using its driver model as the test harness. In any case, the specification semantics should be precise, so that automatic tools can be applied to validate the specification; the specification framework should consider the differences between hardware and software, so that hardware and software can be described in such a way close to their implementation semantics; and the specification process should exploit the unique features of HW/SW interactions, so that reductions can be applied to alleviate the cost of co-verification.

With respect to hardware and software, there are three types of concurrency in a system, i.e., the hardware concurrency, the software concurrency, and the HW/SW concurrency. One major challenge to co-specification is how to present the three types of concurrency in a proper level of abstraction so that irrelevant details are abstracted away while the specification still preserves the essential system behaviors. We utilize the concept of Transaction Level Modeling (TLM) to specify hardware behaviors. A hardware transaction is essentially a hardware state transition that is atomic in the view of software. Hardware concurrency is specified using hardware transactions with non-determinism. We also propose a semantic model, relative atomicity, to characterize the fact that concurrent components (hardware transactions and software threads) often have different execution priorities. Any concurrency characterized by relative atomicity can be represented by a Pushdown System (PDS), which is very useful to simplify the models constructed by co-specification and therefore, reduce the co-verification cost. Our co-specification language, modelC, as designed based on a restricted C semantics, supports both relative atomicity and non-determinism.

The co-specification framework describes HW/SW interface protocols using three parts: the HW/SW interface specification, the hardware specification, and the software specification. The HW/SW interface specification describes how hardware and software should transition synchronously when they interact with each other. The hardware specification describes the desired hardware behaviors when hardware and software transition asynchronously, i.e., when there is no HW/SW interaction. The software specification describes the desired operation sequences for software to control hardware. The three parts together specify the complete behaviors of a system.

We choose hardware devices and software drivers as the application domain of our research. In industrial settings, device/driver (HW/SW) interface protocols are commonly presented via English specifications. English does not have formal semantics; therefore, English specifications usually have ambiguities and inconsistencies. In co-specification, we write formal models to describe the behaviors of devices and drivers with respect to their interface protocols. Such a co-specification framework should also be utilized in the development process of devices and drivers; therefore, we not only will gain formal semantics for device/driver interface specifications but also can utilize automatic tools to validate these specifications. Furthermore, the formal models from co-specification can serve as the basis of a uniform platform not only for co-verification, but also for co-simulation, conformance testing, etc. Following our co-specification framework, we present a mechanized process to construct formal models of device/driver interface protocols from English specifications. We also discuss how to integrate our approach into the device/driver development process and propose the evaluation criteria for our approach.

3.1 SPECIFICATION TECHNIQUES FOR HW/SW INTERFACES

In the scope of co-verification, a system contains both hardware and software. There are three types of concurrency in such systems, i.e., the hardware concurrency, the software concurrency, and the HW/SW concurrency. One major challenge to co-specification is how to capture the various types of concurrency in a proper level of abstraction so that irrelevant details are abstracted away while the specification still preserves essential system behaviors.

First, we briefly discuss the three types of concurrency and how they are related to the HW/SW interface specification. Only the system behaviors that are closely related to HW/SW interface protocols should be preserved. Second, we discuss the TLM for hardware. TLM is a common practice in hardware design, where the design logic is specified by transaction functions while the implementation details such as clock signals are abstracted away. Third, we present a semantic model, relative atomicity. Although there is a lot of concurrency existing in a system, its complexity can be greatly reduced by characterizing the execution priorities existing in HW/SW interface designs. Fourth, we elaborate on how we utilize the concept of non-determinism to abstract away details unnecessary for interface specifications. Fifth, we present our co-specification language, modelC.

3.1.1 Concurrency in a System

Hardware concurrency. Hardware is concurrent in nature and hardware concurrency exists at various levels of design abstractions. In the view of software, we consider two types of hardware concurrency:

- concurrency between hardware modules; and
- concurrent assignments to registers.

For example, the Intel Ethernet Controller [39] has sub-modules such as command unit, receiving unit, interrupt management, etc., which are fully concurrent. Since they may not be driven by the same clock signal, we should consider their execution as asynchronous. A sub-module can be further divided into smaller sub-modules or directly implemented [38]. When a module is directly implemented, its operation consists of a sequence of steps that are driven by a clock signal. The states of the module are maintained in hardware registers and updated simultaneously upon clock cycles. How the registers should be updated during a clock cycle depends on the registers' states before the clock cycle and the state transition rule specified for the hardware design.

Software concurrency. Device drivers are commonly multi-threaded to service different requests such as interrupts from hardware, I/O requests from user applications, etc. In the view of hardware, we consider two types of software concurrency:

- multiple threads concurrently operate hardware, e.g., read/write hardware interface registers; and
- an Interrupt Service Routines (ISR) is invoked to service a hardware interrupt, where the current executing thread is preempted [60].

Conceptually, we can understand each thread as a PDS. The threads together should be represented as a product of the PDSs, which results in a Concurrent

Pushdown System (CPDS) [73]. Note that even reachability analysis of CPDS is undecidable [75]. Therefore, it is desired that software behaviors are modeled using a single PDS as much as possible. As we shall demonstrate later, the second type of software concurrency can be represented as a single PDS following the semantic model of relative atomicity. We will specify both types of software concurrency; therefore, our approach can be utilized in the formalization of HW/SW interface specifications (see Section 3.3). However, our co-verification model (see Chapter 4) and co-verification algorithms (see Chapter 5) will only address the second type of software concurrency (due to the decidability issue).

HW/SW concurrency. A device and its driver are mostly asynchronous and only transition synchronously when they interact through their interface. The HW/SW concurrency describes such situations:

- mostly, software and hardware transition asynchronously, where their states do not affect each other; and
- when hardware and software interact with each other, their synchronous transition will be decided by the states of both hardware and software.

3.1.2 Transaction Level Modeling (TLM) of Hardware

We utilize the TLM concept to specify hardware behaviors. TLM is a common approach to hardware design, where the key concept is to abstract away implementation details at the design stage so that one can focus on the design logic of a system.

Hardware transaction. Since our goal is to specify HW/SW interface protocols, the design logic, rather than the implementation details, is relevant. Therefore, modeling the clock-driven semantic feature of hardware implementations is not necessary. For example, a data-transfer command is usually processed in multiple clock cycles; however, it may only be necessary to describe this command as one hardware state transition from the view of software. 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 that are only necessary for synthesizable Register Transfer Level (RTL) designs. In the rest of this dissertation, we will describe hardware state transitions on the abstraction level of transactions instead of RTL level.

Hardware transaction function. We define a hardware transaction function as a C function that describes a set of hardware transactions (i.e., hardware state transitions). Because transactions are atomic, the intermediate hardware states during a transaction are invisible to software. In other words, the hardware state variables are simultaneously updated by a hardware transaction function from the software point of view. We define the current-states and next-states of a hardware 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 hardware transaction function, $\mathcal{F} : Q \times Q$, describes a set of hardware state transitions. Following this definition, any terminating C function can be treated as a hardware transaction function. In order to differentiate the definition of hardware transaction functions from other C functions, we use the keyword `__atomic` to indicate the type of hardware transaction function (see Figure 3.2 for example).

3.1.3 Relative Atomicity

Concurrent threads usually have different execution priorities. Since higher-priority threads preempt lower-priority threads, they should be considered atomic to the lower-priority threads. Relative atomicity captures this semantic feature: the execution of a higher-priority thread is atomic to that of a lower-priority thread. Any concurrency that follows the relative atomicity model can be represented by a single PDS. In our HW/SW interface protocol specifications, relative atomicity mainly captures two ideas:

- hardware transactions are atomic in the view of software; and
- ISRs are atomic with respect to other software routines, since ISRs have the highest priority.

Consider the execution model of a software program and a hardware design, the program contains a set of statements that are atomic in the view of hardware; and the hardware design is specified as a hardware transaction function. Algorithm 3.1 illustrates the scheduling algorithm for such an execution model, which demonstrates the idea of relative atomicity. The algorithm runs a hardware transaction or an atomic software statement based on non-deterministic choices. In computer systems, when some hardware raises an interrupt, the Operating System (OS) typically calls all the ISRs that are registered in the interrupt vector table

Algorithm 3.1 RelativeAtomicity()

```
if Non-deterministic-Choice() then
    "Run hardware transaction function for one time"
else
    if Interrupt-Pending() or Isr-Running() then
        "Run one atomic statement of ISR"
    else
        "Run one atomic statement of lower-priority routines"
    end if
end if
```

in sequence until an ISR acknowledges its ownership of the interrupt. During this process, only one ISR can run at a time and other hardware interrupts on the same bus are suppressed [62]. Although uncommon, it is possible that a driver provides ISRs with different priorities, where an ISR can preempt another lower-priority ISR. However, this does not affect the atomicity of an ISR with respect to other lower-priority driver routines. Furthermore, the higher-priority ISR is atomic with respect to the lower-priority ISR.

Relative atomicity captures the execution semantics of a system with different execution priorities; therefore, we can represent such execution semantics using a less complex model, PDS, compared to CPDS. In verification, relative atomicity can help us not only achieve decidability but also reduce complexity. Since relative atomicity is based on the observation of real execution semantics between hardware and software, there is no abstraction in this semantic model. However, it is important to note that when we need to represent more than one concurrent software thread with the same priority in a verification run, relative atomicity may not be applicable; therefore, we need a CPDS as the formal model for software.

3.1.4 Non-determinism in Co-specification

In co-specification, we utilize non-determinism mainly in two ways: (1) updating the variable values; and (2) deciding the conditions of branches or loops. For both ways, the use of non-determinism abstracts away the details unnecessary for interface specification. For example, one important utilization of non-determinism in our approach is how we model the hardware concurrency.

Non-deterministic interleaving. Hardware is concurrent in nature. For example, a network card processes software command and receives data concurrently. To specify this kind of hardware concurrency, we design an approach called non-deterministic interleaving which has three steps:

1. identify the concurrent modules (e.g., command unit, receive unit, etc.) of the target hardware device;
2. specify the modules using separate C functions which we refer to as module functions; and
3. non-deterministically invoke these module functions in a hardware transaction function.

The hardware concurrency is simulated in such a way that the module functions are executed in a non-deterministic sequence when the hardware transaction function is executed multiple times (see Section 3.2 for examples).

3.1.5 The modelC Language

As an important part of co-specification, we need a modeling language. Currently, the C language (or its variants) is commonly used in TLM specifications, since C semantics is widely understood by both hardware and software developers. Therefore, we present a modeling language, modelC, based on C semantics.

The *modelC language* uses C semantics with two extensions to support non-determinism and relative atomicity as well as the following three restrictions:

- numbers are treated as bounded integers so that hardware registers can be properly modeled;
- unbounded recursion is not allowed; and
- dynamic memory allocation is not allowed.

It is important to note that modelC is simply a C language dialect with these extensions and restrictions. Hardware description languages such as SystemC [66] and Verilog [38] also can be adapted to support the formal specification following our approach.

3.2 SPECIFICATION OF HW/SW INTERFACE PROTOCOLS

We demonstrate how we specify the HW/SW interface protocols through an example. One important rule for our specification is to capture all possible HW/SW behaviors that are allowed by interface protocols. This rule provides guidance for our modeling, which converges through refinement processes assisted by automation tools.

As illustrated in Figure 3.1, our formal specification has: a HW/SW interface, a hardware model, and a software model, where the hardware states are specified using the (hardware) global variables; the software states are specified using both the (software) global variables and stack contents of modelC programs.

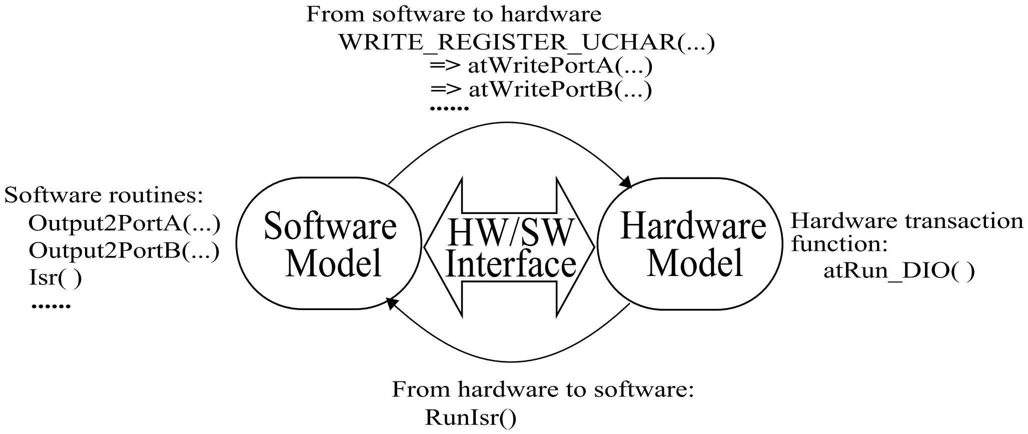


Figure 3.1: Co-specification framework.

The HW/SW interface describes how hardware and software should transition synchronously when they interact with each other. Consider the PIO-24 digital I/O device/driver interface (see Section 2.5): when software writes to the hardware interface registers by invoking `WRITE_REGISTER_UCHAR`, we specify how the interface registers should be updated using hardware transaction functions such as `atWritePortA`, `atWritePortB`, etc. In the other direction, hardware can raise an interrupt which will cause the software to invoke an ISR to service the interrupt.

We model this process using a function, `RunIsr`.

The hardware model describes the behaviors of hardware when it transitions asynchronously with software, i.e., when there is no HW/SW interaction. We specify the hardware model using one hardware transaction function, `atRun_DIO`.

The software model describes the desired operation sequences for software to control hardware. For each functionality, we use a function to describe the operation sequences of software. For example, the function, `Output2PortA`, specifies the sequence of operations that software should take in order to write to a hardware port of the PIO-24 digital I/O device.

3.2.1 HW/SW Interface Specification

The HW/SW interface, the abstraction of the HW/SW intermediate layers (see Figure 2.7) between the target device and driver, propagates hardware (respectively, software) interface events to software (respectively, hardware). Conceptually, a HW/SW interface has two parts: interface states and interface events. Interface states are state variables provided either by hardware or software and accessible by both. Interface events have two types: hardware or software. When hardware updates the software interface states, a hardware interface event occurs, and vice versa. For example, when a device raises an interrupt, the HW/SW interface will set the interrupt pending status and invoke the corresponding ISR to service the interrupt. On the other hand, when a driver writes to a hardware interface register, the HW/SW interface will update the related hardware registers accordingly. In general, the HW/SW interface describes the synchronous transitions of hardware and software when an interface event occurs.

Figure 3.2 illustrates a software interface event function, `atWritePortA`, which is actually a hardware transaction function in response to a software register write operation. This example describes a set of hardware state transitions when the driver writes to the interface register, Port A, of the PIO-24 digital I/O device.

```
__atomic VOID atWritePortA ( 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;
    }
}
```

Figure 3.2: An implementation of a software interface event function in the form of a hardware transaction function.

Figure 3.3 shows how function calls to a software write-register function (originally provided by the OS) are related to interface event functions. A software interface event occurs when the entry stack symbol of the interface event function is reached.

When hardware raises an interrupt, the ISR should be invoked to service this interrupt. The HW/SW interface simulates this process as shown in Figure 3.4. The variable `IsrRunning` represents the software status and the variable `InterruptPending` represents the hardware status. The function `RunIsr` has three steps, (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 `__atomic` blocks are used to indicate that the first and third steps describe synchronous state transitions of both hardware and software.

```
VOID WRITE_REGISTER_UCHAR
    (PUCHAR Register, UCHAR ucRegData) {
  switch ( Register ) {
    case REG_PORTA:  atWritePortA(ucRegData); return;
    case REG_PORTB:  atWritePortB(ucRegData); return;
    . . .
    case REG_CONFIG:  atWriteConfig(ucRegData); return;
    case REG_STATUS:  atWriteStatus(ucRegData); return;
    default: abort “Register address error.”; return;
  }
}
```

Figure 3.3: Relating register calls to software interface event functions.

3.2.2 Hardware Specification

The hardware model describes the behaviors of hardware when it works asynchronously with software to realize system functionalities. Consider the PIO-24 digital I/O device: when there is an input to Port A, the hardware model decides whether an interrupt should be raised based on both the current hardware state and the input value. Figure 3.5 illustrates an example of a hardware transaction function, `atRun_DIO`, that models the set of state transitions for the PIO-24 device when this device executes asynchronously with the driver. During each execution of the hardware transaction function, one module function (such as `RunPorts` or `RunInterrupt`) is non-deterministically selected; therefore, only one module executes and its related state variables get updated. The concurrency between these modules is simulated by non-deterministic interleaving between the module functions when the hardware transaction function is executed multiple times.

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

    DioIsr();    // Invoke the ISR

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

Figure 3.4: Interrupt monitoring function.

```
__atomic VOID atRun_DIO() {
    switch ( choice() ) {    // non-deterministic choices
        case 0: RunPorts(); break;    // Port I/O Management
        case 1: RunInterrupt(); break;    // Interrupt Management
        . . .
    }
}
```

Figure 3.5: Hardware transaction function of the PIO-24 digital I/O card device model.

3.2.3 Software Specification

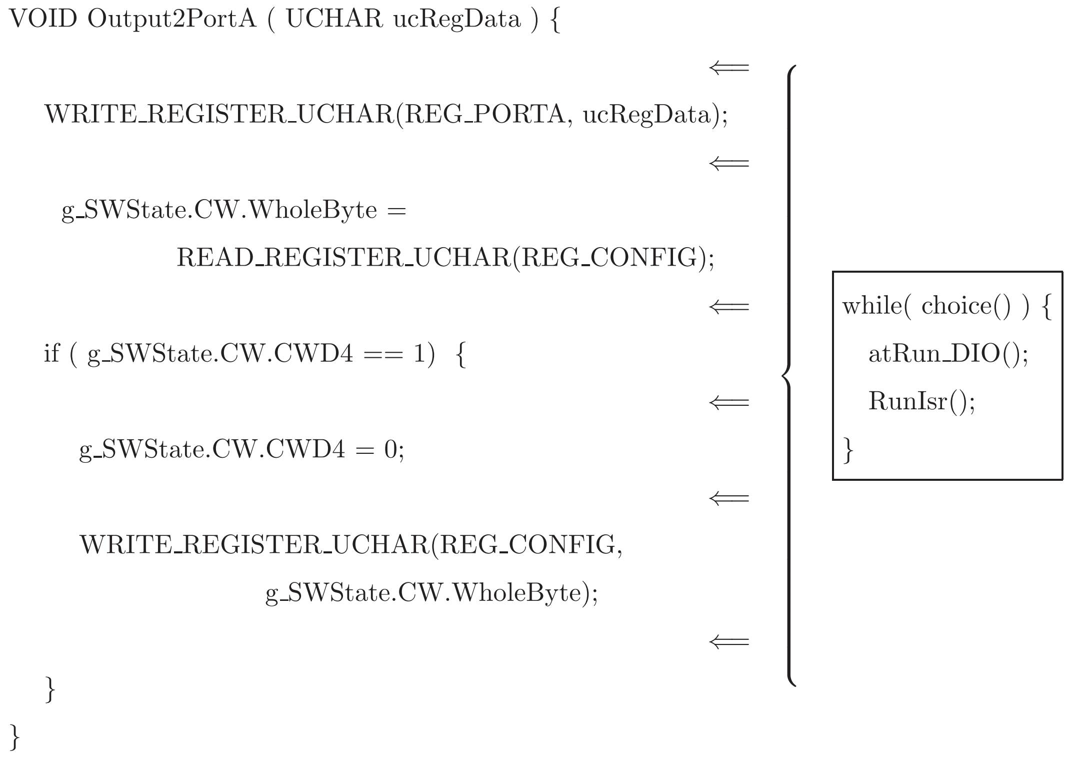
The software model describes the desired operation sequences for software to control hardware. It is straightforward to specify software behaviors using modelC, because modelC is designed based on the C semantics. In the English documents for HW/SW interface protocols, software specifications are usually categorized by functionality. For every functionality, a piece of English-based pseudo-code is provided to describe the desired software operations. We use a C function to replace each of the pseudo-code pieces. Figure 3.6 illustrates an example of such a C function for the PIO-24 driver model. This function describes the desired software operations for outputting a byte to Port A. Conceptually, all these C functions are implemented in several concurrent driver threads (the number of threads and

```
VOID Output2PortA ( UCHAR ucRegData ) {
    // Write to Port A
    WRITE_REGISTER_UCHAR(REG_PORTA, ucRegData);
    // Read the I/O configuration
    g_SWState.CW.WholeByte =
            READ_REGISTER_UCHAR(REG_CONFIG);
    // If Port A is configured as “input”, set it as “output”
    if ( g_SWState.CW.CWD4 == 1 ) {
        g_SWState.CW.CWD4 = 0;
        WRITE_REGISTER_UCHAR(REG_CONFIG,
                    g_SWState.CW.WholeByte);
    }
}
```

Figure 3.6: A C function for outputting to Port A.

how the functions should be assigned to the threads depend highly on implementation details). Some types of software concurrency can be captured by relative atomicity and, therefore, represented by a single PDS. For example, the thread for `DioIsr` (see Figure 3.4) should always be atomic to other driver threads, because ISRs have the highest priority. A CPDS is necessary as the software representation when relative atomicity is inapplicable, i.e., (1) there is more than one unbounded stack of software threads; and (2) the context-switches between the threads are also unbounded.

3.2.4 A Realization of Relative Atomicity



```
VOID Output2PortA ( UCHAR ucRegData ) {
                                                    ⟸
    WRITE_REGISTER_UCHAR(REG_PORTA, ucRegData);
                                                    ⟸
      g_SWState.CW.WholeByte =
                   READ_REGISTER_UCHAR(REG_CONFIG);
                                                    ⟸     while( choice() ) {
    if ( g_SWState.CW.CWD4 == 1)  {                          atRun_DIO();
                                                    ⟸        RunIsr();
        g_SWState.CW.CWD4 = 0;                            }
                                                    ⟸
        WRITE_REGISTER_UCHAR(REG_CONFIG,
                          g_SWState.CW.WholeByte);
                                                    ⟸
    }
}
```

Figure 3.7: Execution model of relative atomicity.

Given the examples of HW/SW interface, hardware model, and software model, Figure 3.7 illustrates how they can be combined into a single-threaded program following the concept of relative atomicity. After every software statement, we nondeterministically invoke the hardware transaction function, `atRun_DIO`, to let the asynchronous hardware model run. Meanwhile, since a hardware transaction may raise an interrupt, we invoke the interrupt monitoring function, `RunIsr`; therefore, ISR can be invoked if an interrupt has been raised. Note that software statements are not really atomic in this example; however, we use this format to retain the readability. There are two types of synchronous transitions between hardware and software. First, when software invokes register operation functions such as `WRITE_REGISTER_UCHAR`, the hardware transaction functions such as `atWritePortA` will be invoked to update the hardware interface registers. Second, in `RunIsr`, the ISR routine will be invoked if an interrupt has been raised.

3.2.5 Summary and Generalization

We have demonstrated our co-specification framework via an example. Our approach is also applicable to other common HW/SW interfaces in devices/drivers and microcode/firmware. There are three reasons:

1. TLM is already widely used in hardware development. Hardware designers usually specify transaction level models in order to evaluate the performance and correctness of their designs.

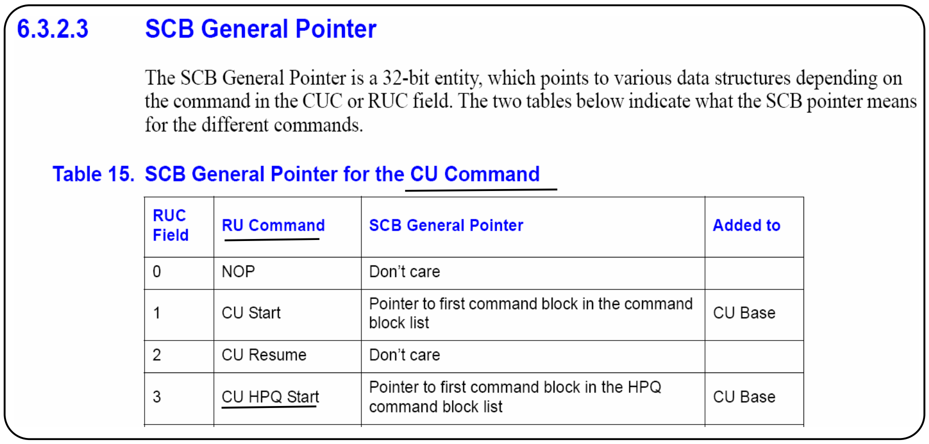
2. In HW/SW interface designs, it is standard to have different execution priorities for concurrent components such as software threads and hardware transactions. For example, when hardware raises an interrupt, software needs to service the interrupt in a high priority thread to prevent losing any volatile hardware state.

3. The models constructed by co-specification can be formally represented by BA and PDS, which are suitable formal representations for hardware and

software respectively (see Chapter 4). Furthermore, BA and PDS have already been successfully used in hardware and software verification.

3.3 APPLICATIONS AND EVALUATION CRITERIA

In industrial settings, hardware and software are often manufactured separately, because their development processes require highly different expertise. English is the de facto language for specifying HW/SW interface protocols. Since lots of HW/SW interfaces are public standards, their English specifications have to be self-explanatory. However, English does not have formal semantics, so these specifications commonly contain ambiguities and inconsistencies. A single misinterpretation of an interface protocol can cause bugs in products, which will likely lead to system failures. These failures are hard to diagnose especially when they happen only in a product release as a specific combination of device and driver.



6.3.2.3 SCB General Pointer

The SCB General Pointer is a 32-bit entity, which points to various data structures depending on the command in the CUC or RUC field. The two tables below indicate what the SCB pointer means for the different commands.

Table 15. SCB General Pointer for the CU Command

RUC Field	RU Command	SCB General Pointer	Added to
0	NOP	Don't care	
1	CU Start	Pointer to first command block in the command block list	CU Base
2	CU Resume	Don't care	
3	CU HPQ Start	Pointer to first command block in the HPQ command block list	CU Base

Figure 3.8: An excerpt from Intel 10/100 Mbps Ethernet Controller document.

Figure 3.8 illustrates an excerpt from the English document of the Intel 8255x 10/100Mbps Ethernet Controller Specification [39]. This excerpt describes how the shared memory between hardware and software should be operated by hardware when a CU/RU (Command Unit/Receive Unit) command is issued from software.

There are two issues: First, the content of `Table 15` is inconsistent with its title (underlined, the RU and CU difference). Second, the `CU HPQ Start` command is neither defined nor mentioned in any other part of this document. This is quite confusing when compared to the `CU Start` command. Such specification issues are pervasive (see Chapter 7 for details) in various English documents for HW/SW interfaces. These issues can cause confusion, produce bugs, and lead to product failures.

English has four significant drawbacks as the HW/SW interface specification language:

- First, since English lacks a formal semantics, we cannot guarantee a unique interpretation from the same English specification.

- Second, no automatic tool can be offered to validate the correctness of an English-based interface protocol due to lack of formal semantics.

- Third, implementation semantics of HW/SW interfaces are quite different from the specification semantics; therefore, the implementations can significantly deviate from their interface protocols.

- Fourth, it is not straightforward to design test cases for HW/SW interface implementations based on English specifications. Instead of using a systematic approach, ad-hoc test cases are commonly used.

To address the drawbacks of English specifications, we need to describe HW/SW interface protocols in a precise manner, so that the protocols can be specified formally and analyzed automatically. We refer to this specification process as HW/SW interface formalization.

In our co-specification framework, we construct formal models of HW/SW interface protocols. Compared to English-based specifications, there are three advantages of our approach in specifying HW/SW interface protocols:

- First, our specification is exact and free of ambiguities, since a programming language such as modelC is used to describe the HW/SW interface protocols rather than English.

- Second, our specifications can be represented as formal models. Therefore, automatic formal verification tools (such as CoVer presented in Chapter 6) can be used to ensure that critical properties about HW/SW interface protocols are correctly expressed through co-specification.

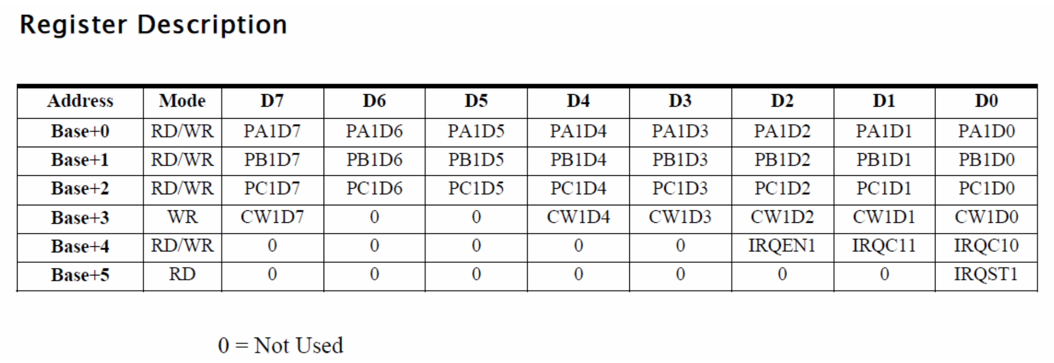
- Third, the formal hardware (respectively, software) specifications can be readily utilized in the co-verification with software (respectively, hardware) implementations.

In this section, we first present a mechanized process to construct formal models of HW/SW interface protocols from English specifications. This process is quite helpful to apply our approach to legacy HW/SW interfaces, where their English specifications already exist. Second, we discuss how to integrate our approach into the HW/SW development process. Finally, we propose the evaluation criteria for our approach.

3.3.1 Formalization Process from English Specifications

Specifications of HW/SW interface protocols are often presented in structured English that describes HW/SW interface protocols in three parts:

1. *Definition of hardware interface registers.* Hardware uses interface registers to receive commands from software, and in the other direction provide software with the current hardware state. Figure 3.9 illustrates an excerpt from the English specification of the PIO-24 device [78]. This excerpt presents the offsets and access (read/write) modes of the 8-bit registers, where the names from `D0` to `D7` represent the bits of a register. The register names such



Register Description

Address	Mode	D7	D6	D5	D4	D3	D2	D1	D0
Base+0	RD/WR	PA1D7	PA1D6	PA1D5	PA1D4	PA1D3	PA1D2	PA1D1	PA1D0
Base+1	RD/WR	PB1D7	PB1D6	PB1D5	PB1D4	PB1D3	PB1D2	PB1D1	PB1D0
Base+2	RD/WR	PC1D7	PC1D6	PC1D5	PC1D4	PC1D3	PC1D2	PC1D1	PC1D0
Base+3	WR	CW1D7	0	0	CW1D4	CW1D3	CW1D2	CW1D1	CW1D0
Base+4	RD/WR	0	0	0	0	0	IRQEN1	IRQC11	IRQC10
Base+5	RD	0	0	0	0	0	0	0	IRQST1

0 = Not Used

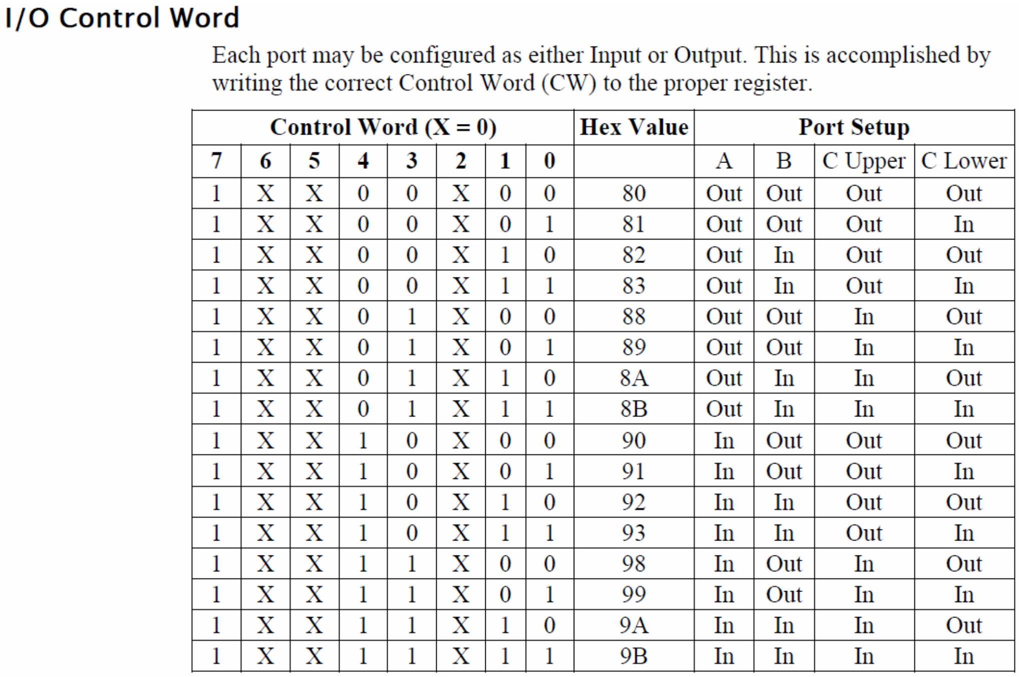
Figure 3.9: Excerpt of PIO-24 specification: hardware interface registers.

as PA (a.k.a., Port A), PB (a.k.a., Port B), etc. are defined in the English specification (omitted in the excerpt). The PIO-24 driver relies on register offsets to access different registers. Figure 3.10 illustrates how we define the register offset macros based on the English specification. For example, REG_PORTA corresponds to the name PA; REG_CONFIG corresponds to the name CW (a.k.a., Control Word). As another example, Figure 3.11 illustrates the excerpt about how the I/O direction of port registers are controlled by the value of the CW register. The hardware transaction function atWritePortA illustrated in Figure 3.2 is specified according to this excerpt, i.e., the Port A register is configured as "input" (respectively, "output") if the D4 bit of the CW register is 1 (respectively, 0). The symbol X represents that this bit is not used, which is indicated in Figure 3.9.

2. *Hardware (environment) behaviors.* Mostly, hardware execution is asynchronous with software, e.g., when hardware processes software commands or inputs from the environment. Figure 3.12 illustrates the excerpt from the English specification about how the PIO-24 device should process the input to Port A and raise an interrupt to notify the driver. We specify this behavior using the module function RunInterrupt illustrated in Figure 3.13. RunInterrupt is invoked by the hardware transaction function atRun_DIO

```
#define      REG_PORTA      BASE_ADDRESS + 0

#define      REG_PORTB      BASE_ADDRESS + 1

#define      REG_PORTC      BASE_ADDRESS + 2

#define      REG_CONFIG     BASE_ADDRESS + 3

#define      REG_IRQ        BASE_ADDRESS + 4

#define      REG_STATUS     BASE_ADDRESS + 5
```

Figure 3.10: Macros for interface register offsets of the PIO-24 device.

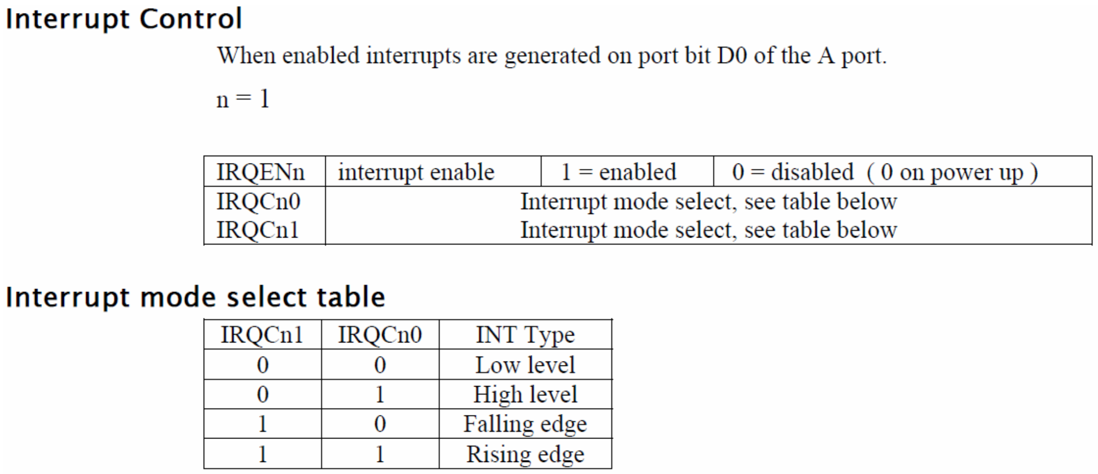


I/O Control Word

Each port may be configured as either Input or Output. This is accomplished by writing the correct Control Word (CW) to the proper register.

Control Word (X = 0)								Hex Value	Port Setup			
7	6	5	4	3	2	1	0		A	B	C Upper	C Lower
1	X	X	0	0	X	0	0	80	Out	Out	Out	Out
1	X	X	0	0	X	0	1	81	Out	Out	Out	In
1	X	X	0	0	X	1	0	82	Out	In	Out	Out
1	X	X	0	0	X	1	1	83	Out	In	Out	In
1	X	X	0	1	X	0	0	88	Out	Out	In	Out
1	X	X	0	1	X	0	1	89	Out	Out	In	In
1	X	X	0	1	X	1	0	8A	Out	In	In	Out
1	X	X	0	1	X	1	1	8B	Out	In	In	In
1	X	X	1	0	X	0	0	90	In	Out	Out	Out
1	X	X	1	0	X	0	1	91	In	Out	Out	In
1	X	X	1	0	X	1	0	92	In	In	Out	Out
1	X	X	1	0	X	1	1	93	In	In	Out	In
1	X	X	1	1	X	0	0	98	In	Out	In	Out
1	X	X	1	1	X	0	1	99	In	Out	In	In
1	X	X	1	1	X	1	0	9A	In	In	In	Out
1	X	X	1	1	X	1	1	9B	In	In	In	In

Figure 3.11: Excerpt of PIO-24 specification: meaning of `CW` (a.k.a., the Control Word register).



Interrupt Control

When enabled interrupts are generated on port bit D0 of the A port.

n = 1

IRQENn	interrupt enable	1 = enabled	0 = disabled (0 on power up)
IRQCn0	Interrupt mode select, see table below		
IRQCn1	Interrupt mode select, see table below		

Interrupt mode select table

IRQCn1	IRQCn0	INT Type
0	0	Low level
0	1	High level
1	0	Falling edge
1	1	Rising edge

Figure 3.12: Excerpt of PIO-24 specification: input to Port A.

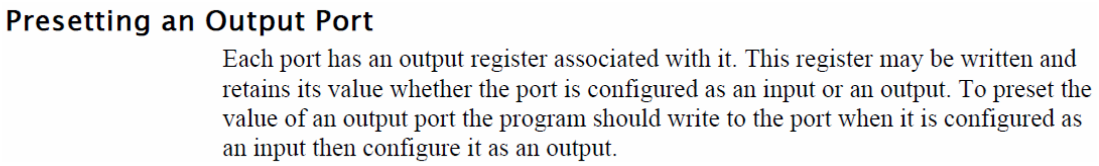
which describes the asynchronous hardware behavior in the view of software. Based on the current hardware state and input to Port A from the environment, `RunInterrupt` decides whether or not hardware should raise an interrupt. Note that the input from the environment can be modeled by assigning non-deterministic values to the Port A register after every execution of `atRun_DIO`.

3. *Recommendations for software to operate hardware.* Software often needs to follow certain rules when operating hardware. Since software implementations are more flexible than hardware, the rules are usually presented in the form of recommendations. For example, Figure 3.14 illustrates an excerpt that recommends the desired behaviors for software to output to the ports of the PIO-24 device. Our software specification illustrated in Figure 3.6 captures this recommendation.

Given an English specification of a HW/SW interface protocol, we summarize our formalization process of developing the corresponding formal model by three steps as illustrated in Figure 3.15. First, we identify the structure of the target

```
VOID RunInterrupt() {
    // If interrupt is disabled
    if(g_DIORegs.IRQ.IRQENn == 0) goto Exit;

    // If Port A is not configured as input
    if(g_DIORegs.CW.CWD4 != 1) goto Exit;
    . . .

    // Low level triggers an interrupt
    if( (g_DIORegs.IRQ.IRQCn == 0) && (g_DIORegs.A.D0 == 0) )
        g_DIORegs.IRQST.IRQST1 = 1;
    // High level triggers an interrupt
    else if( (g_DIORegs.IRQ.IRQCn == 1) && (g_DIORegs.A.D0 == 1) )
        g_DIORegs.IRQST.IRQST1 = 1;
    // Falling edge triggers an interrupt (we assume that if this path
    // is executed, falling edge just occurred)
    else if( g_DIORegs.IRQ.IRQCn == 2 )
        g_DIORegs.IRQST.IRQST1 = 1;
    // Rising edge triggers an interrupt (we assume that if this path
    // is executed, rising edge just occurred)
    else if( g_DIORegs.IRQ.IRQCn == 3 )
        g_DIORegs.IRQST.IRQST1 = 1;

Exit: return;
}
```

Figure 3.13: The module function, `RunInterrupt`, invoked by `atRun_DIO` (see Figure 3.5).



Presetting an Output Port

Each port has an output register associated with it. This register may be written and retains its value whether the port is configured as an input or an output. To preset the value of an output port the program should write to the port when it is configured as an input then configure it as an output.

Figure 3.14: Excerpt of PIO-24 specification: a recommendation for software.

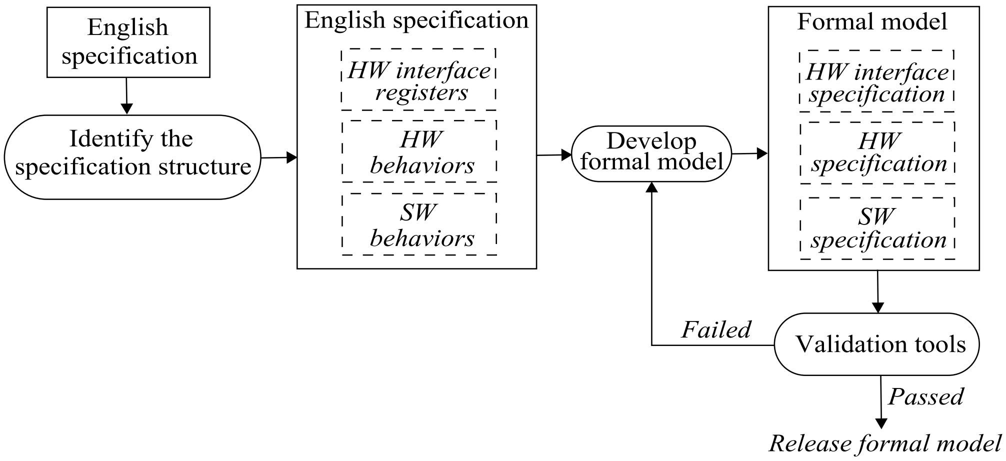


Figure 3.15: The formalization process from an English specification.

English specification. The goal of this step is to decide the three specification parts of the HW/SW interface protocol: definition of hardware interface registers, hardware behaviors, and recommendations for software behaviors. Second, we develop the formal model from the English specification following the mappings between the structure of the English specification and that of the formal model. We have already discussed how to develop each part of the formal model from the corresponding part of the English specification. For example, Figure 3.9 and Figure 3.10 illustrate how to define the hardware interface registers' offsets based on the English specification. Third, we utilize automatic tools to validate the formal model. For example, a C compiler can easily detect inconsistencies in the formal model; CoVer can verify correctness properties of the formal model. Issues discovered during the validation are sent back to the development process for refinement. The formal model can be released only if no validation fails.

3.3.2 Applications in the HW/SW Development Process

A *device/driver framework* refers to a type of HW/SW interface as well as the devices and drivers that both utilize this interface. Device/driver frameworks can have different levels of abstractions. For example, the PCI device/driver framework refers to the PCI HW/SW interface, all the PCI devices, and all the PCI drivers; the Sealevel PIO-24 device/driver framework refers to the PIO-24 HW/SW interface, the PIO-24 driver, and the PIO-24 device. Note that, although the PIO-24 HW/SW interface is built on the PCI HW/SW interface, the two interfaces describe different HW/SW interface protocols and are logically separate.

We have observed a common development process for device/driver frameworks in industrial settings as illustrated in Figure 1.1. The process contains three stages:

Design stage. Usually, a device/driver framework is designed by a group of hardware and software companies together. The HW/SW interface is described in a draft English specification which is shared between these participant companies for revision. Engineers from these companies proof-read the English specification and try to identify potential problems in the HW/SW interface design. However, there are no automatic tools that can be used to help identify specification problems because the HW/SW interface protocols are not formally specified.

Development stage. After the English specification has been agreed upon by the participant companies, it is made public. The companies will start to develop their own hardware (respectively, software) products for this device/driver framework based on the English specification. During this stage, other companies, who have not participated in the design stage, may also develop their own hardware (respectively, software) products that are compliant with this device/driver framework. How well a product complies with the HW/SW interface protocol depends greatly on the development engineers' interpretations of the English specification. In order to further ensure the HW/SW interface compliance, a product also needs to be tested according to the English specification. Because test engineers from different companies may have their own interpretations of the specification, the test cases vary; therefore, the test coverage of different products can be significantly different, as well as the products' quality in terms of the HW/SW interface compatibilities. Figure 3.16 illustrates such an example, which contains two excerpts respectively from a Linux driver and a Windows driver for the same hardware device, the Intel 8255x 10/100Mbps Ethernet controller. The two C functions respectively illustrated in Figure 3.16a and Figure 3.16b have the same functionality which is to issue a software command to the device; however, the implementations are different. Before issuing a new command, the Linux driver always waits until the command register becomes free (this rule is indicated by the English specification); however the Windows driver does not wait before issuing any new command unless the parameter `WaitForScb` is set to be true, a performance optimization.

```
int e100_exec_cmd( nic *nic, u8 cmd,
    dma_addr_t dma_addr ) {
  int err = 0;
  . . .
  spin_lock_irqsave( . . . );

  /* Previous command is accepted when SCB clears */
  for (i = 0; i < E100_WAIT_SCB_TIMEOUT; i++) {
    /* If last command has been completed */
    if (likely(!ioread8(&nic->csr->scb.cmd_lo))) break;
    cpu_relax();
    if (unlikely(i > E100_WAIT_SCB_FAST)) udelay(5);
  }
  /* If last command timeout */
  if (unlikely(i == E100_WAIT_SCB_TIMEOUT)) {
    err = -EAGAIN;
    goto err_unlock;
  }

  /* Issue a new command */
  if (unlikely(cmd != cuc_resume))
    iowrite32(dma_addr, &nic->csr->scb.gen_ptr);
  iowrite8(cmd, &nic->csr->scb.cmd_lo);

err_unlock: spin_unlock_irqrestore( . . . );
  return err;
}
```

(a) Linux driver code excerpt.

```
NTSTATUS D100IssueScbCommand(
  PFDO_DATA FdoData,
  PUCHAR ScbCommandLow,
  BOOLEAN WaitForScb ) {

  // Wait for the last command to complete?
  if ( WaitForScb == TRUE ) {

    // If last command timeout
    if ( !WaitScb(FdoData) )
      return
        (STATUS_DEVICE_DATA_ERROR);

  }

  // Issue a new command
  WRITE_REGISTER_UCHAR (
    ((PUCHAR)(FdoData->CSRAddress +
          SCB_COMMAND_LOW_BYTE),
    ScbCommandLow );

  return (STATUS_SUCCESS);
}
```

(b) Windows driver code excerpt.

Figure 3.16: Excerpts from the Linux and Windows drivers for the Intel 8255x 10/100Mbps Ethernet controller.

Obviously, the Windows driver is more efficient, because it tries to avoid unnecessary checks on hardware registers. On the other hand, it is also more challenging to maintain the driver's correctness, because the driver developer must guarantee that when `D100IssueScbCommand` is called with `WaitForScb` being `FALSE`, the command register should always be free.

Post-release stage. After a product passes in-house testing, the company may choose to ship it to market directly or send it to a third-party organization for conformance testing. Conformance testing decides whether a hardware (respectively, software) product complies with the HW/SW interface protocol. Test cases are developed by engineers from the third-party organization based on the English specification. The effectiveness of the test cases highly relies on the engineers' interpretations of the specification. The product passes the certification if all the test cases succeed.

As discussed above, English specifications serve an important role to ensure the reliability of HW/SW interface implementations. Not only are the products developed based on English specifications, but the validation processes also rely on the precise understanding of English specifications.

In our approach of formalizing HW/SW interface specifications, formal models are employed to describe the HW/SW interface protocols instead of English specifications as illustrated in Figure 1.3. Our approach improves the development process of devices and drivers in the following four aspects:

- In the design stage of a device/driver framework, automatic verification tools are applied to check the correctness of the formal model which describes the HW/SW interface protocol. This is more efficient and reliable than manual proof reading.

- During the development of a product, the formal model is referred to rather than the English specification; therefore, it is easier for both development engineers and test engineers to have precise understanding about how hardware and software should interact following the HW/SW interface protocol.

- During in-house testing, because the formal model closely resembles the implementation semantics of both hardware and software, it can be readily utilized by validation techniques such as co-verification [49, 50] and co-simulation [56] (also see Section 8.2). This not only reduces the duplicate efforts in developing test harnesses but also provides a uniform and systematic platform for validation.

- In conformance testing, the formal model can also serve as the golden model. Two types of testing can be applied. First, equivalence checking/testing [70, 84, 87] can be used to check if a hardware (respectively, software) product complies with the hardware (respectively, software) formal model. Second, a hardware (respectively, software) formal model can be used as the test harness of the software (respectively, hardware) products. Furthermore, because the formal model is shared between manufacturing companies and certification organizations, product issues discovered by certification organizations will be easier to resolve with manufacturing companies.

Among these advantages, the ability to provide a uniform and systematic platform for validation is very important. In traditional testing, because devices and drivers are manufactured separately, some failures due to interface incompatibility only occur when a specific version of device is combined with a specific version of driver. It is hard to pinpoint the responsibility for such failures, because both the device and the driver are black boxes (or at least one of them is). Using formal specifications as the uniform validation platform will greatly relieve this problem.

Although formalizing the HW/SW interface specification can significantly help the development process and improve the product reliability, we do not expect to abandon English documents completely, because there are five aspects we do not specify in formalization:

- terminology definitions;
- introduction and architecture overview;
- non-interface-related specifications, e.g., device virtualization support;
- physical criteria, e.g., physical shapes and sizes of device interfaces; and
- timing criteria, e.g., the device initialization time (in milliseconds).

In other words, our formalization approach augments the English specifications of HW/SW interface protocols by formal models, so that the interaction logic between hardware and software can be precisely captured.

3.3.3 Evaluation Criteria

We propose four criteria for evaluating our approach: (1) whether a formal model is easy to read compared to its English specification; (2) how to ensure the correctness of the formal model, i.e., whether the HW/SW behaviors are correctly captured; (3) how much manual effort is required in the formalization process; and (4) how to compare the formal model with its English specification. We also present how we categorize the English specification issues into two types.

Reference convenience. We discuss this criterion from the perspective of hardware and software engineers respectively. In hardware design, it is very common that a high-level model of a product is firstly specified using programming languages such as C or SystemC so that the correctness and performance of the design can be evaluated by verification or simulation. After a hardware product is actually implemented as RTL design, the high-level model can also serve as the golden model for equivalence testing, because it is easier in practice to maintain the correctness of a high-level design rather than a complicated implementation that is optimized for performance. For software development, we have interviewed several device driver developers, they all agree that it is easier to refer to formal models rather than English specifications for HW/SW interface protocols. There mainly are two reasons: (1) device driver developers are familiar with the C language; and (2) IDEs (Integrated Development Environments) such as Microsoft Visual Studio [58] and Eclipse CDT (C/C++ Development Tooling) [29] can be utilized to help review the formal models.

Correctness assurance. We argue that the correctness of formal models is easier to maintain than that of English specifications, because in addition to manual review, we can apply all kinds of tools in order to help validate the correctness of formal models. For example, the C compiler alone can detect a large amount of specification inconsistencies; we can also use our co-verification tool, CoVer, to verify the correctness of formal models (see Chapter 7). Even for manual review, reading formal models are easier than reading English specifications with the help of IDEs.

Manual effort. The manual effort required in the specification of a formal model mainly depends on the complexity of the HW/SW interface protocol and the experience of the specification engineer. In general, the complexity of a HW/SW interface protocol can be approximately quantified by the size of its English document (i.e., the number of pages); the experience of specification engineers can be quantified by their years of experience in hardware and software development. More quantification about the evaluation of manual effort is discussed in Chapter 7.

Comparison with the English specification. It is important to compare formal models with their English specifications. Different English specifications may describe HW/SW interface protocols in different levels of details. However, enough details must be included when specifying a formal model using our approach, because formal models are designed to closely resemble the HW/SW implementation semantics. For example, an English specification may omit the input restrictions on a device's I/O port, however this detail must be specified explicitly in the formal model, i.e., if the device's input is not specified in the English specification, non-deterministic values should be given as the input in the formal model. We define a concept, model-document ratio, to help analyze the relation between a formal model and its English specification. Given that the formal model has L_{FM} lines of modelC code and the English document has P_{doc} pages of specification about the HW/SW interface protocol, we define the model-document ratio[1] as:

Definition 3.1. $\textsc{Model-Doc} = \frac{L_{FM}}{P_{doc}}$

When the Model-Doc ratio is high, it suggests that the HW/SW interface protocol is loosely described by the English specification; therefore, the deviations of HW/SW interface behaviors in various products may be very high. This is a potential hazard that can cause HW/SW interface incompatibility problems, because the less information provided by the English specification, the more interpretations development engineers have to make by their own. When the Model-Doc ratio is low, it suggests that the English specification is elaborate, where the HW/SW interface protocol is described in details. However, the lower a Model-Doc ratio is, the higher possibility the English specification may have inconsistencies. In other words, it is hard to maintain the consistency of an English specification when it is elaborate (usually large) and the same rule about a HW/SW interface protocol is described in many places.

Two types of specification issues in English documents.

[1] Since HW/SW interface specifications are presented in structured English, they are often similar in format and structures (but different in the level of details). Therefore, the pages can be used to indicate the size of the specifications.

- *Spec-inconsistency.* Multiple places of an English specification are partially contradictory to each other. For example, the first issue illustrated in Figure 3.8 is a spec-inconsistency issue, where the name `RU` contradicts the name `CU`.

- *Spec-incompleteness.* The information provided by an English specification is not enough to guide the implementation. As a result, engineers need to develop products based on their own interpretations, which can cause product failures due to the potential incompatibility between different HW/SW interface implementations. The second issue of Figure 3.8 is a spec-incompleteness issue, because the description about the `CU HPQ Start` command is incomplete.

The two types of specification issues commonly exist even after an English specification has been published for years and revised for several editions (see Chapter 7 for more examples). This is a strong reason for applying our approach, so that HW/SW interface specifications can be checked by automatic tools for correctness.

Chapter 4

CO-VERIFICATION MODEL

Chapter 3 has presented a co-specification framework for a system that contains hardware, software, and a hardware/software (HW/SW) interface. The specification cannot be formally analyzed until we provide a formal representation for it. Automata theory, as a formal method, studies abstract machines that are useful to represent concrete systems. These abstract machines usually capture only the certain aspects of target systems, for example, their design logic. Therefore, analysis on abstract machines is not only more precise but also more efficient than that on the original systems. Furthermore, the operations on abstract machines can be studied with mathematical methods so that we can reason about the correctness of the system designs.

Since hardware and software are designed in different approaches, they are commonly represented by different formal models. Because hardware has a bounded state space, finite state machines, such as *Büchi automata*, are commonly used as hardware formal models. On the other hand, software programs can have unbounded stacks; therefore, *pushdown systems* are more suitable for software compared to finite state machines. Since pushdown systems do not have input alphabet, they cannot take inputs from hardware models. Therefore, we extend pushdown systems to *labeled pushdown systems*, where each state transition of a labeled pushdown system is labeled by a symbol defined on the states of the hardware model.

Having separate formal models for hardware and software is cumbersome for co-verification. A unifying model that combines the merits of Büchi automata and labeled pushdown systems is desired so that hardware, software, and their interactions can be analyzed together as an integrated system. A *Büchi pushdown system*, as the Cartesian product of a Büchi automaton and a labeled pushdown system, is such a unifying model. Büchi pushdown systems capture both the synchronous transitions and asynchronous transitions of hardware and software.

State transitions of formal models can be described in different ways. By definition, a state transition specifies a relation between two states, i.e., the current-state which is the model state before the state transition and the next-state which is the model state after the state transition. We refer to such representation as an *explicit representation*. It is straightforward to explain algorithms using an explicit representation. However, the implementation of explicit representations is inefficient in practice, because complete state graphs must be built in order to realize any analysis algorithm. By contrast, *symbolic representations* describe a set of state transitions using a few *symbolic rules*. A symbolic rule, similar to a mapping function, specifies a relation between two sets of states, i.e., the set of current-states which are the possible model states before the transitions and the set of next-states which are the possible model states after the state transitions. Symbolic representations are more efficient than explicit representations in implementation. We will discuss both the explicit representations and the symbolic representations for the formal models used in co-verification.

4.1 BÜCHI AUTOMATON AS HARDWARE MODEL

Formally, a hardware design can be represented as a BA $\mathcal{B} = (\Sigma, Q, \delta, q_0, F)$. More specifically, the alphabet Σ is the power set of the set of propositions induced by software interface events. In other words, the evaluation of the propositional variables in Σ depends on whether a software interface event occurs. The set of states Q represents the hardware states. The initial state q_0 represents the initial hardware state. The transition relation R describes the hardware behavior, i.e.,

$\mathcal{B} = (\ \Sigma,\ Q,\ \delta,\ q_0,\ F\ )$

$\Sigma = \{\ \emptyset,\ \{reset\},\ \{no\_event\},\ \{stop\},\ \{reset, no\_event\},$
$\{no\_event, stop\}, \{reset, stop\},\ \{reset, no\_event, stop\}\ \}$

$Q = \{\ \texttt{Init},\ \texttt{Rst},\ \texttt{Wrk},\ \texttt{Idle},\ \texttt{Intr}\ \}$

$\delta = \{\ t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, t_9, t_{10}\ \}$

$t_1 = \texttt{Init} \xrightarrow{\{reset\}} \texttt{Rst}$

$t_2 = \texttt{Wrk} \xrightarrow{\{reset\}} \texttt{Rst}$

$t_3 = \texttt{Wrk} \xrightarrow{\{stop\}} \texttt{Init}$

$t_4 = \texttt{Rst} \xrightarrow{\emptyset} \texttt{Rst}$

$t_5 = \texttt{Rst} \xrightarrow{\emptyset} \texttt{Wrk}$

$t_6 = \texttt{Wrk} \xrightarrow{\{no\_event\}} \texttt{Idle}$

$t_7 = \texttt{Wrk} \xrightarrow{\{no\_event\}} \texttt{Intr}$

$t_8 = \texttt{Idle} \xrightarrow{\{no\_event\}} \texttt{Init}$

$t_9 = \texttt{Init} \xrightarrow{\{no\_event\}} \texttt{Init}$

$t_{10} = \texttt{Intr} \xrightarrow{\{no\_event\}} \texttt{Wrk}$

$q_0 = \texttt{Init}$

$F = \{\ \texttt{Init},\ \texttt{Wrk},\ \texttt{Idle}\ \}$

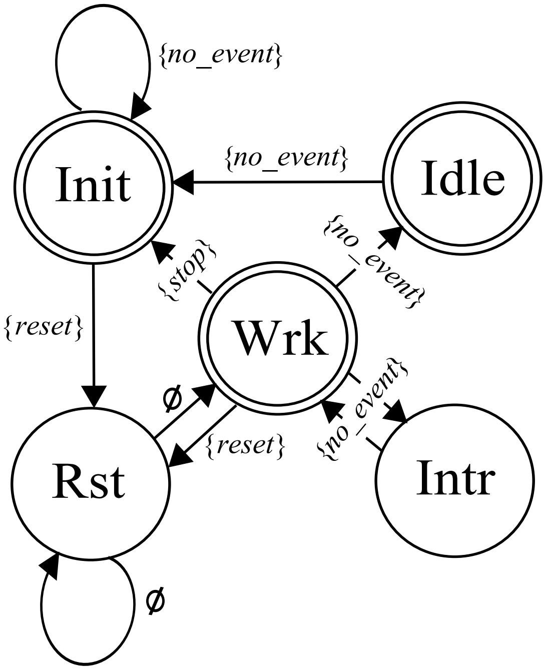


Figure 4.1: A hardware design represented by BA.

how hardware should transition from current state to next state. The set of final states (a.k.a., accepting states) F constrains the state transitions of the hardware design in such a way that at least one of the final states should be visited infinitely often.

Example. Figure 4.1 illustrates an example of a BA representing a hardware design. This BA has five states: the initial state (`Init`), the resetting state (`Rst`), the working state (`Wrk`), the idle state (`Idle`), and the interrupt triggering state (`Intr`). The transitions are labeled by the sets of propositions that are induced by software interface events. A transition is said to be *enabled* when all propositional variables in its label are true. In this example, there are two software interface events, i.e., reset and stop. The propositional variables *reset* and *stop* are true if and only if their corresponding software interface events occur. The propositional variable *no_event* is true if and only if no software interface event occurs. The empty set $\emptyset$ is always considered as true. If two transitions starting from the same state can be enabled by the same condition, the transitions are referred to as non-deterministic transitions. For example, t_4 and t_5 are non-deterministic transitions from the state `Rst`; and t_6 and t_7 are non-deterministic transitions from the state `Wrk`. Note that the alphabet Σ is constructed strictly following its definition as the power set of the set of propositions induced by software interface events; therefore, symbols such as $\{reset, no\_event\}$ and $\{no\_event, stop\}$ are elements of Σ by definition. They do not have concrete meaning in this example.

This hardware design is in the initial state, `Init`, by default. It requires a reset command from software in order to transition from the initial state to the working state, `Wrk`. While in the working state, the hardware can raise an interrupt by transitioning to the interrupt triggering state, `Intr`; it can also transition to the idle state, `Idle`, and consequently, transition to the initial state. If there is a software interface event *stop*, the hardware should transition from the working state to the initial state. If there is a software interface event *reset*, the hardware should transition from the working state to the resetting state, `Rst`. Since the resetting process may take time to complete, a delay is modeled by a self-loop transition on `Rst`, i.e., t_4. However, a resetting process should not delay infinitely. Both the resetting state and interrupt triggering state are intermediate states that model the resetting process and interrupt triggering event respectively. Therefore, the two states are not in the set of final states, F. An alternative way to specify the set of final states is via an LTL formula, i.e., $\boldsymbol{G}$ ($\boldsymbol{F}$ (`Init`||`Wrk`||`Idle`)), which states that one of the final states should be repeatedly visited.

4.2 LABELED PUSHDOWN SYSTEM AS SOFTWARE MODEL

4.2.1 Representing Software Design

Formally, a software program can be represented as a PDS $\mathcal{P} = (G, \Gamma, \Delta, \langle g_0, \omega_0 \rangle)$. More specifically, the set of states G represents the states of global variables. The set of strings Γ^* represents the states of a stack. A configuration, denoted by $\langle g, \omega \rangle \in G \times \Gamma^*$, represents a program state. The set of PDS rules Δ represents program statements. A PDS rule, written as $\langle g, \gamma \rangle \hookrightarrow \langle g', \omega \rangle \in (G \times \Gamma) \times (G \times \Gamma^*)$, summarizes a set of PDS state transitions in the form of $\langle g, \gamma v \rangle \Rightarrow \langle g', \omega v \rangle$, where $v \in \Gamma^*$. Since the stack may be unbounded, a PDS can have infinite states; therefore, the number of PDS transitions can also be infinite. However, the number of PDS rules is finite. In general, there are three types of PDS rules:

- $\langle g, \gamma \rangle \hookrightarrow \langle g', \gamma' \rangle$, where $\gamma, \gamma' \in \Gamma$, i.e., the stack control location transitions from γ to γ' without a procedure call, e.g., through an assignment;

- $\langle g, \gamma \rangle \hookrightarrow \langle g', \gamma' \gamma'' \rangle$, where $\gamma, \gamma', \gamma'' \in \Gamma$, i.e., a call to a procedure f such that γ' represents the entry point of f and γ'' represents the return address of the call;

- $\langle g, \gamma \rangle \hookrightarrow \langle g', \varepsilon \rangle$, where $\gamma \in \Gamma$ and ε denotes the empty string, i.e., a return statement.

Example. Figure 4.2 shows how a Boolean program can be represented as a PDS $\mathcal{P} = \{G, \Gamma, \Delta, \langle g_0, \omega_0 \rangle\}$, where

- $G = \{\diamond, a, !a\}$;

- $\Gamma = \{\sqcup, main_0, main_1, main_2, main_3, main_4, main_5, reset_0, reset_1, stop_0, stop_1, NonHWRelated_0, NonHWRelated_1\}$;

- Δ is given in Figure 4.2b;

	decl a;	$\langle g_0, \omega_0\rangle = \langle \diamond, \sqcup\rangle$, i.e., the initial configuration.
	void main() begin	$\langle \diamond,\ \sqcup\rangle \hookrightarrow \langle !a,\ main_0\rangle$
		$\langle \diamond,\ \sqcup\rangle \hookrightarrow \langle a,\ main_0\rangle$
0	reset();	$\langle \cdot,\ main_0\rangle \hookrightarrow \langle \cdot,\ reset_0\ main_1\rangle$
		"·" is used when the rules (1) do not modify a; and
		(2) are not affected by the value of a.
1	$a := 0$;	$\langle a,\ main_1\rangle \hookrightarrow \langle !a,\ main_2\rangle$
		$\langle !a,\ main_1\rangle \hookrightarrow \langle !a,\ main_2\rangle$
2	NonHWRelated();	$\langle \cdot,\ main_2\rangle \hookrightarrow \langle \cdot,\ NonHWRelated_0\ main_3\rangle$
3	if(a) then	$\langle a,\ main_3\rangle \hookrightarrow \langle a,\ main_4\rangle$
		$\langle !a,\ main_3\rangle \hookrightarrow \langle !a,\ main_5\rangle$
4	stop();	$\langle \cdot,\ main_4\rangle \hookrightarrow \langle \cdot,\ stop_0\ main_5\rangle$
	fi	
5	return;	$\langle \cdot,\ main_5\rangle \hookrightarrow \langle \cdot,\ \varepsilon\rangle$
	end	
	void reset() begin	
0	skip;	$\langle \cdot,\ reset_0\rangle \hookrightarrow \langle \cdot,\ reset_1\rangle$
1	return	$\langle \cdot,\ reset_1\rangle \hookrightarrow \langle \cdot,\ \varepsilon\rangle$
	end	
	void NonHWRelated() begin	
0	skip;	$\langle \cdot,\ NonHWRelated_0\rangle \hookrightarrow \langle \cdot,\ NonHWRelated_1\rangle$
1	return	$\langle \cdot,\ NonHWRelated_1\rangle \hookrightarrow \langle \cdot,\ \varepsilon\rangle$
	end	
	void stop() begin	
0	skip;	$\langle \cdot,\ stop_0\rangle \hookrightarrow \langle \cdot,\ stop_1\rangle$
1	return	$\langle \cdot,\ stop_1\rangle \hookrightarrow \langle \cdot,\ \varepsilon\rangle$
	end	

(a) Boolean program.

(b) Corresponding PDS rules.

Figure 4.2: Representing a Boolean program using PDS.

- $\langle g_0, \omega_0 \rangle = \langle \diamond, \sqcup \rangle$.

The Boolean program has a global variable a and four procedures named `main`, `reset`, `NonHWRelated`, and `stop`. The entry point of the program is the first statement of `main`, where the stack location is denoted by $main_0$. For every program statement, there are several corresponding PDS rules that model the transitions of both the stack locations and the data (e.g., the value of a). Since the global variable a is not initialized, there are two non-deterministic rules from the initial configuration to the configurations $\langle a, main_0 \rangle$ and $\langle !a, main_0 \rangle$ respectively. A center dot "$\cdot$" is used to simplify the representation of the PDS rules that (1) do not modify the data; and (2) are not affected by the value of the data.

This program is designed to operate hardware. The two procedures `reset` and `stop` are used to reset or stop the hardware respectively. Conceptually, a software interface event occurs when the entry stack location of such a procedure is reached. For example, when $reset_0$ is reached, the software interface event, reset, occurs; therefore, the propositional variable *reset* (see Figure 4.1) is evaluated as true. On the other hand, the Boolean program also can have computation steps that are not related to any hardware operations. For example, the propositional variable *no_event* is true on every stack location of the procedure `NonHWRelated`. In other words, the program statements in `NonHWRelated` do not operate hardware.

4.2.2 Accepting Inputs from Hardware

While PDS is a suitable model for programs, it is not designed to accept any inputs. In co-verification, hardware behaviors may also affect software executions, e.g., through an interrupt; therefore, the PDS software model should be extended to accept inputs from the BA hardware model.

Definition 4.1. A Labeled Pushdown System (LPDS), an extension of a pushdown system, is denoted as $\mathcal{P} = (I, G, \Gamma, \Delta, \langle g_0, \omega_0 \rangle)$, where I is a finite 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. An LPDS rule is written as $\langle g, \gamma \rangle \overset{\tau}{\hookrightarrow} \langle g', \omega \rangle \in \Delta$.

LPDS extends PDS in such a way that a rule in Δ is labeled by a symbol in I. In the context of co-verification, I is defined as the power set of the set of propositions that may hold on a state of $\mathcal{B}$. In the rest of this dissertation, the notation $\mathcal{P}$ represents an LPDS unless it is indicated otherwise. 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 if $c_0 = \langle g_0, w_0 \rangle$ is the initial configuration. Given an input string s, the reachability relation between two configurations $c, c' \in Conf(\mathcal{P})$ is written as $c \overset{s}{\Rightarrow}{}^* c'$. It can also be written as $c \Rightarrow^* c'$ if the input string is irrelevant to the context.

Example. In Figure 4.1, the hardware design raises an interrupt when it transitions to the interrupt triggering state, `Intr`. This hardware interface event should cause a context-switch in software so that the software's Interrupt Service Routine (ISR) will be executed. Figure 4.3a shows the implementation of such an ISR procedure. The PDS software model illustrated in Figure 4.2 can then be extended to an LPDS $\mathcal{P} = (I, G, \Gamma, \Delta, \langle g_0, \omega_0 \rangle)$ in such a way that:

- $I = \{\emptyset, \{intr\}, \{no\_intr\}, \{intr, no\_intr\}\}$, where the propositional variable $intr$ is true if and only if the BA (in Figure 4.1) is at the state `Intr`; otherwise the propositional variable no_intr is true. Note that $\{intr, no\_intr\}$ is an element of I by definition, although it does not have concrete meaning in this example;

- $G = \{\diamond, a, !a\}$;

```
     void isr() begin
0         a := 1;

1         return;
     end
```

(a) ISR in Boolean program.

$$\langle a,\ isr_0\rangle \overset{\emptyset}{\hookrightarrow} \langle a,\ isr_1\rangle$$
$$\langle !a,\ isr_0\rangle \overset{\emptyset}{\hookrightarrow} \langle a,\ isr_1\rangle$$
$$\langle \cdot,\ isr_1\rangle \overset{\emptyset}{\hookrightarrow} \langle \cdot,\ \varepsilon\rangle$$

(b) Corresponding LPDS rules.

Figure 4.3: An Interrupt Service Routine (ISR) procedure specified in Boolean program and LPDS respectively.

- $\Gamma = \{\sqcup, main_0, main_1, main_2, main_3, main_4, main_5, reset_0, reset_1, stop_0, stop_1, NonHWRelated_0, NonHWRelated_1, isr_0, isr_1\}$;
- Δ is given in Figure 4.3b, Figure 4.4b and Figure 4.5b, where all rules are labeled by symbols in I;
- $\langle g_0, \omega_0\rangle = \langle \diamond, \sqcup\rangle$.

An LPDS rule is said to be *enabled* when all propositional variables of its label are evaluated as true. Since the empty set, $\emptyset$, is always true, an LPDS rule labeled by $\emptyset$ is always enabled regardless the current state of the BA. All LPDS rules for the ISR procedure are labeled by $\emptyset$, because ISR has the highest execution priority. When an ISR is being executed, no more hardware interrupts can be serviced by software. On the other hand, procedures with the lower execution priority, such as `main`, can be interrupted by hardware; therefore their LPDS rules are labeled by the set $\{no\_intr\}$. A special kind of rule (bold in the figures) is introduced to model the context-switch to ISR when an interrupt is raised. These rules are all labeled by the set $\{intr\}$, i.e., when hardware raises an interrupt, ISR should preempt the current executing procedure to service the interrupt.

```
    decl a;
    void main() begin
0       reset();
1       a := 0;
2       NonHWRelated();
3       if(a) then
4           stop();
        fi
5       return;
    end
```

(a) Boolean program.

$\langle g_0, \omega_0 \rangle = \langle \diamond, \sqcup \rangle$, i.e., the initial configuration.

$\langle \diamond,\ \sqcup \rangle \overset{\emptyset}{\hookrightarrow} \langle !a,\ main_0 \rangle$

$\langle \diamond,\ \sqcup \rangle \overset{\emptyset}{\hookrightarrow} \langle a,\ main_0 \rangle$

$\langle \cdot,\ main_0 \rangle \xhookrightarrow{\{no\_intr\}} \langle \cdot,\ reset_0\ main_1 \rangle$

$\boldsymbol{\langle \cdot,\ main_0 \rangle \xhookrightarrow{\{intr\}} \langle \cdot,\ isr_0\ main_0 \rangle}$

$\langle a,\ main_1 \rangle \xhookrightarrow{\{no\_intr\}} \langle !a,\ main_2 \rangle$

$\langle !a,\ main_1 \rangle \xhookrightarrow{\{no\_intr\}} \langle !a,\ main_2 \rangle$

$\boldsymbol{\langle \cdot,\ main_1 \rangle \xhookrightarrow{\{intr\}} \langle \cdot,\ isr_0\ main_1 \rangle}$

$\langle \cdot,\ main_2 \rangle \xhookrightarrow{\{no\_intr\}} \langle \cdot,\ NonHWRelated_0\ main_3 \rangle$

$\boldsymbol{\langle \cdot,\ main_2 \rangle \xhookrightarrow{\{intr\}} \langle \cdot,\ isr_0\ main_2 \rangle}$

$\langle a,\ main_3 \rangle \xhookrightarrow{\{no\_intr\}} \langle a,\ main_4 \rangle$

$\langle !a,\ main_3 \rangle \xhookrightarrow{\{no\_intr\}} \langle !a,\ main_5 \rangle$

$\boldsymbol{\langle \cdot,\ main_3 \rangle \xhookrightarrow{\{intr\}} \langle \cdot,\ isr_0\ main_3 \rangle}$

$\langle \cdot,\ main_4 \rangle \xhookrightarrow{\{no\_intr\}} \langle \cdot,\ stop_0\ main_5 \rangle$

$\boldsymbol{\langle \cdot,\ main_4 \rangle \xhookrightarrow{\{intr\}} \langle \cdot,\ isr_0\ main_4 \rangle}$

$\langle \cdot,\ main_5 \rangle \xhookrightarrow{\{no\_intr\}} \langle \cdot,\ \varepsilon \rangle$

$\boldsymbol{\langle \cdot,\ main_5 \rangle \xhookrightarrow{\{intr\}} \langle \cdot,\ isr_0\ main_5 \rangle}$

(b) Corresponding LPDS rules.

Figure 4.4: Representing the procedure `main` using LPDS.

	void reset() begin	
0	skip;	$\langle \cdot,\ reset_0 \rangle \xhookrightarrow{\{no\_intr\}} \langle \cdot,\ reset_1 \rangle$
		$\boldsymbol{\langle \cdot,\ reset_0 \rangle \xhookrightarrow{\{intr\}} \langle \cdot,\ isr_0\ reset_0 \rangle}$
1	return	$\langle \cdot,\ reset_1 \rangle \xhookrightarrow{\{no\_intr\}} \langle \cdot,\ \varepsilon \rangle$
		$\boldsymbol{\langle \cdot,\ reset_1 \rangle \xhookrightarrow{\{intr\}} \langle \cdot,\ isr_0\ reset_1 \rangle}$
	end	
	void NonHWRelated() begin	
0	skip;	$\langle \cdot,\ NonHWRelated_0 \rangle \xhookrightarrow{\{no\_intr\}} \langle \cdot,\ NonHWRelated_1 \rangle$
		$\boldsymbol{\langle \cdot,\ NonHWRelated_0 \rangle \xhookrightarrow{\{intr\}} \langle \cdot,\ isr_0\ NonHWRelated_0 \rangle}$
1	return	$\langle \cdot,\ NonHWRelated_1 \rangle \xhookrightarrow{\{no\_intr\}} \langle \cdot,\ \varepsilon \rangle$
		$\boldsymbol{\langle \cdot,\ NonHWRelated_1 \rangle \xhookrightarrow{\{intr\}} \langle \cdot,\ isr_0\ NonHWRelated_1 \rangle}$
	end	
	void stop() begin	
0	skip;	$\langle \cdot,\ stop_0 \rangle \xhookrightarrow{\{no\_intr\}} \langle \cdot,\ stop_1 \rangle$
		$\boldsymbol{\langle \cdot,\ stop_0 \rangle \xhookrightarrow{\{intr\}} \langle \cdot,\ isr_0\ stop_0 \rangle}$
1	return	$\langle \cdot,\ stop_1 \rangle \xhookrightarrow{\{no\_intr\}} \langle \cdot,\ \varepsilon \rangle$
		$\boldsymbol{\langle \cdot,\ stop_1 \rangle \xhookrightarrow{\{intr\}} \langle \cdot,\ isr_0\ stop_1 \rangle}$
	end	

(a) Boolean program. (b) Corresponding LPDS rules.

Figure 4.5: Representing the procedures `reset`, etc. using LPDS.

4.3 UNIFYING MODEL FOR CO-VERIFICATION

As discussed in the previous sections, BA is a suitable model for hardware designs and LPDS is a suitable model for software programs. The state transitions of both the BA and LPDS are labeled by the symbols induced by the interface events of each other, so that the two models are synchronized. It is further desired that a unifying model is built to combine the BA and LPDS; therefore their behaviors can be analyzed together as an integrated system.

4.3.1 Preliminaries

Given a BA $\mathcal{B} = (\Sigma, Q, \delta, q_0, F)$ and an LPDS $\mathcal{P} = (I, G, \Gamma, \Delta, \langle g_0, \omega_0 \rangle)$, we define two labeling functions such that

- $L_{\mathcal{P}2\mathcal{B}} : (G \times \Gamma) \rightarrow \Sigma$, which associates the head of an 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 dissertation.
- $L_{\mathcal{B}2\mathcal{P}} : Q \rightarrow I$, which associates a state of $\mathcal{B}$ with the set of propositions that hold on it.

Three concepts are defined using the labeling functions: enabledness, indistinguishability, and independence.

Enabledness

A BA transition $t = q \xrightarrow{\sigma} q' \in \delta$ is *enabled* by an LPDS configuration $c \in Conf(\mathcal{P})$ (respectively, an LPDS rule $r = c \overset{\tau}{\hookrightarrow} c' \in \Delta$) if and only if $\sigma \subseteq L_{\mathcal{P}2\mathcal{B}}(c)$; otherwise t is *disabled* by c (respectively, r). On the other hand, an LPDS rule $r = c \overset{\tau}{\hookrightarrow} c' \in \Delta$ is *enabled* by a BA state $q \in Q$ (respectively, a BA transition $t = q \xrightarrow{\sigma} q' \in \delta$) if and only if $\tau \subseteq L_{\mathcal{B}2\mathcal{P}}(q)$; otherwise r is *disabled* by q (respectively, t).

Example 1. Given the following BA transition, LPDS rule, and instances of the labeling functions:

- a BA transition, $t = \texttt{Rst} \xrightarrow{\emptyset} \texttt{Wrk}$, as illustrated in Figure 4.1;
- an LPDS rule, $r = \langle a,\ main_3\rangle \xhookrightarrow{\{no\_intr\}} \langle a,\ main_4\rangle$, as illustrated in Figure 4.4b;
- $L_{\mathcal{P}2\mathcal{B}}(\langle a,\ main_3\rangle) = \{no\_event\}$;
- $L_{\mathcal{B}2\mathcal{P}}(\texttt{Rst}) = \{no\_intr\}$.

Because $\emptyset \subseteq L_{\mathcal{P}2\mathcal{B}}(\langle a,\ main_3\rangle)$, t is enabled by the LPDS rule r as well as all LPDS configurations in the form of $\langle a,\ main_3\ v\rangle$, where $v \in \Gamma^*$; because $\{no\_intr\} \subseteq L_{\mathcal{B}2\mathcal{P}}(\texttt{Rst})$, r is enabled by the BA transition t as well as the BA state $\texttt{Rst}$.

Example 2. Given the following BA transition, LPDS rule, and instances of the labeling functions:

- a BA transition, $t = \texttt{Wrk} \xrightarrow{\{stop\}} \texttt{Init}$, as illustrated in Figure 4.1;
- an LPDS rule, $r = \langle a,\ main_0\rangle \xhookrightarrow{\{intr\}} \langle a,\ isr_0\ main_0\rangle$, as illustrated in Figure 4.4b;
- $L_{\mathcal{P}2\mathcal{B}}(\langle a,\ main_0\rangle) = \{no\_event\}$;
- $L_{\mathcal{B}2\mathcal{P}}(\texttt{Wrk}) = \{no\_intr\}$.

Because $\{stop\} \nsubseteq L_{\mathcal{P}2\mathcal{B}}(\langle a,\ main_0\rangle)$, t is disabled by the LPDS rule r as well as all LPDS configurations in the form of $\langle a,\ main_0\ v\rangle$, where $v \in \Gamma^*$; because $\{intr\} \nsubseteq L_{\mathcal{B}2\mathcal{P}}(\texttt{Wrk})$, r is disabled by the BA transition t as well as the BA state $\texttt{Wrk}$.

Indistinguishability

Given a BA transition $t = q \xrightarrow{\sigma} q' \in \delta$, two LPDS configurations $c, c' \in Conf(\mathcal{P})$ are (respectively, an LPDS rule $r = c \overset{\tau}{\hookrightarrow} c' \in \Delta$ is) *indistinguishable* to t if and only if $\sigma \subseteq L_{\mathcal{P}2\mathcal{B}}(c) \cap L_{\mathcal{P}2\mathcal{B}}(c')$, i.e., t is enabled by both c and c'. On the other hand, given an LPDS rule $r = c \overset{\tau}{\hookrightarrow} c' \in \Delta$, two BA states $q, q' \in Q$ are (respectively, a BA transition $t = q \xrightarrow{\sigma} q' \in \delta$ is) *indistinguishable* to r if and only if $\tau \subseteq L_{\mathcal{B}2\mathcal{P}}(q) \cap L_{\mathcal{B}2\mathcal{P}}(q')$, i.e., r is enabled by both q and q'.

Although the antonym of the word *indistinguishable* should be *distinguishable*, it is not used to describe the negation of indistinguishability. This is because when an LPDS rule r is disabled by both the BA states q and q', it is inappropriate to say that q and q' are distinguishable[1] to r. As an alternative, q and q' are said to be *not indistinguishable* to r.

Example 1. Given the following BA transition, LPDS rule, and instances of the labeling functions:

- a BA transition, $t = \texttt{Intr} \xrightarrow{\{no\_event\}} \texttt{Wrk}$, as illustrated in Figure 4.1;
- an LPDS rule, $r = \langle a,\ main_0 \rangle \overset{\{intr\}}{\hookrightarrow} \langle a,\ isr_0\ main_0 \rangle$, as illustrated in Figure 4.4b;
- $L_{\mathcal{P}2\mathcal{B}}(\langle a,\ main_0 \rangle) = \{no\_event\}$ and $L_{\mathcal{P}2\mathcal{B}}(\langle a,\ isr_0 \rangle) = \{no\_event\}$;
- $L_{\mathcal{B}2\mathcal{P}}(\texttt{Intr}) = \{intr\}$ and $L_{\mathcal{B}2\mathcal{P}}(\texttt{Wrk}) = \{no\_intr\}$.

Because $\{no\_event\} \subseteq L_{\mathcal{P}2\mathcal{B}}(\langle a,\ main_0 \rangle) \cap L_{\mathcal{P}2\mathcal{B}}(\langle a,\ isr_0 \rangle)$, r is indistinguishable to t; because $\{intr\} \nsubseteq L_{\mathcal{B}2\mathcal{P}}(\texttt{Intr}) \cap L_{\mathcal{B}2\mathcal{P}}(\texttt{Wrk})$, t is not indistinguishable to r.

Example 2. Given the following BA transition, LPDS rule, and instances of the labeling functions:

[1] The word indistinguishable is inappropriate neither, because we are interested in the relation between a BA transition and an LPDS rule only when they enable each other.

- a BA transition, $t = \texttt{Wrk} \xrightarrow{\{stop\}} \texttt{Init}$, as illustrated in Figure 4.1;
- an LPDS rule, $r = \langle !a,\ stop_0\rangle \xhookrightarrow{\{no\_intr\}} \langle !a,\ stop_1\rangle$, as illustrated in Figure 4.5b;
- $L_{\mathcal{P}2\mathcal{B}}(\langle !a,\ stop_0\rangle) = \{stop\}$ and $L_{\mathcal{P}2\mathcal{B}}(\langle !a,\ stop_1\rangle) = \{no\_event\}$;
- $L_{\mathcal{B}2\mathcal{P}}(\texttt{Wrk}) = \{no\_intr\}$ and $L_{\mathcal{B}2\mathcal{P}}(\texttt{Init}) = \{no\_intr\}$.

Because $\{stop\} \nsubseteq L_{\mathcal{P}2\mathcal{B}}(\langle !a,\ stop_0\rangle) \cap L_{\mathcal{P}2\mathcal{B}}(\langle !a,\ stop_1\rangle)$, r is not indistinguishable to t; because $\{no\_intr\} \subseteq L_{\mathcal{B}2\mathcal{P}}(\texttt{Wrk}) \cap L_{\mathcal{B}2\mathcal{P}}(\texttt{Init})$, t is indistinguishable to r.

Independence

Given a BA transition $t \in \delta$ and an LPDS rule $r \in \Delta$, if they are indistinguishable to each other, t and r are said to be *independent*; otherwise if either t or r is not indistinguishable to the other but they still enable each other, t and r are said to be *dependent*. The independence relation is symmetric.

Example 1. Given the following BA transition, LPDS rule, and instances of the labeling functions:

- a BA transition, $t = \texttt{Rst} \xrightarrow{\emptyset} \texttt{Wrk}$, as illustrated in Figure 4.1;
- an LPDS rule, $r = \langle a,\ main_3\rangle \xhookrightarrow{\{no\_intr\}} \langle a,\ main_4\rangle$, as illustrated in Figure 4.4b;
- $L_{\mathcal{P}2\mathcal{B}}(\langle a,\ main_3\rangle) = \{no\_event\}$ and $L_{\mathcal{P}2\mathcal{B}}(\langle a,\ main_4\rangle) = \{no\_event\}$;
- $L_{\mathcal{B}2\mathcal{P}}(\texttt{Rst}) = \{no\_intr\}$ and $L_{\mathcal{B}2\mathcal{P}}(\texttt{Wrk}) = \{no\_intr\}$.

Because $\emptyset \subseteq L_{\mathcal{P}2\mathcal{B}}(\langle a,\ main_3\rangle) \cap L_{\mathcal{P}2\mathcal{B}}(\langle a,\ main_4\rangle)$, r is indistinguishable to t; because $\{no\_intr\} \subseteq L_{\mathcal{B}2\mathcal{P}}(\texttt{Rst}) \cap L_{\mathcal{B}2\mathcal{P}}(\texttt{Wrk})$, t is indistinguishable to r. Therefore, t and r are independent.

Example 2. Given the following BA transition, LPDS rule, and instances of the labeling functions:

- a BA transition, $t = \texttt{Wrk} \xrightarrow{\{stop\}} \texttt{Init}$, as illustrated in Figure 4.1;
- an LPDS rule, $r = \langle !a,\ stop_0 \rangle \xhookrightarrow{\{no\_intr\}} \langle !a,\ stop_1 \rangle$, as illustrated in Figure 4.5b;
- $L_{\mathcal{P}2\mathcal{B}}(\langle !a,\ stop_0 \rangle) = \{stop\}$ and $L_{\mathcal{P}2\mathcal{B}}(\langle !a,\ stop_1 \rangle) = \{no\_event\}$;
- $L_{\mathcal{B}2\mathcal{P}}(\texttt{Wrk}) = \{no\_intr\}$ and $L_{\mathcal{B}2\mathcal{P}}(\texttt{Init}) = \{no\_intr\}$.

Because $\{stop\} \not\subseteq L_{\mathcal{P}2\mathcal{B}}(\langle !a,\ stop_0 \rangle) \cap L_{\mathcal{P}2\mathcal{B}}(\langle !a,\ stop_1 \rangle)$, r is not indistinguishable to t; because $\{no\_intr\} \subseteq L_{\mathcal{B}2\mathcal{P}}(\texttt{Wrk})$ and $\{stop\} \subseteq L_{\mathcal{P}2\mathcal{B}}(\langle !a,\ stop_0 \rangle)$, t and r enable each other. Therefore, t and r are dependent.

4.3.2 Büchi Pushdown System (BPDS)

Definition 4.2. Given a BA $\mathcal{B} = (\Sigma, Q, \delta, q_0, F)$ and an LPDS $\mathcal{P} = (I, G, \Gamma, \Delta, \langle g_0, \omega_0 \rangle)$, a Büchi Pushdown System (BPDS), denoted by $\mathcal{BP}$, is the Cartesian product of $\mathcal{B}$ and $\mathcal{P}$. Formally, $\mathcal{BP} = (G \times Q, \Gamma, \Delta', \langle (g_0, q_0), \omega_0 \rangle, F')$, where $G \times Q$, as the product of G and Q, is the set of global states; Γ is the stack alphabet from $\mathcal{P}$; Δ' is constructed by Algorithm 4.1 from δ and Δ; $\langle (g_0, q_0), \omega_0 \rangle$ is the initial configuration; $\langle (g, q), \gamma \rangle \in F'$ if $q \in F$, where $g \in G$ and $\gamma \in \Gamma$.

Algorithm 4.1 does not create a strict Cartesian product of the transition rules from $\mathcal{B}$ and $\mathcal{P}$ respectively. Given a BA transition $t = q \xrightarrow{\sigma} q'$ from δ and an LPDS rule $r = \langle g, \gamma \rangle \xhookrightarrow{\tau} \langle g', \omega \rangle$ from Δ, the algorithm constructs BPDS rules from t and r only if they enable each other. When t and r are dependent, $\mathcal{B}$ and $\mathcal{P}$ must transition together, which is modeled by the BPDS rule $\langle (g, q), \gamma \rangle \hookrightarrow_{\mathcal{BP}} \langle (g', q'), \omega \rangle$. In such situation, the BPDS rule represents synchronous transitions of $\mathcal{B}$ and $\mathcal{P}$. In other words, $\mathcal{B}$ and $\mathcal{P}$ cannot transition in an interleaved manner, because one transition may disable the other. When t and r are independent, $\mathcal{B}$ and $\mathcal{P}$ can transition asynchronously. There are three types of BPDS rules, i.e., $\mathcal{B}$ transitions and $\mathcal{P}$ self-loops as modeled by the BPDS rule $\langle (g, q), \gamma \rangle \hookrightarrow_{\mathcal{BP}} \langle (g, q'), \gamma \rangle$, $\mathcal{P}$ transitions

Algorithm 4.1 BPDSRules($\delta \times \Delta$)

```
$\Delta_{sycn} \leftarrow \emptyset\ \Delta_{hori} \leftarrow \emptyset,\ \Delta_{vert} \leftarrow \emptyset,\ \Delta_{diag} \leftarrow \emptyset$
for all $r = \langle g, \gamma \rangle \overset{\tau}{\hookrightarrow} \langle g', \omega \rangle \in \Delta$ do
    for all $t = q \xrightarrow{\sigma} q' \in \delta$ and $\sigma \subseteq L_{\mathcal{P}2\mathcal{B}}(\langle g, \gamma \rangle)$ and $\tau \subseteq L_{\mathcal{B}2\mathcal{P}}(q)$ do
        if  $r$ and $t$ are dependent then
            {$\mathcal{B}$ and $\mathcal{P}$ must transition together}
            $\Delta_{sync} \leftarrow \Delta_{sync} \bigcup \{\langle (g, q), \gamma \rangle \hookrightarrow_{\mathcal{BP}} \langle (g', q'), \omega \rangle\}$
        else
            {$\mathcal{B}$ transitions and $\mathcal{P}$ self-loops}
            $\Delta_{hori} \leftarrow \Delta_{hori} \bigcup \{\langle (g, q), \gamma \rangle \hookrightarrow_{\mathcal{BP}} \langle (g, q'), \gamma \rangle\}$
            {$\mathcal{P}$ transitions and $\mathcal{B}$ self-loops}
            $\Delta_{vert} \leftarrow \Delta_{vert} \bigcup \{\langle (g, q), \gamma \rangle \hookrightarrow_{\mathcal{BP}} \langle (g', q), \omega \rangle\}$
            {$\mathcal{B}$ and $\mathcal{P}$ transition together}
            $\Delta_{diag} \leftarrow \Delta_{diag} \bigcup \{\langle (g, q), \gamma \rangle \hookrightarrow_{\mathcal{BP}} \langle (g', q'), \omega \rangle\}$
        end if
    end for
end for
$\Delta' \leftarrow \Delta_{sync} \bigcup \Delta_{hori} \bigcup \Delta_{vert} \bigcup \Delta_{diag}$
return  $\Delta'$
```

and $\mathcal{B}$ self-loops as modeled by the BPDS rule $\langle (g, q), \gamma \rangle \hookrightarrow_{\mathcal{BP}} \langle (g', q), \omega \rangle$, and $\mathcal{B}$ and $\mathcal{P}$ transition together as modeled by the BPDS rule $\langle (g, q), \gamma \rangle \hookrightarrow_{\mathcal{BP}} \langle (g', q'), \omega \rangle$. There is one self-loop BA transition, $q \xrightarrow{\sigma} q$, used in the first BPDS rule; and there is one self-loop LPDS rule, $\langle g, \gamma \rangle \overset{\tau}{\hookrightarrow} \langle g, \gamma \rangle$, used in the second BPDS rule. These self-loop transitions/rules may not exist in the original design of $\mathcal{B}$ or $\mathcal{P}$; however, it is necessary to introduce such self-loop transitions/rules so that the asynchronous transitions between $\mathcal{B}$ and $\mathcal{P}$ can be modeled. Let $\mathcal{B}_{loop}$ denote the BA with the self-loop transitions introduced to $\mathcal{B}$ and $\mathcal{P}_{loop}$ denote the LPDS with the self-loop transitions introduced to $\mathcal{P}$. Algorithm 4.1 actually constructs a set of BPDS rules as the Cartesian product of the transition rules from $\mathcal{B}_{loop}$ and $\mathcal{P}_{loop}$ respectively.

The input alphabets Σ and I are used to synchronize the BA transitions and LPDS rules. Once $\mathcal{BP}$ is constructed, these input alphabets are no longer useful; therefore, they are not in the definition of $\mathcal{BP}$. A configuration of $\mathcal{BP}$ is denoted by $\langle (g,q), \omega \rangle \in (G \times Q) \times \Gamma^*$. The set of all configurations is denoted by $Conf(\mathcal{BP})$. The head[2] of a configuration $c = \langle (g,q), \gamma v \rangle$ $(\gamma \in \Gamma, v \in \Gamma^*)$ is $\langle (g,q), \gamma \rangle$ and denoted by $head(c)$. Similarly the head of a rule $r = \langle (g,q), \gamma \rangle \overset{\tau}{\hookrightarrow} \langle (g',q'), \omega \rangle$ is $\langle (g,q), \gamma \rangle$ and denoted by $head(r)$. 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 $\langle (g,q), \gamma v \rangle$. The immediate successor relation in BPDS is written 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 $\mathcal{BP}$, denoted ϕ, is a sequence of BPDS configurations, $c_0 \Rightarrow_{\mathcal{BP}} c_1 \ldots \Rightarrow_{\mathcal{BP}} c_i \Rightarrow_{\mathcal{BP}} \ldots$, where $i \geq 0$, $c_i \in Conf(\mathcal{BP})$, and $\phi(i) = c_i$ denotes the i^{th} configuration on the path. 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.

Definition 4.3. Given a BPDS path ϕ, the Büchi constraint of BPDS requires that if ϕ is infinitely long, it should have infinite many occurrences of BPDS configurations from the set $\{ \ c \mid head(c) \in F' \ \}$.

Given a BPSD path ϕ, it is straightforward to infer that:

- the projection of ϕ on $\mathcal{B}$, denoted by $\phi^{\mathcal{B}}$, is a path of $\mathcal{B}$; and
- the projection of ϕ on $\mathcal{P}$, denoted by $\phi^{\mathcal{P}}$, is a path of $\mathcal{P}$.

[2] Note that a BPDS head can also be considered as a special type of BPDS configuration.

Note that $\phi^{\mathcal{B}}$ does not contain any self-loop BA transition; and $\phi^{\mathcal{P}}$ does not contain any self-loop LPDS transition, as introduced by Algorithm 4.1.

Complexity analysis. For every LPDS rule, Algorithm 4.1 explores all BA transitions to construct the BPDS rules; therefore, it takes $O(|\delta| \times |\Delta|)$ time. We denote the set of BPDS rules constructed by Algorithm 4.1 as $\Delta' = \delta \times \Delta$. Apparently, $|\delta| \times |\Delta| \neq |\delta \times \Delta|$, since Algorithm 4.1 constructs BPDS rules based on the information whether the LPDS rule and BA transition enable each other. Therefore, the algorithm takes $O(|\delta \times \Delta|)$ space to store the new BPDS rules.

Example. A BPDS model $\mathcal{BP} = (G \times Q, \Gamma, \Delta', \langle (g_0, q_0), \omega_0 \rangle, F')$ is constructed using the BA example illustrated in Section 4.1 and the LPDS example illustrated in Section 4.2.2 as follows:

- $G \times Q = \{(\diamond, \texttt{Init}), (\diamond, \texttt{Rst}), (\diamond, \texttt{Wrk}), (\diamond, \texttt{Idle}), (\diamond, \texttt{Intr}), (a, \texttt{Init}),$ $(a, \texttt{Rst}), (a, \texttt{Wrk}), (a, \texttt{Idle}), (a, \texttt{Intr}), (!a, \texttt{Init}), (!a, \texttt{Rst}), (!a, \texttt{Wrk}),$ $(!a, \texttt{Idle}), (!a, \texttt{Intr})\}$;

- $\Gamma = \{\sqcup, main_0, main_1, main_2, main_3, main_4, main_5, reset_0, reset_1, stop_0,$ $stop_1, NonHWRelated_0, NonHWRelated_1, isr_0, isr_1\}$;

- Δ' is constructed according to Algorithm 4.1. Figure 4.6 illustrates an example of constructing the BPDS rules;

- $\langle (g_0, q_0), \omega_0 \rangle = \langle (\diamond, \texttt{Init}), \sqcup \rangle$;

- $F = \{\ \langle (g, \texttt{Init}), \gamma \rangle,\ \langle (g, \texttt{Wrk}), \gamma \rangle,\ \langle (g, \texttt{Idle}), \gamma \rangle\ \}$, where $g \in G$ and $\gamma \in \Gamma$.

Figure 4.6, item 1 demonstrates the case where the LPDS rule and BA transitions are independent and self-loop rules/transitions are introduced to the BA or LPDS in order to model the asynchronous transitions. Item 2 demonstrates the case where LPDS rules and BA transitions are dependent, the BPDS rules should be

1. Given an LPDS rule $\langle a,\ main_1\rangle \xhookrightarrow{\{no\_intr\}} \langle !a,\ main_2\rangle$, for all BA transitions, the following BPDS rules are constructed by Algorithm 4.1:

$$\langle(a,\mathtt{Init}),\ main_1\rangle \hookrightarrow_{\mathcal{BP}} \langle(!a,\mathtt{Init}),\ main_2\rangle \qquad \langle(a,\mathtt{Rst}),\ main_1\rangle \hookrightarrow_{\mathcal{BP}} \langle(!a,\mathtt{Rst}),\ main_2\rangle$$

$$\langle(a,\mathtt{Rst}),\ main_1\rangle \hookrightarrow_{\mathcal{BP}} \langle(!a,\mathtt{Wrk}),\ main_2\rangle \qquad \langle(a,\mathtt{Wrk}),\ main_1\rangle \hookrightarrow_{\mathcal{BP}} \langle(!a,\mathtt{Idle}),\ main_2\rangle$$

$$\langle(a,\mathtt{Wrk}),\ main_1\rangle \hookrightarrow_{\mathcal{BP}} \langle(!a,\mathtt{Intr}),\ main_2\rangle \qquad \langle(a,\mathtt{Idle}),\ main_1\rangle \hookrightarrow_{\mathcal{BP}} \langle(!a,\mathtt{Init}),\ main_2\rangle$$

$$\boldsymbol{\langle(a,\mathtt{Wrk}),\ main_1\rangle \hookrightarrow_{\mathcal{BP}} \langle(!a,\mathtt{Wrk}),\ main_2\rangle} \qquad \boldsymbol{\langle(a,\mathtt{Idle}),\ main_1\rangle \hookrightarrow_{\mathcal{BP}} \langle(!a,\mathtt{Idle}),\ main_2\rangle}$$

$$\boldsymbol{\langle(a,\mathtt{Init}),\ main_1\rangle \hookrightarrow_{\mathcal{BP}} \langle(a,\mathtt{Init}),\ main_1\rangle} \qquad \boldsymbol{\langle(a,\mathtt{Rst}),\ main_1\rangle \hookrightarrow_{\mathcal{BP}} \langle(a,\mathtt{Rst}),\ main_1\rangle}$$

$$\boldsymbol{\langle(a,\mathtt{Rst}),\ main_1\rangle \hookrightarrow_{\mathcal{BP}} \langle(a,\mathtt{Wrk}),\ main_1\rangle} \qquad \boldsymbol{\langle(a,\mathtt{Wrk}),\ main_1\rangle \hookrightarrow_{\mathcal{BP}} \langle(a,\mathtt{Idle}),\ main_1\rangle}$$

$$\boldsymbol{\langle(a,\mathtt{Wrk}),\ main_1\rangle \hookrightarrow_{\mathcal{BP}} \langle(a,\mathtt{Intr}),\ main_1\rangle} \qquad \boldsymbol{\langle(a,\mathtt{Idle}),\ main_1\rangle \hookrightarrow_{\mathcal{BP}} \langle(a,\mathtt{Init}),\ main_1\rangle}$$

2. Given an LPDS rule $\langle\cdot,\ main_1\rangle \xhookrightarrow{\{intr\}} \langle\cdot,\ isr_0\ main_1\rangle$, for all BA transitions, the following BPDS rule is constructed by Algorithm 4.1 (it is actually two rules, since "." represents a or $!a$):

$$\langle(\cdot,\mathtt{Intr}),\ main_1\rangle \hookrightarrow_{\mathcal{BP}} \langle(\cdot,\mathtt{Wrk}),\ isr_0\ main_1\rangle$$

3. Given an LPDS rule $\langle\cdot,\ reset_0\rangle \xhookrightarrow{\{no\_intr\}} \langle\cdot,\ reset_1\rangle$, for all BA transitions, the following BPDS rules are constructed by Algorithm 4.1:

$$\langle(\cdot,\mathtt{Init}),\ reset_0\rangle \hookrightarrow_{\mathcal{BP}} \langle(\cdot,\mathtt{Rst}),\ reset_1\rangle \qquad \langle(\cdot,\mathtt{Wrk}),\ reset_0\rangle \hookrightarrow_{\mathcal{BP}} \langle(\cdot,\mathtt{Rst}),\ reset_1\rangle$$

$$\langle(\cdot,\mathtt{Rst}),\ reset_0\rangle \hookrightarrow_{\mathcal{BP}} \langle(\cdot,\mathtt{Rst}),\ reset_1\rangle \qquad \langle(\cdot,\mathtt{Rst}),\ reset_0\rangle \hookrightarrow_{\mathcal{BP}} \langle(\cdot,\mathtt{Wrk}),\ reset_1\rangle$$

$$\boldsymbol{\langle(\cdot,\mathtt{Rst}),\ reset_0\rangle \hookrightarrow_{\mathcal{BP}} \langle(\cdot,\mathtt{Rst}),\ reset_0\rangle} \qquad \boldsymbol{\langle(\cdot,\mathtt{Rst}),\ reset_0\rangle \hookrightarrow_{\mathcal{BP}} \langle(\cdot,\mathtt{Wrk}),\ reset_0\rangle}$$

Figure 4.6: Example of constructing BPDS rules. The bold rules are constructed by introducing self-loop transitions to $\mathcal{B}$ or $\mathcal{P}$.

constructed in such a way that can represent the synchronous transitions between the BA and LPDS. For item 3, since the LPDS rule $\langle \cdot,\ reset_0\rangle \xhookrightarrow{\{no\_intr\}} \langle \cdot,\ reset_1\rangle$ is dependent with the BA transitions $\texttt{Init} \xrightarrow{\{reset\}} \texttt{Rst}$ and $\texttt{Wrk} \xrightarrow{\{reset\}} \texttt{Rst}$, the first two BPDS rules are constructed to represent the synchronous transitions between the BA and LPDS. On the other hand, since the LPDS rule is independent with the BA transitions $\texttt{Rst} \xrightarrow{\emptyset} \texttt{Rst}$ and $\texttt{Rst} \xrightarrow{\emptyset} \texttt{Wrk}$, the rest four BPDS rules are constructed to represent the asynchronous transitions.

4.3.3 BPDS Loop Constraint

According to the BPDS rules shown in Figure 4.6, there exists an infinite path $\phi = \langle (a, \texttt{Rst}),\ main_1\rangle \Rightarrow_{\mathcal{BP}} \langle (a, \texttt{Wrk}),\ main_1\rangle \Rightarrow_{\mathcal{BP}} \langle (a, \texttt{Idle}),\ main_1\rangle \Rightarrow_{\mathcal{BP}} \langle (a, \texttt{Init}), main_1\rangle \Rightarrow_{\mathcal{BP}} \langle (a, \texttt{Init}),\ main_1\rangle \ldots$. Because the BA $\mathcal{B}$ and LPDS $\mathcal{P}$ can transition asynchronously at the BPDS configurations on ϕ, it is possible that $\mathcal{B}$ keeps on moving forward while $\mathcal{P}$ self-loops. However, after reaching the configuration $\langle (a, \texttt{Init}),\ main_1\rangle$, the BPDS $\mathcal{BP}$ starts to self-loop forever, where the self-loop transition follows the BPDS rule $\langle (a, \texttt{Init}),\ main_1\rangle \hookrightarrow_{\mathcal{BP}} \langle (a, \texttt{Init}),\ main_1\rangle$. This BPDS rule is constructed from the BA transition $\texttt{Init} \xrightarrow{\{no\_event\}} \texttt{Init}$ and the LPDS rule $\langle a,\ main_1\rangle \xhookrightarrow{no\_intr} \langle a,\ main_1\rangle$. The BA transition belongs to the original design of $\mathcal{B}$, which indicates that $\mathcal{B}$ can self-loop at the state $\texttt{Init}$. However, the LPDS rule is a self-loop rule introduced to $\mathcal{P}$ in order to model the asynchronous transitions between $\mathcal{B}$ and $\mathcal{P}$. This type of self-loop rule should not affect the fairness of other rules that belong to the original design. In other words, since $\mathcal{P}$ always self-loops at the configuration $\langle (a, \texttt{Init}),\ main_1\rangle$, no transition that belongs to the original design of $\mathcal{P}$ occurs on the path ϕ. Therefore, ϕ is not a path of $\mathcal{BP}$ and should be ruled out.

Definition 4.4. BPDS loop constraint. Given any infinite BPDS path ϕ, it must satisfy the requirements such that $\phi^{\mathcal{B}}$ and $\phi^{\mathcal{P}}$ are also infinite. In other words, the BA (respectively, LPDS) transitions that are not self-loop transitions introduced to Δ_{vert} (respectively, Δ_{hori}) in Algorithm 4.1 should occur infinitely often on ϕ.

The BPDS loop constraint requires that the transitions from δ or Δ repeatedly occur on all BPDS paths. It implies the fairness between $\mathcal{B}$ and $\mathcal{P}$, because either $\mathcal{B}$ or $\mathcal{P}$ needs to repeatedly transition to satisfy the constraint. However, $\mathcal{B}$ and/or $\mathcal{P}$ can still self-loop infinitely if their designs contain self-loop transitions. For example, since the design of $\mathcal{B}$ contains a self-loop transition $\texttt{Init} \xrightarrow{\{no\_event\}} \texttt{Init}$, $\mathcal{B}$ can always self-loop at the state `Init`, which is not affected by the BPDS loop constraint.

4.4 SYMBOLIC REPRESENTATIONS

Algorithm 4.1 needs to explore every BA transition and every LPDS rule in order to construct BPDS rules. The cost of the algorithm mainly depends the number of BA transitions and LPDS rules, i.e., $|\delta| \times |\Delta|$. Furthermore, as we shall demonstrate in Chapter 5, the complexity of the algorithms for BPDS depends highly on how well the transition rules can be represented. Therefore, it is desired that both δ and Δ are represented in a compact way so that analyzing algorithms for BPDS can be carried out more efficiently.

4.4.1 Symbolic representation of BA

Given a BA $\mathcal{B} = (\Sigma, Q, \delta, q_0, F)$, the set of transitions δ can be divided into several subsets such that $\delta = \delta_1 \cup \delta_2 \cup \ldots \cup \delta_n$, where

- $0 < n \leq |\delta|$;
- $\forall i \in [1, n], \exists \sigma \in \Sigma, \exists \tau \in I$ such that $\delta_i = \{t | t = q \xrightarrow{\sigma} q' \in \delta, L_{\mathcal{B}2\mathcal{P}}(q) = \tau\}$.

This condition requires that the BA transitions in each subset not only have the same input symbol but also enable the same LPDS rules. Therefore, a subset δ_i

can be understood as a rule that describes a set of BA transitions. Normally, these subsets are referred to as *symbolic BA transition rules*. This type of representation is referred to as a *symbolic representation*. Given a BA state q and a symbolic BA transition rule δ_i, $\delta_i(q) = q'$ represents the transition $q \xrightarrow{\sigma} q'$; while $\delta_i(q) = \varepsilon$ indicates that δ_i is not applicable to q. Since a symbolic BA transition rule describes BA transitions labeled by the same input symbol, a label on the rule is unnecessary.

Example. Consider the BA example illustrated in Figure 4.1, the set of transitions δ can be written as $\delta = \delta_1 \cup \delta_2 \cup \delta_3 \cup \delta_4 \cup \delta_5$, where

- $\delta_1 = Q \times \{\{reset\}\} \times Q = \{t_1, t_2\}$,
  $L_{\mathcal{B}2\mathcal{P}}(\texttt{Init}) = L_{\mathcal{B}2\mathcal{P}}(\texttt{Wrk}) = \{no\_intr\}$;

- $\delta_2 = Q \times \{\{stop\}\} \times Q = \{t_3\}$,
  $L_{\mathcal{B}2\mathcal{P}}(\texttt{Wrk}) = \{no\_intr\}$;

- $\delta_3 = Q \times \{\emptyset\} \times Q = \{t_4, t_5\}$,
  $L_{\mathcal{B}2\mathcal{P}}(\texttt{Rst}) = \{no\_intr\}$;

- $\delta_4 = Q \times \{\{no\_event\}\} \times Q = \{t_6, t_7, t_8, t_9\}$,
  $L_{\mathcal{B}2\mathcal{P}}(\texttt{Init}) = L_{\mathcal{B}2\mathcal{P}}(\texttt{Wrk}) = L_{\mathcal{B}2\mathcal{P}}(\texttt{Idle}) = \{no\_intr\}$;

- $\delta_5 = Q \times \{\{no\_event\}\} \times Q = \{t_{10}\}$,
  $L_{\mathcal{B}2\mathcal{P}}(\texttt{Intr}) = \{intr\}$.

Since the number of the symbolic BA transition rules, $n = 5$, is less than the number of BA transitions, $|\delta| = 10$, Algorithm 4.1 operates more efficiently on the symbolic representation compared to the explicit representation. The rest of the problem is how to describe these symbolic BA transition rules.

For various purposes, symbolic transition rules can be described at different levels of abstraction. For example, a Binary Decision Diagram (BDD) [16, 53] is a data structure commonly used to represent transition relations such as δ_1, δ_2, etc. Because BDDs are compact and the logical operations on BDDs are efficient, model checking algorithms usually utilize BDD representations for their target models. Although BDD representations are efficient for automatic analysis, it is hard to read or construct them manually.

Symbolic transition rules also can be described at a higher abstraction level without BDD representations. Figure 4.7 illustrates an example of the symbolic BA transition rules specified in the modelC language. The set of BA states Q is described by the global variable Q. The five symbolic BA transition rules, δ_1, δ_2, δ_3, δ_4, and δ_5 are described by five hardware transaction functions (see definition in Chapter 3), `delta1`, `delta2`, `delta3`, `delta4`, and `delta5` respectively. Consider $\rho \subseteq Q$ as the current-states of BA, i.e., the BA states when entering a hardware transaction function `delta`i (where $1 \leq i \leq 5$), $\rho' \subseteq Q$ as the next-states of BA, i.e., the BA states when exiting `delta`i, and $\sigma \in \Sigma$ as the label on δ_i, `delta`i specifies the set of BA transitions in δ_i, i.e., $\rho \times \{\sigma\} \times \rho'$. Each hardware transaction function updates the value of Q based on Q's current value. A symbolic BA transition rule may not be applicable to certain states. For example, δ_1 should not be applied to states other than `Init` or `Wrk`. The keyword `halt` signifies that if the current state of Q is not applicable for a rule, the state transition based on this rule should halt. Non-deterministic transitions are specified using the non-deterministic function `choice`. For example, `delta3` specifies two non-deterministic transitions from the state `Rst` to the states `Rst` and `Wrk` respectively.

4.4.2 Symbolic representation of LPDS

The control flow of a program refers to the order in which individual statements, instructions, or function calls are evaluated; and data flow refers to how the value of program variables should be updated along with the execution. The symbolic representation of LPDS is based on the control flow and data flow. Given an LPDS

```
enum { Init, Rst, Wrk, Idle, Intr } Q;

// $\delta_1$, labeled by {reset}
__atomic void delta1() {
    switch( Q ) {
        case Init: Q = Rst; break;
        case Wrk: Q = Rst; break;
        // halt: current state is not applicable to
        // delta1, since delta1 can only be applied
        // to the states Init and Wrk
        default: halt;
    }
}

// $\delta_3$, labeled by $\emptyset$
__atomic void delta3() {
    switch( Q ) {
        case Rst:
            // model the non-deterministic
            // transitions from Rst to
            // Rst and Wrk respectively
            if ( choice() ) Q = Rst;
            else Q = Wrk;
            break;
        default: halt;
    }
}
```

```
// $\delta_2$, labeled by {stop}
__atomic void delta2() {
    switch( Q ) {
        case Wrk: Q = Init; break;
        default: halt;
    }
}

// $\delta_4$, labeled by {no_event}
__atomic void delta4() {
    switch( Q ) {
        case Wrk:
            if ( choice() ) Q = Idle;
            else Q = Intr;
            break;
        case Idle: Q = Init; break;
        case Init: Q = Init; break;
        default: halt;
    }
}

// $\delta_5$, labeled by {no_event}
__atomic void delta5() {
    switch( Q ) {
        case Intr: Q = Wrk; break;
        default: halt;
    }
}
```

Figure 4.7: Symbolic BA transition rules specified in modelC.

$\mathcal{P} = (I, G, \Gamma, \Delta, \langle g_0, \omega_0 \rangle)$, the global states can be written as $G = G_c \times G_d$ and the stack alphabet can be written as $\Gamma = \Gamma_c \times \Gamma_d$. A *symbolic LPDS rule* is written as follows:

$$\langle g, \gamma \rangle \underset{R}{\overset{\tau}{\hookrightarrow}} \langle g', \gamma_1 \ldots \gamma_n \rangle,$$

where $g, g' \in G_c$, $\gamma, \gamma_1, \ldots, \gamma_n \in \Gamma_c$, and $R \subseteq (G_d \times \Gamma_d) \times (G_d \times \Gamma_d^n)$. Conceptually, symbols from G_c and Γ_c describe the control flow of LPDS and symbols from G_d and Γ_d describe the data flow of LPDS. A symbolic LPDS rule describes a set of LPDS rules that are labeled by the same input symbol and have the same state transition with respect to the control flow.

There are three motivations for separating the representation of control flow and data flow. First, many properties focus on control flow instead of data flow in software verification. For example, SLIC properties focus on the order of how functions are entered or exited in the execution of a program. Second, the input alphabet of BA is defined on the control flow of LPDS. For example, in Figure 4.2, the propositional variable *reset* in Σ is evaluated as true when the stack location $reset_0$ is reached, which represents a software interface event that attempts to reset hardware. Third, the separated representation can help the analysis of LPDS, which will be utilized in Chapter 5 to optimize the model checking algorithm for BPDS.

Similar to the symbolic representation of BA, the symbolic representation of LPDS exists on different level of abstractions. BDDs are an efficient representation for implementing model checking algorithms. However, a program can also be considered as a symbolic representation of LPDS at a higher level of abstraction, where each atomic statement of the program corresponding to a symbolic rule of LPDS.

4.4.3 Symbolic representation of BPDS

A symbolic LPDS rule describes multiple LPDS rules that may not necessarily have the same head; a symbolic transition rule of BA describes multiple BA transitions that may not necessarily have the same current-state, i.e., the state on the left side of a BA transition. Therefore, the labeling functions should be modified to support the symbolic representations as follows:

- $L'_{\mathcal{P}2\mathcal{B}}: (G_c \times \Gamma_c) \rightarrow \Sigma$, which associates the control flow location of LPDS with the set of propositions that hold on it;
- $L'_{\mathcal{B}2\mathcal{P}}: 2^{\delta} \rightarrow I$, which associates a symbolic rule of BA with the set of propositions that hold on it.

$L'_{\mathcal{P}2\mathcal{B}}$ is equivalent to $L_{\mathcal{P}2\mathcal{B}}$, since the input alphabet of BA is defined on the control flow of LPDS. $L'_{\mathcal{B}2\mathcal{P}}$ is equivalent to $L_{\mathcal{B}2\mathcal{P}}$, since for all BA transitions described by the same symbolic transition rule, their current-states should have the same label by $L_{\mathcal{B}2\mathcal{P}}$. After the labeling functions are defined for symbolic representations of BA and LPDS, the indistinguishability relation and independence relation can be defined for symbolic representations based on $L'_{\mathcal{P}2\mathcal{B}}$ and $L'_{\mathcal{B}2\mathcal{P}}$.

Let $\mathcal{BP} = (G \times Q, \Gamma, \Delta', \langle (g_0, q_0), \omega_0 \rangle, F')$ be a BPDS. The symbolic representation of $\mathcal{BP}$ is constructed from the symbolic representations of $\mathcal{B}$ and $\mathcal{P}$. A symbolic BPDS rule, $\langle g, \gamma \rangle \underset{R'}{\hookrightarrow}{}_{\mathcal{BP}} \langle g', \omega \rangle \subseteq \Delta'$, has two parts. First, the state transition regarding the control flow of $\mathcal{P}$ is explicitly specified. Second, the transition relation $R' \subseteq (Q \times G_d \times \Gamma_d) \times (Q \times G_d \times \Gamma_d^{|\omega|})$, associated with the symbolic BPDS rule, describes a set of state transitions regarding both the states of $\mathcal{B}$ and the data flow of $\mathcal{P}$.

In order to model the asynchronous transitions between BA and LPDS, Algorithm 4.1 introduces self-loop transitions to BA and LPDS respectively. For symbolic representations, these self-loop transitions should be introduced to BA and LPDS as well. Therefore, we need to define two self-loop transition relations:

- $R_{loop} \subseteq (G_d \times \Gamma_d) \times (G_d \times \Gamma_d^*)$ and $\forall \langle g_d, \gamma_d \rangle \in (G_d \times \Gamma_d), R_{loop}(\langle g_d, \gamma_d \rangle) = \langle g_d, \gamma_d \rangle$, i.e., R_{loop} is a self-loop transition relation for data flow of LPDS;
- $U_{loop} \subseteq Q \times \Sigma \times Q$ and $\forall q \in Q, U_{loop}(q) = q$, i.e., U_{loop} is a self-loop transition relation for BA.

Given the set of symbolic BA transition rules δ and the set of symbolic LPDS rules Δ, Algorithm 4.2 constructs the set of symbolic BPDS rules Δ'. Let $\mathcal{R} \subseteq \Delta$ and $U \subseteq \delta$. If $\mathcal{R}$ and U are dependent, $\mathcal{B}$ and $\mathcal{P}$ must transition together; therefore,

Algorithm 4.2 SymbolicBPDSRules($\delta \times \Delta$)

```
$\Delta_{sycn} \leftarrow \emptyset$ $\Delta_{hori} \leftarrow \emptyset$, $\Delta_{vert} \leftarrow \emptyset$, $\Delta_{diag} \leftarrow \emptyset$
**for all** $\mathcal{R} = \langle g, \gamma \rangle \overset{\tau}{\underset{R}{\hookrightarrow}} \langle g', \omega \rangle \subseteq \Delta$ **do**
  **for all** $U = Q \times \{\sigma\} \times Q \subseteq \delta$ **and** $\sigma \subseteq L'_{\mathcal{P}2\mathcal{B}}(\langle g, \gamma \rangle)$ **and** $\tau \subseteq L'_{\mathcal{B}2\mathcal{P}}(U)$ **do**
    **if** $\mathcal{R}$ and $U$ are dependent **then**
      {*$\mathcal{B}$ and $\mathcal{P}$ must transition together*}
      $\Delta_{sync} \leftarrow \Delta_{sync} \bigcup \langle g, \gamma \rangle \underset{R'}{\hookrightarrow}{}_{\mathcal{BP}} \langle g', \omega \rangle$, where $R' = R \times U$
    **else**
      {*$\mathcal{B}$ transitions and $\mathcal{P}$ self-loops*}
      $\Delta_{hori} \leftarrow \Delta_{hori} \bigcup \langle g, \gamma \rangle \underset{R'}{\hookrightarrow}{}_{\mathcal{BP}} \langle g, \gamma \rangle$, where $R' = R_{loop} \times U$
      {*$\mathcal{P}$ transitions and $\mathcal{B}$ self-loops*}
      $\Delta_{vert} \leftarrow \Delta_{vert} \bigcup \langle g, \gamma \rangle \underset{R'}{\hookrightarrow}{}_{\mathcal{BP}} \langle g', \omega \rangle$, where $R' = R \times U_{loop}$
      {*$\mathcal{B}$ and $\mathcal{P}$ transition together*}
      $\Delta_{diag} \leftarrow \Delta_{diag} \bigcup \langle g, \gamma \rangle \underset{R'}{\hookrightarrow}{}_{\mathcal{BP}} \langle g', \omega \rangle$, where $R' = R \times U$
    **end if**
  **end for**
**end for**
$\Delta' \leftarrow \Delta_{sync} \bigcup \Delta_{hori} \bigcup \Delta_{vert} \bigcup \Delta_{diag}$
**return** $\Delta'$
```

a symbolic BPDS rule $\langle g, \gamma\rangle \underset{R'}{\hookrightarrow}{}_{\mathcal{BP}}\langle g', \omega\rangle$ is constructed, where R' is the Cartesian product of U and R (the transition relation associated with $\mathcal{R}$). On the other hand, if $\mathcal{R}$ and U are independent, $\mathcal{B}$ and $\mathcal{P}$ can transition asynchronously; therefore, Algorithm 4.2 introduces self-loop transitions to $\mathcal{B}$ or $\mathcal{P}$. When a symbolic BPDS rule, $\langle g, \gamma\rangle \underset{R'}{\hookrightarrow}{}_{\mathcal{BP}}\langle g, \gamma\rangle$, is constructed to represent the situation that $\mathcal{B}$ transitions and $\mathcal{P}$ self-loops, the transition relation R' is constructed as the Cartesian product of U and the self-loop transition relation R_{loop}. When a symbolic BPDS rule, $\langle g, \gamma\rangle \underset{R'}{\hookrightarrow}{}_{\mathcal{BP}}\langle g', \omega\rangle$, is constructed to represent the situation that $\mathcal{P}$ transitions and $\mathcal{B}$ self-loops, the transition relation R' is constructed as the Cartesian product of the self-loop transition relation U_{loop} and R.

Chapter 5

CO-VERIFICATION ALGORITHM

Formal verification has three basic elements. First, we need a model for abstracting the target system. A system design usually has implementation details in various aspects, e.g., design logic, power consumption, and physical layout. A model captures only the interested aspects to be verified. As discussed in Chapter 4, the Büchi Pushdown System (BPDS) is the unifying model in our co-verification framework. Second, we need a property which predicates the possible observations that can be made of the model. A property simply has two values. If the observation of the model agrees with the predication, the property is said to be true on the model. Otherwise, the property is said to be false on the model. Third, we need a methodology to prove or disprove a property on the model. Model checking is an automatic process that explores the state space of the model to verify the property.

We commonly verify two types of properties in model checking. *Safety properties* are understood as "bad events never happen." For example, a computer system should never crash. *Liveness properties* are understood as "good events will eventually happen." For example, a program should eventually terminate. Due to their differences, safety properties and liveness properties are usually treated differently in verification. We will discuss the model checking algorithms for both safety properties and liveness properties on BPDS.

Since a BPDS $\mathcal{BP}$ is the Cartesian product of a BA $\mathcal{B}$ and an LPDS $\mathcal{P}$, the size of the $\mathcal{BP}$ is very sensitive to how the Cartesian product is constructed. Algorithm 4.1 constructs BPDS rules from the BA transitions and LPDS rules in a straightforward way. However, since $\mathcal{B}$ and $\mathcal{P}$ are mostly asynchronous, some of their transition orders do not affect the properties to be verified. If model checking can avoid exploring these transition orders, the verification cost should be greatly reduced. Therefore, we can improve Algorithm 4.1 so that the unnecessary transition orders between $\mathcal{B}$ and $\mathcal{P}$ will not be explored in model checking. Our approach follows the idea of static partial order reduction, where the reduction is applied at compile time before model checking. As a result, no modification is required in the model checking algorithm.

As discussed in Chapter 4, explicit representations of the BA and LPDS are inefficient in practice. In contrast, we have two benefits using symbolic representations. First, the target systems are specified using programming languages such as C, SystemC, modelC, etc., in co-verification. It is inefficient to translate these programs into their explicit representations in order to construct a BPDS model. Indeed, we can treat these programs as a symbolic representation of a BA or LPDS and apply our reduction algorithms directly to the programs. Second, it is neither necessary nor efficient to build a complete state graph (even if it is possible) for a BPDS during model checking. Symbolic model checking algorithms encode the transition relation of a BPDS implicitly using data structures such as BDDs. This approach can greatly alleviate the state explosion problem in model checking. Obviously, the two benefits of symbolic representations exist on different levels of abstractions; however the symbolic algorithms are described in a similar manner. In this dissertation, we will elaborate on the first type of symbolic representation, where the algorithms work directly on the programs in order to construct reduced BPDS models.

5.1 MODEL CHECKING PROBLEMS OF BPDS

Given a property φ to be verified on a model M, if we consider φ as some kind of state machine, the model checking problem of φ on M can be generally expressed as a process of checking whether the language of M is contained by the language of φ, $\mathcal{L}(M) \subseteq \mathcal{L}(\varphi)$. There are two types of properties:

- *Safety properties* refer to those critical requirements that ensure bad events never happen. For example, SLIC rules (see Chapter 2) are all safety properties. The question of whether or not a safety property holds on a model is equivalent to the problem of whether or not the model has a state that violates the property and is reachable from the model's initial state. The second problem can be solved via *reachability analysis*.

- *Liveness properties* refer to those functional requirements that ensure good events will eventually happen. For example, the LTL formula (see Chapter 2), $\boldsymbol{F}\ a$, states that the propositional variable a should eventually become true at some time point starting from the model's initial state. This verification problem can be solved as two sub-problems. First, does there exist a path from the initial state to a state on which a is evaluated as true? Second, if the path exists, does there exist any loop on the path such that the event, a becoming true, may be delayed infinitely? The first sub-problem can be solved via reachability analysis and the second sub-problem can be solved via loop computation.

In our co-verification framework, the formal model is a BPDS and the properties are specified using SLIC rules or LTL formulae. When a property is specified using a SLIC rule, we can solve the model checking problem through reachability analysis. When a property is specified using an LTL formula, we need to construct a Büchi automaton to represent the negation of the formula. The model checking problem is to compute whether the Büchi automaton has an accepting run on a trace of BPDS. Given a BPDS $\mathcal{BP} = (G \times Q, \Gamma, \Delta', \langle (g_0, q_0), \omega_0 \rangle, F')$, the following subsections discuss the reachability analysis problem and the LTL checking problem of $\mathcal{BP}$ respectively.

5.1.1 Reachability Analysis

A safety property can be specified as a label on a configuration of $\mathcal{BP}$ or a finite state machine that monitors the state transition of $\mathcal{BP}$. In both ways, verification of the safety property can be solved via reachability analysis. For the first type of safety property, the problem is to solve whether the labeled BPDS configuration is reachable from the initial configuration. For the second type of safety property, we can instrument $\mathcal{BP}$ using the property. The resulting transition system is still considered as a BPDS, since the state machine of the property only monitors the state transition of $\mathcal{BP}$. The model checking process verifies whether a labeled violation state in the new BPDS (due to the instrumentation) is reachable from the initial configuration.

Definition 5.1. Given a BPDS configuration $c \in Conf(\mathcal{BP})$, reachability analysis computes if c is reachable from the initial configuration c_0, i.e., $c_0 \Rightarrow^*_{\mathcal{BP}} c$.

Reachability analysis is applied in two ways: *pre** and *post**. Given a set of BPDS configurations $C \subseteq Conf(\mathcal{BP})$,

- *pre*$^*(C) = \{\, p \mid c \in C$ and $p \Rightarrow^*_{\mathcal{BP}} c \,\}$, i.e., the backward reachability analysis computes the predecessors of elements in C;
- *post*$^*(C) = \{\, p \mid c \in C$ and $c \Rightarrow^*_{\mathcal{BP}} p \,\}$, i.e., the forward reachability analysis computes the successors of elements in C.

The reachability analysis problem of BPDS can be solved via the backward reachability analysis algorithm by computing whether $c_0 \in$ *pre*$^*(\{c\})$ or via the forward reachability analysis algorithm by computing whether $c \in$ *post*$^*(\{c_0\})$.

5.1.2 LTL Checking

Given an LTL formula φ and a BPDS $\mathcal{BP}$, we want to compute if $\mathcal{L}(\mathcal{BP}) \subseteq \mathcal{L}(\varphi)$, i.e., the language of $\mathcal{BP}$ is contained by the language of φ. Vardi and Wolper [86] have introduced an automata-theoretic approach for LTL checking on finite state systems. Schwoon [77] has demonstrated an automata-theoretic approach to model checking PDS, an infinite state system. There are two observations. First, a PDS is finitely representable. Second, there exists a labeling function that can map from a possibly infinite state space of PDS to a finite set of symbols. These two observations are true of BPDS as well.

Let $At(\varphi)$ be the set of atomic propositions in φ. There exists a labeling function,

$$L_{\varphi} : Conf(\mathcal{BP}) \rightarrow 2^{At(\varphi)},$$

i.e., L_{φ} maps a BPDS configuration to a set of propositions that are true of it. Although $Conf(\mathcal{BP})$ is an infinite set, the heads of $\mathcal{BP}$ belongs to a finite set, i.e., $(G \times Q) \times \Gamma$. Therefore, L_{φ} can be built in such a way that it only looks into the head of a BPDS configuration to decide the output symbol.

According to the idea of automata-theoretic approach, the two conditions, $\mathcal{L}(\mathcal{BP}) \subseteq \mathcal{L}(\varphi)$ and $\mathcal{L}(\mathcal{BP}) \cap \mathcal{L}(\neg\varphi) = \emptyset$, are equivalent. Furthermore, the second condition can be checked via computing the accepting run of the Büchi automaton constructed from $\neg\varphi$. Since there always exists a BA $\mathcal{B}_{\varphi} = (2^{At(\varphi)}, Q_{\varphi}, \delta_{\varphi}, q_{\varphi 0}, F_{\varphi})$ that accepts the language $\mathcal{L}(\neg\varphi)$, we can synthesize a transition system $\mathcal{B}^2\mathcal{P}$ from $\mathcal{BP}$ and $\mathcal{B}_{\varphi}$ using the labeling function L_{φ}, where conceptually $\mathcal{B}_{\varphi}$ monitors the state transitions of $\mathcal{BP}$.

Formally, $\mathcal{B}^2\mathcal{P} = (G \times Q \times Q_{\varphi},\, \Gamma,\, \Delta_{\mathcal{B}^2\mathcal{P}},\, \langle (g_0, q_0, q_{\varphi 0}), \omega_0\rangle, F_{\mathcal{B}^2\mathcal{P}})$, where

- $G \times Q \times Q_{\varphi}$ is the finite set of global states,
- Γ is the stack alphabet,
- $\Delta_{\mathcal{B}^2\mathcal{P}}$ is the finite set of transition rules,
- $\langle (g_0, q_0, q_{\varphi 0}), \omega_0\rangle$ is the initial configuration, and

- $F_{\mathcal{B}^2\mathcal{P}} = F' \times F_\varphi$ specifies the accepting states of $\mathcal{B}^2\mathcal{P}$.

The transition relation $\Delta_{\mathcal{B}^2\mathcal{P}}$ is constructed in such a way that we add $(c, q_\varphi) \hookrightarrow_{\mathcal{B}^2\mathcal{P}} (c', q'_\varphi)$ to $\Delta_{\mathcal{B}^2\mathcal{P}}$ if and only if

- $c \hookrightarrow_{\mathcal{BP}} c' \in \Delta'$,
- $q_\varphi \xrightarrow{\sigma} q'_\varphi \in \delta_\varphi$, and
- $\sigma \subseteq L_\varphi(c)$.

The set of all configurations is denoted by $Conf(\mathcal{B}^2\mathcal{P}) \subseteq G \times Q \times Q_\varphi \times \Gamma^*$. For the purpose of simplicity, we also write

$$\mathcal{B}^2\mathcal{P} = (P, \Gamma, \Delta_{\mathcal{B}^2\mathcal{P}}, F_{\mathcal{B}^2\mathcal{P}}),$$

where $P = G \times Q \times Q_\varphi$. Given $\gamma \in \Gamma$ and $v \in \Gamma^*$, the head of a configuration $c = \langle p, \gamma v \rangle$ is $\langle p, \gamma \rangle$ and denoted by *head*(c); and the head of a rule $r = \langle p, \gamma \rangle \hookrightarrow_{\mathcal{B}^2\mathcal{P}} \langle p', \omega \rangle$ is $\langle p, \gamma \rangle$ and denoted by *head*(r).

Given a rule $r = \langle p, \gamma \rangle \hookrightarrow_{\mathcal{B}^2\mathcal{P}} \langle p', \omega \rangle \in \Delta_{\mathcal{B}^2\mathcal{P}}$, for every $v \in \Gamma^*$ the configuration $\langle p, \gamma v \rangle$ is an immediate predecessor of $\langle p', \omega v \rangle$, and $\langle p', \omega v \rangle$ is an immediate successor of $\langle p, \gamma v \rangle$. The immediate successor relation of $\mathcal{B}^2\mathcal{P}$ is written as $\langle p, \gamma v \rangle \Rightarrow_{\mathcal{B}^2\mathcal{P}} \langle p', \omega v \rangle$, where we say this state transition *follows* the rule r. The reachability relation, $\Rightarrow^*_{\mathcal{B}^2\mathcal{P}}$, is the reflexive and transitive closure of the immediate successor relation.

A path of $\mathcal{B}^2\mathcal{P}$, denoted by ϕ, is a sequence of configurations, $c_0 \Rightarrow_{\mathcal{B}^2\mathcal{P}} c_1 \ldots \Rightarrow_{\mathcal{B}^2\mathcal{P}} c_i \Rightarrow_{\mathcal{B}^2\mathcal{P}} \ldots$, where $i \geq 0$, $c_i \in Conf(\mathcal{B}^2\mathcal{P})$, and $\phi(i) = c_i$ denotes the i^{th} configuration on the path. The path is also referred to as a trace of $\mathcal{B}^2\mathcal{P}$ if c_0 is the initial configuration. An infinite path of $\mathcal{B}^2\mathcal{P}$ needs to satisfy both the Büchi constraint and the BPDS loop constraint, which are defined for BPDS paths in Chapter 4. It is straightforward to infer that:

- the projection of ϕ on $\mathcal{B}_\varphi$, denoted by $\phi^{\mathcal{B}_\varphi}$, is a path of $\mathcal{B}_\varphi$;

- the projection of ϕ on $\mathcal{B}$, denoted by $\phi^{\mathcal{B}}$, is a path of $\mathcal{B}$; and
- the projection of ϕ on $\mathcal{P}$, denoted by $\phi^{\mathcal{P}}$, is a path of $\mathcal{P}$.

Definition 5.2. An accepting run of $\mathcal{B}^2\mathcal{P}$ is an infinite trace ϕ such that (1) ϕ has infinitely many occurrences of configurations from the set $\{ c \mid \mathit{head}(c) \in F_{\mathcal{B}^2\mathcal{P}} \}$, i.e., the Büchi acceptance condition is satisfied; and (2) both $\phi^{\mathcal{B}}$ and $\phi^{\mathcal{P}}$ are also infinite, i.e., the BPDS loop constraint is satisfied.

Definition 5.3. Given a BPDS $\mathcal{BP}$ and an LTL formula φ, the model checking problem is to compute if the $\mathcal{B}^2\mathcal{P}$ model constructed from $\mathcal{BP}$ and φ has an accepting run.

5.2 REACHABILITY ANALYSIS ALGORITHM

Given a BPDS $\mathcal{BP} = (G \times Q, \Gamma, \Delta', \langle (g_0, q_0), \omega_0 \rangle, F')$ and a configuration $c \in Conf(\mathcal{BP})$, reachability analysis computes whether $c_0 \Rightarrow^*_{\mathcal{BP}} c$. Obviously, the shortest path ϕ that demonstrates $c_0 \Rightarrow^*_{\mathcal{BP}} c$ should have a finite length, because the set of global states $G \times Q$ and the stack alphabet Γ are both finite. Therefore, the Büchi constraint F' is not relevant to reachability analysis. Furthermore, since ϕ is finite, the BPDS loop constraint is not applicable to ϕ either.

A $\mathcal{BP}$ without the Büchi constraint F' has a set of global states, a stack alphabet, a set of transition rules, and an initial configuration. Intuitively, if we can convert $\mathcal{BP}$ to a PDS model, the reachability problem of $\mathcal{BP}$ can be solved on PDS using existing algorithms [6, 50, 77]. Figure 5.1 illustrates the idea. The reachability analysis algorithm first converts a $\mathcal{BP}$ to a PDS $\mathcal{P}'$ and then checks the reachability property using model checking algorithms for PDS models. We refer to $\mathcal{P}'$ as the verification model for $\mathcal{BP}$. Compared to the LPDS $\mathcal{P}$ used to construct $\mathcal{BP}$, it is important to note that $\mathcal{P}'$ is a standard PDS in the sense that $\mathcal{P}'$ does not have inputs.

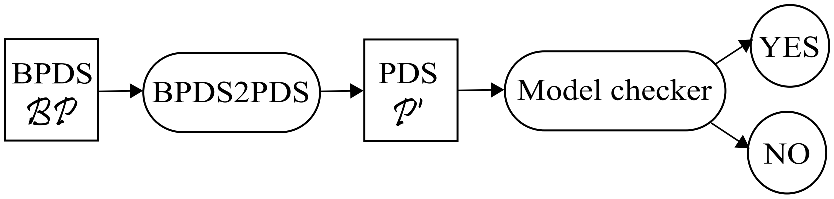


Figure 5.1: Reachability analysis of BPDS.

Now, we present a straightforward conversion algorithm from $\mathcal{BP}$ to $\mathcal{P}'$, namely BPDS2PDS. Given $\mathcal{BP}$, we can construct $\mathcal{P}' = (G_{\mathcal{P}'}, \Gamma_{\mathcal{P}'}, \Delta_{\mathcal{P}'}, c_0)$ such that

- $G_{\mathcal{P}'} = (G \times Q)$,
- $\Gamma_{\mathcal{P}'} = \Gamma$,
- $\Delta_{\mathcal{P}'} = \{ \langle (g,q), \gamma \rangle \hookrightarrow \langle (g',q'), \omega \rangle \mid \langle (g,q), \gamma \rangle \hookrightarrow_{\mathcal{BP}} \langle (g',q'), \omega \rangle \in \Delta' \}$, and
- $c_0 = \langle (g_0, q_0), \omega_0 \rangle$.

Theorem 5.1. *$\mathcal{P}'$ preserves the reachability property of $\mathcal{BP}$.*

Proof. We use the idea of proof by construction. According to how $\mathcal{P}'$ is constructed, it is straightforward that: (1) the state space of $\mathcal{P}'$ equals to that of $\mathcal{BP}$; (2) $\mathcal{P}'$ preserves all the transition rules of $\mathcal{BP}$; (3) the initial state of $\mathcal{P}'$ is the initial state of $\mathcal{BP}$; and (4) the Büchi constraint F' in $\mathcal{BP}$ is not preserved in $\mathcal{P}'$, since F' is not used for reachability checking. □

A PDS rule of $\mathcal{P}'$ represents the transitions of both the BA and LPDS by simply treating the BA transition as a part of the global state transition and keeping the LPDS transition in its original form. When representing software using PDS, such a transition rule, $\langle g, \gamma \rangle \hookrightarrow \langle g', \gamma \rangle$ where $g \neq g'$, is uncommon, since the stack control location needs to be updated when a global state is modified. However, this kind of rule is allowed by PDS and can be used to represent a BPDS transition where BA executes asynchronously with LPDS.

Complexity analysis. In theory, the conversion algorithm, BPDS2PDS, needs to go through every BPDS rule of $\mathcal{BP}$ and constructs a corresponding PDS rule for $\mathcal{P}'$. This is unnecessary in practice, since the BPDS rules and PDS rules are different only in concept, but indeed they can be stored using the same data structure. BPDS2PDS should be considered as an abstract algorithm that helps us understand the reachability analysis of BPDS. However, the algorithm does not require an actual implementation.

With respect to the reachability analysis of PDS-equivalent models, the Bebop [6] model checker computes the reachability status of a Boolean program statement (from the entry statement of the `main` procedure) with both the time and space complexity of $O(E \times 2^k)$, where E is the number of edges in the interprocedural control-flow graph of the Boolean program and k is the maximal number of variables in scope at any program location. Compared to the PDS representation, the edges in the interprocedural control-flow graph correspond to the transition rules, and the number of the states of the visible Boolean variables has an upper bound by the number of the heads of $\mathcal{P}'$. Therefore, we have $E \in O(|\Delta_{\mathcal{P}'}|)$ and $2^k \in O(|G_{\mathcal{P}'} \times \Gamma_{\mathcal{P}'}|)$.

The Moped model checker [77] provides the reachability analysis of PDS in two ways: pre^* and $post^*$. Let c_0 be the initial configuration of $\mathcal{P}'$. Given a configuration $c \in Conf(\mathcal{P}')$, the backward reachability analysis algorithm, pre^*, computes whether $c_0 \in pre^*(\{c\})$ using $O(|G_{\mathcal{P}'}|^2 \times |\Delta_{\mathcal{P}'}|)$ in time and $O(|G_{\mathcal{P}'}| \times |\Delta_{\mathcal{P}'}|)$ in space; while the forward reachability analysis algorithm, $post^*$, computes whether $c \in post^*(\{c_0\})$ using $O((|G_{\mathcal{P}'}| + |\Delta_{\mathcal{P}'}|)^3)$ in both time and space.

5.3 LTL CHECKING ALGORITHM

According to Definition 5.2 and Definition 5.3, the LTL checking of $\mathcal{BP}$ has two steps. First, we need to compute some kind of loops in the state graph of $\mathcal{B}^2\mathcal{P}$, where each loop should satisfy both the Büchi acceptance condition and the BPDS loop constraint. Second, we need to check if the loops are reachable from the initial configuration of $\mathcal{B}^2\mathcal{P}$.

Definition 5.4. In order to identify those loops that satisfy our requirement, we define a binary relation $\boldsymbol{\Rightarrow^r_{\mathcal{B}^2\mathcal{P}}}$ between two configurations of $\mathcal{B}^2\mathcal{P}$. Given two configurations c and c', we write $c \Rightarrow^r_{\mathcal{B}^2\mathcal{P}} c'$ if and only if $\exists\langle p, \gamma\rangle \in F_{\mathcal{B}^2\mathcal{P}}$ such that $c \Rightarrow^*_{\mathcal{B}^2\mathcal{P}} \langle p, \gamma\omega\rangle \Rightarrow^+_{\mathcal{B}^2\mathcal{P}} c'$, where $\omega \in \Gamma^*$. A head $\langle p, \gamma\rangle$ is *repeating* if $\exists v \in \Gamma^*$ for some $\langle p, \gamma\rangle \Rightarrow^r_{\mathcal{B}^2\mathcal{P}} \langle p, \gamma v\rangle$. The set of repeating heads is denoted by $Rep(\mathcal{B}^2\mathcal{P})$. We refer to the path that demonstrates a repeating head as a *repeating path.*

Based on Definition 5.4, given a repeating head $\langle p, \gamma\rangle$ and $v, \omega \in \Gamma^*$, there must exist a path that transitions from $\langle p, \gamma v\rangle$ to $\langle p, \gamma\omega v\rangle$ without touching the stack content v. In other words, no transition on the path pops the stack content v. There may be more than one repeating path for a repeating head.

Proposition 5.1. *Given the initial configuration c_0, $\mathcal{B}^2\mathcal{P}$ has an accepting run if and only if (1) $\exists c_0 \Rightarrow^*_{\mathcal{B}^2\mathcal{P}} c'$ such that head$(c') \in Rep(\mathcal{B}^2\mathcal{P})$; and (2) a repeating path ϕ_s of head(c') satisfies the condition that $|\phi_s^{\mathcal{B}}| \neq 0$ and $|\phi_s^{\mathcal{P}}| \neq 0$.*

Proof. "$\Rightarrow$": Let ϕ be an accepting run of $\mathcal{B}^2\mathcal{P}$. We know that ϕ has an infinite length. Since the set of heads in $\mathcal{B}^2\mathcal{P}$ is finite, there must be at least one head $\langle p, \gamma\rangle \in P \times \Gamma$ that occurs on ϕ infinitely often. Furthermore, we can always find a sequence of positive integers $0 = i_0 < i_1 < i_2 < \ldots$ such that

$$\phi_s \quad = \quad \langle p, \gamma v\rangle \Rightarrow^*_{\mathcal{B}^2\mathcal{P}} \langle p', \gamma'\omega' v\rangle \Rightarrow^+_{\mathcal{B}^2\mathcal{P}} \langle p, \gamma\omega v\rangle \quad = \quad \langle p, \gamma v\rangle \Rightarrow^r_{\mathcal{B}^2\mathcal{P}} \langle p, \gamma\omega v\rangle,$$

where

1. $\phi(i_k) = \langle p, \gamma v\rangle$, $\phi(i_{k+1}) = \langle p, \gamma\omega v\rangle$, $k \geq 0$, $v, \omega, \omega' \in \Gamma^*$;
2. $\langle p', \gamma'\rangle \in F_{\mathcal{B}^2\mathcal{P}}$; and
3. $|\phi_s^{\mathcal{B}}| \neq 0$ and $|\phi_s^{\mathcal{P}}| \neq 0$, since a run of $\mathcal{B}^2\mathcal{P}$ must satisfy the BPDS loop constraint.

Therefore, we have proven this direction of the proposition.

"$\Leftarrow$": Let $\langle p, \gamma\rangle = \mathit{head}(c') \in \mathit{Rep}(\mathcal{B}^2\mathcal{P})$. Based on the hypothesis we have

$$\phi_s \quad = \quad (c' : \langle p, \gamma v\rangle) \Rightarrow^r_{\mathcal{B}^2\mathcal{P}} \langle p, \gamma\omega v\rangle,$$

where $v, \omega \in \Gamma^*$, $|\phi_s^{\mathcal{B}}| \neq 0$, and $|\phi_s^{\mathcal{P}}| \neq 0$. Using ϕ_s, we can construct an infinite trace such that

$$\phi \quad = \quad c_0 \Rightarrow^*_{\mathcal{B}^2\mathcal{P}} (c' : \langle p, \gamma v\rangle) \Rightarrow^r_{\mathcal{B}^2\mathcal{P}} \langle p, \gamma\omega v\rangle \Rightarrow^r_{\mathcal{B}^2\mathcal{P}} \langle p, \gamma\omega\omega v\rangle \Rightarrow^r_{\mathcal{B}^2\mathcal{P}} \ldots.$$

Because paths in the form of ϕ_s repeatedly occurs on ϕ, both the acceptance condition and BPDS loop constraint are satisfied on ϕ. Therefore, ϕ is an accepting run of $\mathcal{B}^2\mathcal{P}$. □

Proposition 5.1 presents a practical strategy for us to check LTL properties on BPDS. As illustrated in Figure 5.2, there are two phases. First, we need to compute a special set of repeating heads, $R \subseteq \mathit{Rep}(\mathcal{B}^2\mathcal{P})$, where the repeating paths of the heads satisfy the BPDS loop constraint. Second, we need to check if there exists a path of $\mathcal{B}^2\mathcal{P}$ that leads from the initial configuration c_0 to a configuration c such that $\mathit{head}(c) \in R$.

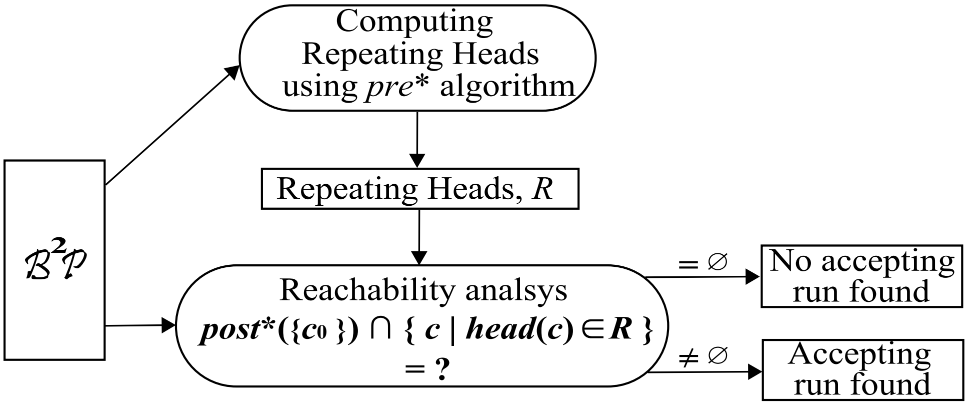


Figure 5.2: Computing the accepting run of $\mathcal{B}^2\mathcal{P}$ (c_0 is the initial configuration).

5.3.1 Computing the Repeating Heads

As an important observation, when computing the repeating heads, we are looking for a path between $\langle p, \gamma \rangle$ and $\langle p, \gamma v \rangle$. However, the actual content of v is not interested. Therefore, we can compute the repeating heads solely based on the information about which heads are reachable from each other and whether the accepting requirements (i.e., the Büchi acceptance condition and the BPDS loop constraint) on the paths between these heads are satisfied. Such kind of information can be encoded into a finite graph and the repeating heads can be computed by detecting the strongly connected components that satisfy the accepting requirements.

Before constructing the graph, we need to define the notions about how an edge of the graph can satisfy the accepting requirements. Therefore, we define three labeling functions on the rules of $\mathcal{B}^2\mathcal{P}$:

1. $F_{\mathcal{B}^2\mathcal{P}} : \Delta_{\mathcal{B}^2\mathcal{P}} \rightarrow \{0, 1\}$, where given $r \in \Delta_{\mathcal{B}^2\mathcal{P}}$, $F_{\mathcal{B}^2\mathcal{P}}(r) = 1$ if $\mathit{head}(r) \in F_{\mathcal{B}^2\mathcal{P}}$ and $F_{\mathcal{B}^2\mathcal{P}}(r) = 0$ if otherwise;

2. $R_{\mathcal{B}}(r) : \Delta_{\mathcal{B}^2\mathcal{P}} \rightarrow \{0, 1\}$, where given $r \in \Delta_{\mathcal{B}^2\mathcal{P}}$, $R_{\mathcal{B}}(r) = 1$ if r is constructed using a BA transition from δ (defined for checking the BPDS loop constraint) and $R_{\mathcal{B}}(r) = 0$ if otherwise;

3. $R_{\mathcal{P}}(r) : \Delta_{\mathcal{B}^2\mathcal{P}} \rightarrow \{0, 1\}$, where given $r \in \Delta_{\mathcal{B}^2\mathcal{P}}$, $R_{\mathcal{P}}(r) = 1$ if r is constructed using an LPDS rule from Δ (defined for checking the BPDS loop constraint) and $R_{\mathcal{P}}(r) = 0$ if otherwise.

Definition 5.5. The *head reachability graph* of $\mathcal{B}^2\mathcal{P}$ is a directed labeled graph $\mathcal{G} = ((P \times \Gamma), E)$, where the set of nodes are the heads of $\mathcal{B}^2\mathcal{P}$, the set of edges $E \subseteq (P \times \Gamma) \times \{0, 1\}^3 \times (P \times \Gamma)$ denotes the reachability relation between the heads. Let $p, p', p'' \in P$, $\gamma, \gamma' \in \Gamma$, $v_1, v_2 \in \Gamma^*$, and ε be the empty string. An edge $\boldsymbol{(\langle p, \gamma \rangle, (b_1, b_2, b_3), \langle p', \gamma' \rangle)}$ belongs to E under the following conditions:

- $\exists r = \langle p, \gamma \rangle \hookrightarrow_{\mathcal{B}^2\mathcal{P}} \langle p'', v_1\gamma' v_2 \rangle$;
- $\exists \phi = \langle p'', v_1 \rangle \Rightarrow^*_{\mathcal{B}^2\mathcal{P}} \langle p', \varepsilon \rangle$;
- $b_1 = 1$, if and only if $F_{\mathcal{B}^2\mathcal{P}}(r) = 1$ or $\langle p'', v_1 \rangle \Rightarrow^r_{\mathcal{B}^2\mathcal{P}} \langle p', \varepsilon \rangle$;
- $b_2 = 1$, if and only if $R_{\mathcal{B}}(r) = 1$ or $|\phi^{\mathcal{B}}| \neq 0$;
- $b_3 = 1$, if and only if $R_{\mathcal{P}}(r) = 1$ or $|\phi^{\mathcal{P}}| \neq 0$.

Definition 5.5 is based on the idea of backward reachability computation. Given the head $\langle p', \varepsilon \rangle$ reachable from $\langle p'', v_1 \rangle$, if there exits a rule to indicate that $\langle p'', v_1\gamma' \rangle$ is reachable from $\langle p, \gamma \rangle$, then we know that the head $\langle p', \gamma' \rangle$ (a.k.a., $\langle p', \varepsilon\gamma' \rangle$) is reachable from the head $\langle p, \gamma \rangle$. During such a computation process, we use the three labels, b_1, b_2, and b_3 to record the information whether a path between the heads contains an accepting state in $F_{\mathcal{B}^2\mathcal{P}}$ and satisfies the BPDS loop constraint.

The set of repeating heads, R, can be computed by exploiting the fact that a head $\langle p, \gamma \rangle$ is repeating and the repeating path satisfies the BPDS loop constraint if and only if

- $\langle p, \gamma \rangle$ is part of a strongly connected component of $\mathcal{G}$; and
- this strongly connected component has internal edges labeled by $(1, *, *)$, $(*, 1, *)$, and $(*, *, 1)$, where $*$ represents 0 or 1.

Algorithm 5.1, RepHeads, takes $\mathcal{B}^2\mathcal{P}$ as input in order to compute the set of repeating heads, R. The algorithm has two parts. First, it computes the head reachability graph of $\mathcal{B}^2\mathcal{P}$ using three steps as follows:

1. Between line 4 and line 6, it constructs edges of the head reachability graph from $\Delta_{\mathcal{B}^2\mathcal{P}}$. We refer to such edges as *direct reachability* edges, because reachability between the heads are satisfied through only one transition.

2. At line 8, it invokes the algorithm, HeadReachability, to compute the *indirect reachability relation* between heads, i.e., reachability through more than one transitions. As illustrated in Algorithm 5.2, HeadReachability (see below for discussion) utilizes the backward reachability analysis algorithm pre^* presented in [77] to compute a set of labeled transition rules (see Definition 5.5), Δ_{label}, that describes the indirect reachability relation between heads.

3. Between line 10 and line 12, it constructs edges of the head reachability graph based on Δ_{label}. We refer to these edges as *indirect reachability* edges.

Second, between line 15 and line 21, Algorithm 5.1 computes strongly connected components of the head reachability graph $\mathcal{G}$ and checks whether there exists strongly connected components that satisfy the accepting requirements. If a strongly connected component satisfies the accepting requirements, all the heads on it are added to the set R.

Algorithm 5.2, HeadReachability, computes a set of labeled transition rules Δ_{label}, that describes the indirect reachability relation between heads. The algorithm utilizes the pre^* algorithm [77]. Given $\Delta_{\mathcal{B}^2\mathcal{P}}$, pre^* finds a special set of rules $trans \subseteq \Delta_{\mathcal{B}^2\mathcal{P}}$ such that $trans$ has rules all in the form of $\langle p, \gamma \rangle \hookrightarrow_{\mathcal{B}^2\mathcal{P}} \langle p', \varepsilon \rangle$, also written as (p, γ, p') for simplicity. With the three labels defined on BPDS rules, we can further write a rule in $trans$ as $(p, [\gamma, b_1, b_2, b_3], p')$. Given such a rule, the algorithm between line 7 and 25 computes the reachability relation between heads, where rel stores the processed rules from $trans$. Specifically,

- At line 11 or line 14, when we see a rule $\langle p_1, \gamma_1 \rangle \hookrightarrow_{\mathcal{B}^2\mathcal{P}} \langle p, \gamma \rangle$, we know $\langle p', \varepsilon \rangle$ is reachable from $\langle p_1, \gamma_1 \rangle$; therefore, we add a new rule $\langle p_1, \gamma_1 \rangle \hookrightarrow_{\mathcal{B}^2\mathcal{P}} \langle p', \varepsilon \rangle$ to $trans$;

- At line 17, when we see a rule $\langle p_1, \gamma_1 \rangle \hookrightarrow_{\mathcal{B}^2\mathcal{P}} \langle p, \gamma\gamma_2 \rangle$, we know $\langle p', \gamma_2 \rangle$ is

Algorithm 5.1 RepHeads($\mathcal{B}^2\mathcal{P} = (P, \Gamma, \Delta_{\mathcal{B}^2\mathcal{P}}, F_{\mathcal{B}^2\mathcal{P}})$)

```
$R \leftarrow \emptyset$, $E \leftarrow \emptyset$
{*First, compute the set of edges, $E$, of the head reachability graph from $\mathcal{B}^2\mathcal{P}$*}
{*Direct reachability between two heads, i.e., indicated by a rule of $\mathcal{B}^2\mathcal{P}$*}
**for all** $r = \langle p, \gamma \rangle \hookrightarrow_{\mathcal{B}^2\mathcal{P}} \langle p', \gamma' v \rangle \in \Delta_{\mathcal{B}^2\mathcal{P}}$, where $v \in \Gamma^*$ **do**
    $E \leftarrow E \bigcup \{(\langle p, \gamma \rangle, (F_{\mathcal{B}^2\mathcal{P}}(r), R_{\mathcal{B}}(r), R_{\mathcal{P}}(r)), \langle p', \gamma' \rangle)\}$
**end for**
{*Compute the indirect reachability relation between heads, see Algorithm 5.2.*}
$\Delta_{label} \leftarrow$ HeadReachability($\Delta_{\mathcal{B}^2\mathcal{P}}$)
{*Indirect reachability between two heads, computed by* HeadReachability}
**for all** $\langle p, \gamma \rangle \overset{l}{\hookrightarrow}_{\mathcal{B}^2\mathcal{P}} \langle p', \gamma' \rangle \in \Delta_{label}$ **do**
    $E \leftarrow E \bigcup \{(\langle p, \gamma \rangle, l, \langle p', \gamma' \rangle)\}$
**end for**

{*Second, find $R$ in $\mathcal{G}$*}
Find strongly connected components, $SCC$, in $\mathcal{G} = ((P \times \Gamma), E)$
**for all** $C \in SCC$ **do**
    **if** $C$ has internal edges labeled by $(1, *, *)$, $(*, 1, *)$, and $(*, *, 1)$, where $*$ represents 0 or 1 **then**
        {*$C$ contains a set of repeating heads whose repeating paths satisfy the BPDS loop constraint*}
        $R \leftarrow R \bigcup \{$the heads in $C\}$
    **end if**
**end for**
**return** $R$
```

Algorithm 5.2 HEADREACHABILITY($\Delta_{\mathcal{B}^2\mathcal{P}}$)

```
$\Delta_{label} \leftarrow \emptyset$, $rel \leftarrow \emptyset$, $trans \leftarrow \emptyset$
{Compute the head reachability graph of $\mathcal{B}^2\mathcal{P}$ using the $pre^*$ algorithm}
for all $r = \langle p, \gamma \rangle \hookrightarrow_{\mathcal{B}^2\mathcal{P}} \langle p', \varepsilon \rangle \in \Delta_{\mathcal{B}^2\mathcal{P}}$ do
    {Add the labeled rule $r$ (written in a simplified form) to $trans$}
    $trans \leftarrow trans \bigcup \{\ (p,\ [\gamma, F_{\mathcal{B}^2\mathcal{P}}(r), R_{\mathcal{B}}(r), R_{\mathcal{P}}(r)],\ p')\ \}$
end for
while $trans \neq \emptyset$ do
    pop $t = (p, [\gamma, b_1, b_2, b_3], p')$ from $trans$;
    if $t \notin rel$ then
        $rel \leftarrow rel \bigcup \{t\}$;
        for all $r = \langle p_1, \gamma_1 \rangle \hookrightarrow_{\mathcal{B}^2\mathcal{P}} \langle p, \gamma \rangle \in \Delta_{\mathcal{B}^2\mathcal{P}}$ do
            $trans \leftarrow trans \bigcup \{\ (p_1,\ [\gamma_1, b_1 \bigvee F_{\mathcal{B}^2\mathcal{P}}(r), b_2 \bigvee R_{\mathcal{B}}(r), b_3 \bigvee R_{\mathcal{P}}(r)],\ p')\ \}$
        end for
        for all $\langle p_1, \gamma_1 \rangle \overset{l}{\hookrightarrow}_{\mathcal{B}^2\mathcal{P}} \langle p, \gamma \rangle \in \Delta_{label}$, $l = (b'_1, b'_2, b'_3)$ do
            $trans \leftarrow trans \bigcup \{\ (p_1,\ [\gamma_1, b_1 \bigvee b'_1, b_2 \bigvee b'_2, b_3 \bigvee b'_3],\ p')\ \}$
        end for
        for all $r = \langle p_1, \gamma_1 \rangle \hookrightarrow_{\mathcal{B}^2\mathcal{P}} \langle p, \gamma\gamma_2 \rangle \in \Delta_{\mathcal{B}^2\mathcal{P}}$ do
            $\Delta_{label} \leftarrow \Delta_{label} \bigcup \{\langle p_1, \gamma_1 \rangle \overset{l}{\hookrightarrow}_{\mathcal{B}^2\mathcal{P}} \langle p', \gamma_2 \rangle\}$, where
                        $l = (b_1 \bigvee F_{\mathcal{B}^2\mathcal{P}}(r), b_2 \bigvee R_{\mathcal{B}}(r), b_3 \bigvee R_{\mathcal{P}}(r))$
            {Match the new rule with the rules that have been processed}
            for all $(p', [\gamma_2, b'_1, b'_2, b'_3], p'') \in rel$ do
                $trans \leftarrow trans \bigcup \{\ (p_1,\ [\gamma_1, b_1 \bigvee b'_1 \bigvee F_{\mathcal{B}^2\mathcal{P}}(r),$
                                    $b_2 \bigvee b'_2 \bigvee R_{\mathcal{B}}(r), b_3 \bigvee b'_3 \bigvee R_{\mathcal{P}}(r)],\ p'')\ \}$
            end for
        end for
    end if
end while
return $\Delta_{label}$
```

reachable from $\langle p_1, \gamma_1\rangle$; therefore, we add a new rule $\langle p_1, \gamma_1\rangle \hookrightarrow_{\mathcal{B}^2\mathcal{P}} \langle p', \gamma_2\rangle$ to Δ_{label}.

- Between line 20 and line 22, since there is a new rule $\langle p_1, \gamma_1\rangle \hookrightarrow_{\mathcal{B}^2\mathcal{P}} \langle p', \gamma_2\rangle$ generated, we need to go through the set *rel* in order to check if the new rule can be combined with any processed rules. If there is a rule $\langle p', \gamma_2\rangle \hookrightarrow_{\mathcal{B}^2\mathcal{P}} \langle p'', \varepsilon\rangle$ in *rel*, a new rule $\langle p_1, \gamma_1\rangle \hookrightarrow_{\mathcal{B}^2\mathcal{P}} \langle p'', \varepsilon\rangle$ should be added to *trans*.

During this process, we also use the labels to record the information whether the Büchi acceptance condition and the BPDS loop constraint can be satisfied by repeating the path between two heads.

Complexity analysis. Algorithm 5.2 is actually a pre^* algorithm which takes $O(|P|^2|\Delta_{\mathcal{B}^2\mathcal{P}}|)$ time and $O(|P||\Delta_{\mathcal{B}^2\mathcal{P}}|)$ space [77]. In Algorithm 5.1, the first part generates the head reachability graph $\mathcal{G}$ which takes $O(|P|^2|\Delta_{\mathcal{B}^2\mathcal{P}}|)$ time and $O(|P||\Delta_{\mathcal{B}^2\mathcal{P}}|)$ space by invoking Algorithm 5.2. The second part computes strongly connected components in $\mathcal{G}$ which is a linear time computation with respect to the size of $\mathcal{G}$. The rules of $\Delta_{\mathcal{B}^2\mathcal{P}}$ contribute $O(|\Delta_{\mathcal{B}^2\mathcal{P}}|)$ nodes and edges to the size of $\mathcal{G}$. Since the size of Δ_{label} is $O(|P||\Delta_{\mathcal{B}^2\mathcal{P}}|)$, the total size of $\mathcal{G}$ is $O(|P||\Delta_{\mathcal{B}^2\mathcal{P}}|)$. Obviously, the first part of the algorithm dominates the complexity; therefore Algorithm 5.1 takes $O(|P|^2|\Delta_{\mathcal{B}^2\mathcal{P}}|)$ time and $O(|P||\Delta_{\mathcal{B}^2\mathcal{P}}|)$ space.

5.3.2 Computing the Reachability of Repeating Heads

After R is computed, we need to decide whether $post^*(\{c_0\}) \bigcap \{c|head(c) \in R\} = \emptyset$, i.e., given the initial configuration c_0, if there exits $c_0 \Rightarrow^* c$ for some $head(c) \in R$. Similar to the reachability analysis algorithm discussed in Section 5.2, a $\mathcal{B}^2\mathcal{P}$ model can also be converted into a PDS model for reachability analysis, where the complexity of the conversion is $O(|\mathcal{B}^2\mathcal{P}|)$.

The forward reachability algorithms, $post^*$, for PDS-equivalent models have been well studied. We utilize Schwoon's algorithm [77] in our LTL checking of

BPDS, where the complexity of the algorithm is $O((|P| + |\Delta_{\mathcal{B}^2\mathcal{P}}|)^3)$.

5.3.3 Summary

Given a BPDS $\mathcal{BP}$ and an LTL property φ, we can construct a transition system $\mathcal{B}^2\mathcal{P}$ as the Cartesian product of $\mathcal{BP}$ and a BA $\mathcal{B}_\varphi$ that recognizes φ. The model checking problem is then reduced to computing if $\mathcal{B}^2\mathcal{P}$ has an accepting run. There are two parts in computing the accepting run of $\mathcal{B}^2\mathcal{P}$. First, we need to detect a special set of repeating heads in $\mathcal{B}^2\mathcal{P}$ such that their repeating paths can help satisfy the BPDS loop constraint. Algorithm 5.1 solves this problem using $O(|P|^2|\Delta_{\mathcal{B}^2\mathcal{P}}|)$ time and $O(|P||\Delta_{\mathcal{B}^2\mathcal{P}}|)$ space. Second, we need to check if a repeating head is reachable from the initial configuration of $\mathcal{B}^2\mathcal{P}$. This problem can be solved using $O((|P| + |\Delta_{\mathcal{B}^2\mathcal{P}}|)^3)$ time and space. In conclusion, the LTL model checking of BPDS has the complexity of $O((|P| + |\Delta_{\mathcal{B}^2\mathcal{P}}|)^3)$.

5.4 OPTIMIZATION OF REACHABILITY ANALYSIS

5.4.1 Reduction Algorithm

As discussed in Chapter 4, a BPDS $\mathcal{BP}$ is constructed from a BA $\mathcal{B}$ and an LPDS $\mathcal{P}$ using a Cartesian product. It is naïve to verify such a BPDS model, since we may not need all the information from a model to prove a specific type of property. Instead, it is a common practice to automatically prune the model according to the property to be verified. For example, the set of BPDS rules is the product of δ that belongs to $\mathcal{B}$ and Δ that belongs to $\mathcal{P}$ in the naïve approach. However, with respect to reachability analysis, a complete product is unnecessary when $\mathcal{B}$ and $\mathcal{P}$ are asynchronous (i.e., when the BA transitions and LPDS rules are independent), since their transition orders usually do not matter. Without affecting the verification result, static partial order reduction can be applied to reduce the BPDS rules generated by the product. The reduced BPDS model $\mathcal{BP}_r$

will have a smaller set of transition rules $\Delta'_r \subseteq \Delta'$ and fewer state transition traces while still preserving the reachability properties of $\mathcal{BP}$. Figure 5.3 illustrates the verification process that supports the reduction. When constructing the BPDS $\mathcal{BP}_r$ from $\mathcal{B}$ and $\mathcal{P}$, static partial order reduction is applied to reduce the BPDS rules that are generated. Since there are fewer BPDS rules to be explored in verification, the reachability analysis is more efficient with reduction than that of the naïve approach.

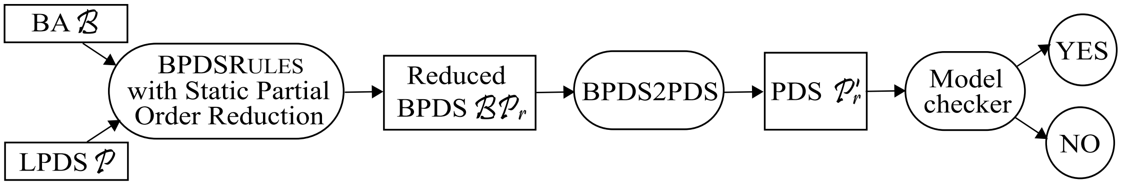
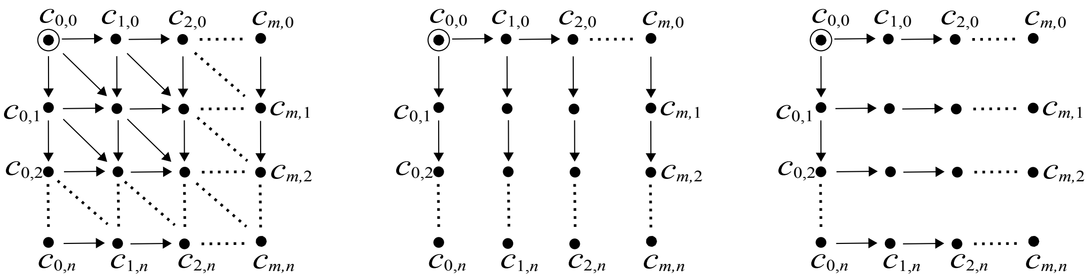


Figure 5.3: Reachability analysis of BPDS with static partial order reduction.

Our reduction [49] is based on the observation that when $\mathcal{B}$ and $\mathcal{P}$ transition asynchronously, one can run continuously while the other one loops. Figure 5.4 illustrates the idea of reducing a BPDS state transition graph that starts from the configuration $c_{0,0}$. Figure 5.4a is a complete state transition graph. There are three types of transition edges:

- a horizontal edge represents a transition when $\mathcal{B}$ transitions and $\mathcal{P}$ self-loops, which follows a BPDS rule in the form of $\langle (g, q), \gamma\rangle \hookrightarrow_{\mathcal{BP}} \langle (g, q'), \gamma\rangle$;
- a vertical edge represents a transition when $\mathcal{P}$ transitions and $\mathcal{B}$ self-loops, which follows a BPDS rule in the form of $\langle (g, q), \gamma\rangle \hookrightarrow_{\mathcal{BP}} \langle (g', q), w\rangle$; and
- 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 BA transition $t = q \xrightarrow{\sigma} q'$ and the LPDS rule $r = \langle g, \gamma\rangle \overset{\tau}{\hookrightarrow} \langle g', \omega\rangle$ that are



(a) Complete transition graph (b) Reduce hori./diag. edges (c) Reduce vert./diag. edges

Figure 5.4: An example of static partial order reduction on BPDS transitions. State transition edges are reduced without affecting the reachability from $c_{0,0}$ when BA and LPDS are asynchronous.

both enabled on $c_{i,j}$, if t and r are independent, we can reduce a large set of state transitions in Figure 5.4a without affecting the reachability from $c_{0,0}$ to other configurations in the graph. Figure 5.4b and Figure 5.4c illustrate two types of static partial order reductions that reduce horizontal/diagonal transition edges and vertical/diagonal transition edges respectively. The reduction can significantly reduce the transition rules of $\mathcal{BP}$, when BA transitions and LPDS rules are independent.

Now we present an optimization of Algorithm 4.1, where the reduction is applied during the rule generation of the BPDS model $\mathcal{BP}_r$. In the reduction process, we need to identify those situations when BPDS rules can be reduced. Since the reduction is applied only if the transitions of $\mathcal{B}$ and $\mathcal{P}$ are independent, a straightforward approach needs to maintain all independent BA transitions and LPDS rules as the reducible candidates. However, such an approach is inefficient. Because $\mathcal{B}$ and $\mathcal{P}$ are asynchronous in most of their transitions, there are many independent BA transitions and LPDS rules. Therefore, we try to identify the situations when BA transitions and LPDS rules are dependent so that we know what BPDS rules cannot be reduced instead of what BPDS can be reduced. Note that both reduction approaches should have the same effect. We define a set of LPDS heads, $SensitiveSet$, on $Conf(\mathcal{P})$ as follows:

Definition 5.6. $SensitiveSet = \{\ head(\langle g_0, \omega_0\rangle)\ \} \bigcup \{\ head(c') \mid \exists r = c \overset{\tau}{\hookrightarrow} c' \in \Delta, \exists t \in \delta,$ r and t 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 [33]. However, instead of identifying transitions that are unnecessary to be executed (i.e., reducible) at a state, $SensitiveSet$ identifies transitions that should be preserved (i.e., irreducible).

Algorithm 5.3 applies the reduction following the idea illustrated in Figure 5.4b, where the horizontal/diagonal edges are reduced.

- At line 6, since the LPDS rule r and the BA transition t are dependent, $\mathcal{B}$ and $\mathcal{P}$ must transition together; therefore, we construct a BPDS rule $\langle(g, q), \gamma\rangle \hookrightarrow_{\mathcal{BP}} \langle(g', q'), \omega\rangle$;

- At line 9, we construct a vertical rule $\langle(g, q), \gamma\rangle \hookrightarrow_{\mathcal{BP}} \langle(g', q), \omega\rangle$ to represent the asynchronous situation when $\mathcal{P}$ transitions and $\mathcal{B}$ self-loops. Since Algorithm 5.3 follows the reduction demonstrated in Figure 5.4b, all vertical BPDS rules are preserved;

- At line 12, we construct a horizontal rule $\langle(g, q), \gamma\rangle \hookrightarrow_{\mathcal{BP}} \langle(g, q'), \gamma\rangle$ to represent the asynchronous situation when $\mathcal{B}$ transitions and $\mathcal{P}$ self-loops, if and only if $head(r)$ belongs to $SensitiveSet$.

Complexity analysis. Same to Algorithm 4.1, Algorithm 5.3 takes $O(|\delta| \times |\Delta|)$ time and $O(|\delta \times \Delta|)$ space, where $|\delta \times \Delta|$ denotes the size of BPDS rules that can be constructed without the reduction.

Let n_{SR} be the number of LPDS rules (in Δ) whose heads belong to $SensitiveSet$, and n_{sync} be the number of BPDS rules (in Δ') where the corresponding BA transitions and LPDS rules are dependent. We have $|\Delta_{hori}| = n_{SR} \times |\delta|$ and $|\Delta_{sync}| = n_{sync}$. As illustrated in Figure 5.4, asynchronous transitions can be organized as

Algorithm 5.3 BPDSRulesViaSPOR($\delta \times \Delta$)

```
$\Delta_{sync} \leftarrow \emptyset$, $\Delta_{vert} \leftarrow \emptyset$, $\Delta_{hori} \leftarrow \emptyset$
for all $r = \langle g, \gamma \rangle \overset{\tau}{\hookrightarrow} \langle g', \omega \rangle \in \Delta$ do
    for all $t = q \xrightarrow{\sigma} q' \in \delta$ and $\sigma \subseteq L_{\mathcal{P}2\mathcal{B}}(\langle g, \gamma \rangle)$ and $\tau \subseteq L_{\mathcal{B}2\mathcal{P}}(q)$ do
        if  $r$ and $t$ are dependent then
            {$\mathcal{B}$ and $\mathcal{P}$ must transition together}
            $\Delta_{sync} \leftarrow \Delta_{sync} \bigcup \{\langle (g, q), \gamma \rangle \hookrightarrow_{\mathcal{BP}} \langle (g', q'), \omega \rangle\}$
        else
            {Vertical edges (see Figure 5.4b), when $\mathcal{P}$ transitions and $\mathcal{B}$ self-loops}
            $\Delta_{vert} \leftarrow \Delta_{vert} \bigcup \{\langle (g, q), \gamma \rangle \hookrightarrow_{\mathcal{BP}} \langle (g', q), \omega \rangle\}$
            if  $\langle g, \gamma \rangle \in SensitiveSet$ then
                {Horizontal edges (see Figure 5.4b), when $\mathcal{B}$ transitions $\mathcal{P}$ self-loops}
                $\Delta_{hori} \leftarrow \Delta_{hori} \bigcup \{\langle (g, q), \gamma \rangle \hookrightarrow_{\mathcal{BP}} \langle (g, q'), \gamma \rangle\}$
            end if
        end if
    end for
end for
$\Delta'_r \leftarrow \Delta_{sync} \bigcup \Delta_{vert} \bigcup \Delta_{hori}$
return  $\Delta'_r$
```

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 5.3 is $|\Delta'_r| = n_{sync} + \frac{|\delta \times \Delta| - n_{sync}}{3} + n_{SR} \times |\delta| = \frac{2}{3} n_{sync} + \frac{|\delta \times \Delta|}{3} + n_{SR} \times |\delta|$. The number of transition rules reduced is $|\Delta'| - |\Delta'_r| = \frac{2}{3}|\delta \times \Delta| - \frac{2}{3} n_{sync} - n_{SR} \times |\delta|$. We can infer from this expression that the fewer dependent transitions of $\mathcal{B}$ and $\mathcal{P}$ the more BPDS rules Algorithm 5.3 can reduce.

5.4.2 Correctness Argument

In Algorithm 5.3, a diagonal rule is reduced if the corresponding BA transition and LPDS rule are independent. 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 $\forall t \in \delta$, if $head(c) = h$ then r and t 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 dependent and independent with the BA transitions. Given a trace of $\mathcal{BP}_r$ in the form of $\langle (g_0, q_0), \omega_0 \rangle \Rightarrow_{\mathcal{BP}} \ldots \Rightarrow_{\mathcal{BP}} \langle (g_j, q_j), \omega_j \rangle \Rightarrow_{\mathcal{BP}} \ldots \Rightarrow_{\mathcal{BP}} \langle (g_k, q_k), \omega_k \rangle \Rightarrow_{\mathcal{BP}} \ldots$ $(0 \leq j < k)$, 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 $\langle (g_j, q_j), \omega_j \rangle$ and $\langle (g_{k-1}, q_{k-1}), \omega_{k-1} \rangle$ in the trace (i.e., $q_j = q_{k-1}$), because the horizontal transitions have been reduced.

Theorem 5.2. *$\mathcal{BP}_r$ preserves the reachability of $\mathcal{BP}$ from the initial configuration.*

Proof. It is easy to observe that $\mathcal{BP}_r$ and $\mathcal{BP}$ have the same state space and initial configuration, so the question is to prove that (1) given a trace of $\mathcal{BP}$ in the form of $\phi = c_0 \Rightarrow_{\mathcal{BP}} c_1 \ldots \Rightarrow_{\mathcal{BP}} c$, there is a corresponding trace of $\mathcal{BP}_r$ such that $\phi' = c_0 \Rightarrow_{\mathcal{BP}} c'_1 \ldots \Rightarrow_{\mathcal{BP}} c$; and (2) vice versa.

"⇒": Two types of transitions are reduced in $\mathcal{BP}_r$, compared to $\mathcal{BP}$. 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 mathematical induction.

Basis. If $|\phi| = 0$, i.e., $c = c_0$, the reachability trivially holds on $\mathcal{BP}_r$. If $|\phi| = 1$, because there is no horizontal transition reduced on the initial configuration, for any transition $c_0 \Rightarrow_{\mathcal{BP}} c$ of $\mathcal{BP}$ there must be a corresponding trace of $\mathcal{BP}_r$ that preserves the reachability.

Inductive step. Given a trace $\phi = c_0 \Rightarrow_{\mathcal{BP}} c_1 \ldots \Rightarrow_{\mathcal{BP}} c_i \Rightarrow_{\mathcal{BP}} c'$ $(i \geq 0)$ of $\mathcal{BP}$ where $|\phi| = i + 1$, if there exists a trace $\phi' = c_0 \Rightarrow_{\mathcal{BP}} c'_1 \ldots \Rightarrow_{\mathcal{BP}} c'_j \Rightarrow_{\mathcal{BP}} c'$ $(j \geq 0)$ of $\mathcal{BP}_r$ where $|\phi'| = j + 1$, we show that for every $t = c' \Rightarrow_{\mathcal{BP}} c$ of $\mathcal{BP}$, there is a trace of $\mathcal{BP}_r$ such that $c_0 \Rightarrow^*_{\mathcal{BP}} c$. Recall that the horizontal transitions are reduced in $\mathcal{BP}_r$ except at configurations whose heads belong to *SensitiveSet*, so we need to prove that this reduction does not affect the reachability if t involves a horizontal transition that is reduced in $\mathcal{BP}_r$. In the trace ϕ', 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 $\mathit{head}(\langle g_k, \omega_k \rangle) \in \mathit{SensitiveSet}$. Thus, the path from c'_k to c' has the form of

$$(c'_k : \langle (g_k, q_k), \omega_k \rangle) \Rightarrow_{\mathcal{BP}} \langle (g_{k+1}, q_k), \omega_{k+1} \rangle \Rightarrow_{\mathcal{BP}} \ldots \Rightarrow_{\mathcal{BP}} (c' : \langle (g_{j+1}, q_k), \omega_{j+1} \rangle),$$

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, $\mathcal{BP}_r$ cannot directly have the transition $(c' : \langle (g_{j+1}, q_k), \omega_{j+1} \rangle) \Rightarrow_{\mathcal{BP}} (c : \langle (g_{j+1}, q_{k+1}), \omega_{j+1} \rangle)$, 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. However, since the BA transitions and LPDS transitions are independent on the path from c'_k to c', we can shift the horizontal transition backward to the position right after c'_k where the horizontal transitions are not reduced. In this case, the path is

$\underline{(c'_k : \langle (g_k, q_k), \omega_k\rangle) \Rightarrow_{\mathcal{BP}} \langle (g_k, q_{k+1}), \omega_k\rangle} \Rightarrow_{\mathcal{BP}} \langle (g_{k+1}, q_{k+1}), \omega_{k+1}\rangle \Rightarrow_{\mathcal{BP}} \ldots \Rightarrow_{\mathcal{BP}} (c : \langle (g_{j+1}, q_{k+1}), \omega_{j+1}\rangle)$.

Therefore, we have proven that there exists a trace of $\mathcal{BP}_r$ such that $c_0 \Rightarrow^*_{\mathcal{BP}} c$.

"$\Leftarrow$": The other direction always holds because $\Delta'_r \subseteq \Delta'$. For every rule of $\mathcal{BP}_r$, $\mathcal{BP}$ has the same rule. Thus, for every trace of $\mathcal{BP}_r$, $\mathcal{BP}$ has the same trace. □

Theorem 5.3. *$\mathcal{BP}_r$ is optimal with respect to static partial order reduction.*

Proof. Since static partial order reduction is applied on the model before model checking, information available only during the model checking process cannot be utilized. For example, given two LPDS rules: $r_1 = c \overset{\tau}{\hookrightarrow} c'$ that is dependent with at least one BA transition and $r_2 = c'' \overset{\tau'}{\hookrightarrow} c'$ that is independent with all BA transitions. A transition path through r_2 clearly does not need to explore a horizontal transition at c' in order to preserve the reachability. However, unless in the model checking process, we cannot know how c' is reached, i.e., via r_1 or r_2. Therefore, $head(c')$ should be added to $SensitiveSet$ and horizontal BPDS rules should not be reduced if they are related to $head(c')$.

We prove the theorem by demonstrating that any BPDS rule constructed by Algorithm 5.3 cannot be reduced without affecting the reachability properties to be verified.

- At line 6, if the BPDS rule $\langle (g, q), \gamma\rangle \hookrightarrow_{\mathcal{BP}} \langle (g', q'), \omega\rangle$ is reduced, the configuration $\langle (g', q'), \omega\rangle$ may not be reachable anymore, since $\mathcal{B}$ and $\mathcal{P}$ must transition together at dependent transitions. Furthermore, any BPDS configuration that is reachable from $\langle (g', q'), \omega\rangle$ may also be affected. Note that $\langle (g', q'), \omega\rangle$ may still be reachable through other BPDS paths even if the rule is reduced, but we cannot know this unless in the model checking process. Therefore, we should not reduce the rule;

- At line 9, if $\langle (g, q), \gamma\rangle \hookrightarrow_{\mathcal{BP}} \langle (g', q), \omega\rangle$ is reduced, the configuration $\langle (g', q), \omega\rangle$ may not be reachable anymore. For example, in Figure 5.4b, reduce any vertical transition may affect the reachability to some BPDS configurations;

- At line 12, if $\langle (g, q), \gamma\rangle \hookrightarrow_{\mathcal{BP}} \langle (g, q'), \gamma\rangle$ is reduced, the configuration $\langle (g, q'), \gamma\rangle$ may not be reachable anymore. For example, in Figure 5.4b, reduce any horizontal transition may affect the reachability to some BPDS configurations.

With the information available in static partial order reduction, we cannot ensure reducing any rule constructed by Algorithm 5.3 without affecting the reachability properties; therefore, we have proven that $\mathcal{BP}_r$ is optimal. □

Example 1. Given a BA transition, $t = \texttt{Wrk} \xrightarrow{\{no\_event\}} \texttt{Idle}$, illustrated in Figure 4.1 and an LPDS rule, $r = \langle a,\ main_1\rangle \xhookrightarrow{\{no\_intr\}} \langle !a,\ main_2\rangle$, illustrated in Figure 4.4, Algorithm 4.1 constructed the following BPDS rules:

- $\langle (a, \texttt{Wrk}),\ main_1\rangle \hookrightarrow_{\mathcal{BP}} \langle (a, \texttt{Idle}),\ main_1\rangle$, i.e., $\mathcal{B}$ transitions and $\mathcal{P}$ self-loops;

- $\langle (a, \texttt{Wrk}),\ main_1\rangle \hookrightarrow_{\mathcal{BP}} \langle (!a, \texttt{Wrk}),\ main_2\rangle$, i.e., $\mathcal{P}$ transitions and $\mathcal{B}$ self-loops; and

- $\langle (a, \texttt{Wrk}),\ main_1\rangle \hookrightarrow_{\mathcal{BP}} \langle (!a, \texttt{Idle}),\ main_2\rangle$, i.e., $\mathcal{B}$ and $\mathcal{P}$ transitions together.

Since t and r are independent, $\mathcal{B}$ and $\mathcal{P}$ do not need to transition together. Furthermore, the LPDS head $\langle a, main_1\rangle$ is not in *SensitiveSet*, since there is no BA transition dependent with an LPDS rule that transition to $\langle a, main_1\rangle$. Therefore, Algorithm 5.3 only constructs one BPDS rule $\langle (a, \texttt{Wrk}),\ main_1\rangle \hookrightarrow_{\mathcal{BP}} \langle (!a, \texttt{Wrk}),\ main_2\rangle$, while the first and third BPDS rules are reduced.

Example 2. Given a BA transition $t = \texttt{Init} \xrightarrow{\{reset\}} \texttt{Rst}$ and an LPDS rule $r = \langle a,\ reset_0\rangle \xhookrightarrow{\{no\_intr\}} \langle a,\ reset_1\rangle$, since t and r are dependent, both Algorithm 4.1 and Algorithm 5.3 need to construct the BPDS rule $\langle (a, \texttt{Init}),\ reset_0\rangle \hookrightarrow_{\mathcal{BP}} \langle (a, \texttt{Rst}),\ reset_1\rangle$ to represent the synchronous transition of $\mathcal{B}$ and $\mathcal{P}$.

Example 3. Given a BA transition $t = \texttt{Intr} \xrightarrow{\{no\_event\}} \texttt{Wrk}$ and an LPDS rule $r = \langle a,\ NonHWRelated_1\rangle \xhookrightarrow{\{intr\}} \langle a,\ isr_0\ NonHWRelated_1\rangle$, since t and r are dependent, both Algorithm 4.1 and Algorithm 5.3 need to construct the BPDS rule $\langle (a, \texttt{Intr}),\ NonHWRelated_1\rangle \hookrightarrow_{\mathcal{BP}} \langle (a, \texttt{Wrk}),\ isr_0\ NonHWRelated_1\rangle$ to represent the synchronous transition of $\mathcal{B}$ and $\mathcal{P}$. However, this BPDS rule is actually unnecessary. Since the procedure *NonHWRelated* neither operates the hardware nor accesses any software global variable, interrupting *NonHWRelated* to execute the ISR will not affect the verification results.

Reducing ISR calls. Example 3 demonstrates that ISR calls are unnecessary after some LPDS transitions; therefore, these ISR calls should be reduced. Following the idea of relative atomicity (see Chapter 3), we can understand the execution of ISR as an atomic transition with respect to other lower-priority software routines. A statement of the lower-priority routines is dependent with such an ISR transition if and only if the statement operates hardware or accesses software global variables; otherwise, the statement is independent with the ISR transition. Based on this observation, the idea similar to Algorithm 5.3 can also be applied to reduced the ISR calls introduced to LPDS. Chapter 6 will further discuss the reduction of ISR calls combined with Algorithm 5.3 in implementation.

5.5 OPTIMIZATION OF LTL CHECKING

5.5.1 Reduction Algorithm

When verifying an LTL property on a BPDS $\mathcal{BP}$, some transition orders between the BA $\mathcal{B}$ and the LPDS $\mathcal{P}$ can also be reduced without affecting the verification result. In this section, we present how to utilize the concept of static partial order reduction in the LTL checking of BPDS. We denote the reduced BPDS model as $\mathcal{BP}_r$. Let Δ'_r be the set of BPDS rules of $\mathcal{BP}_r$ and Δ' be the set of BPDS rules of $\mathcal{BP}$. We have $\Delta'_r \subseteq \Delta'$, i.e., $\mathcal{BP}_r$ has a smaller set of BPDS rules compared to $\mathcal{BP}$.

In reachability analysis, we have demonstrated that static partial order reduction can be applied on BPDS without affecting the reachability from the initial configuration to any other configurations. This reduction is conservative, since there always exists at least one trace that preserves the reachability to certain configuration. However, LTL checking is different, since we not only need to know whether a configuration is reachable, but also need to know how the configuration is reached. In other words, without the knowledge about what LTL property to verify, a reachability-preserving trace may not be able to preserve the LTL property. Therefore, we need to consider the LTL property in our reduction algorithm.

As discussed in Chapter 2, there are five temporal operators that are commonly used to specify LTL properties. Partial order reduction cannot be effectively applied with the next operator, $\boldsymbol{X}$. Intuitively, next operator states the relation between two propositions within one state transition, which can make all transition orders between $\mathcal{B}$ and $\mathcal{P}$ matter to the verification result. Therefore, we apply static partial order reduction with LTL properties that do not use the next operator. This type of LTL property is denoted as LTL$_{-X}$.

Definition 5.7. Given an LTL$_{-X}$ formula φ to be verified on $\mathcal{BP}$, a BPDS rule $c \hookrightarrow_{\mathcal{BP}} c'$ is *invisible* to φ if and only if $L_\varphi(c) = L_\varphi(c')$, i.e., all state transitions that follow this BPDS rule do not change the value of the propositional variables in $At(\varphi)$; otherwise the rule is *visible* to φ. If all the transitions on a BPDS path are invisible to φ, the path is also invisible to φ.

Definition 5.8. Given a BPDS rule $r_{\mathcal{BP}}$, $VisProp(r_{\mathcal{BP}})$ denotes the set of propositional variables whose value is affected by the BPDS rule $r_{\mathcal{BP}}$. Obviously, if $VisProp(r_{\mathcal{BP}}) = \emptyset$, $r_{\mathcal{BP}}$ is invisible.

- Given $t = q \xrightarrow{\sigma} q' \in \delta$ and $a \in 2^{At(\varphi)}$, for every $r_{\mathcal{BP}} = \langle (g, q), \gamma \rangle \hookrightarrow_{\mathcal{BP}} \langle (g, q'), \gamma \rangle \in \Delta'$, if $VisProp(r_{\mathcal{BP}}) = a \neq \emptyset$, t is said to be *horizontally visible.*

- Given $r = \langle g, \gamma \rangle \overset{\tau}{\hookrightarrow} \langle g', \omega \rangle \in \Delta$ and $a \in 2^{At(\varphi)}$, for every $r_{\mathcal{BP}} = \langle (g, q), \gamma \rangle \hookrightarrow_{\mathcal{BP}} \langle (g', q), \omega \rangle \in \Delta'$, if $VisProp(r_{\mathcal{BP}}) = a \neq \emptyset$, r is said to be *vertically visible.*

Intuitively, horizontal visibility describes the situation when some propositional variables are evaluated only based on the states of BA; vertical visibility describes the situation when some propositional variables are evaluated only based on the states of LPDS. This kind of classification, as quite useful in symbolic representations (see Section 5.6), can help us reduce many visible BPDS rules without affecting the LTL$_{-X}$ properties to be verified.

Given a BA transition t and an LPDS rule r, Algorithm 5.4 decides whether the corresponding diagonal/horizontal BPDS rules are reducible candidates. We should assume that t and r are independent; otherwise, since $\mathcal{B}$ and $\mathcal{P}$ must transition together when t and r are dependent, no BPDS rule can be reduced.

- Between line 8 and line 9, if there is no visible BPDS rule, both the horizontal rule r_1 and the diagonal rule r_3 are reducible candidates;

- Between line 11 and line 13, the diagonal rule r_3 is a reducible candidate if it is replaceable by a horizontal rule and a vertical rule. Lemma 5.3 will discuss the correctness of this reduction;

Algorithm 5.4 ReducibleBPDSRules($t \in \delta, r \in \Delta$)

```
Require: $t$ and $r$ are independent.
$ReduceDiag \leftarrow$ FALSE, $ReduceHori \leftarrow$ FALSE
Let $t = q \xrightarrow{\sigma} q'$
    $r = \langle g, \gamma \rangle \overset{\tau}{\hookrightarrow} \langle g', \omega \rangle$
    $r_1 = \langle (g, q), \gamma \rangle \hookrightarrow_{\mathcal{BP}} \langle (g, q'), \gamma \rangle$ {Horizontal BPDS rules, see Figure 5.4a}
    $r_2 = \langle (g, q), \gamma \rangle \hookrightarrow_{\mathcal{BP}} \langle (g', q), \omega \rangle$ {Vertical BPDS rules, see Figure 5.4a}
    $r_3 = \langle (g, q), \gamma \rangle \hookrightarrow_{\mathcal{BP}} \langle (g', q'), \omega \rangle$ {Diagonal BPDS rules, see Figure 5.4a}
if $VisProp(r_1) = \emptyset$ and $VisProp(r_2) = \emptyset$ and $VisProp(r_3) = \emptyset$ then
  {If $r_1$, $r_2$, and $r_3$ are all invisible}
  $ReduceDiag \leftarrow$ TRUE, $ReduceHori \leftarrow$ TRUE
else
  if $VisProp(r_1) = VisProp(r_3)$ or $VisProp(r_2) = VisProp(r_3)$ or
      $VisProp(r_1) = \emptyset$ or $VisProp(r_2) = \emptyset$ then
    $ReduceDiag \leftarrow$ TRUE
  end if
  if $r_1$ is invisible or $t$ is horizontally visible then
    $ReduceHori \leftarrow$ TRUE
  end if
end if
return $(ReduceDiag, ReduceHori)$
```

- Between line 14 and line 16, the horizontal rule r_1 is a reducible candidate if it is either invisible or constructed from a BA transition (i.e., t) that is horizontally visible. Theorem 5.5 will discuss the correctness of this reduction.

Definition 5.9. Similar to the reduction applied in reachability analysis, we need to decide which BPDS rules cannot be reduced. Therefore, we identify three sets of heads, *SensitiveSet*, $VisibleSet$, and *LoopSet* on $Conf(\mathcal{P})$ as follows:

- $SensitiveSet = \{\ head(\langle g_0, \omega_0\rangle)\ \} \bigcup \{\ head(c') \mid \exists r = c \overset{\tau}{\hookrightarrow} c' \in \Delta, \exists t \in \delta, r$ and t are dependent $\}$, where $\langle g_0, \omega_0\rangle$ is the initial configuration of $\mathcal{P}$;

- $VisbileSet = \{\ head(\langle g', \omega\rangle) \mid \exists r = \langle (g, q), \gamma\rangle \hookrightarrow_{\mathcal{BP}} \langle (g', q'), \omega\rangle \in \Delta'$ that is visible to φ; and r is irreducible according to Algorithm 5.4 $\}$;

- $LoopSet = \{\ h \mid$ for every strongly connected component C in $\mathcal{G}_{\mathcal{P}}$, pick a head h from $C\ \}$, where $\mathcal{G}_{\mathcal{P}}$ is the head reachability graph of $\mathcal{P}$ and there is no preference on how h is selected from C.

SensitiveSet is necessary to preserve the reachability from the initial configuration to other configurations; the concept of $VisibleSet$ is similar to that of $SensitiveSet$, i.e., preserving the reachability of BPDS paths right after a visible transition that cannot be reduced according to Algorithm 5.4; *LoopSet*, similar to the concept of cycle closing condition [44], is introduced to satisfy the BPDS loop constraint when a loop of $\mathcal{P}$ is involved in the accepting run of $\mathcal{B}^2\mathcal{P}$.

Algorithm 5.5 applies the reduction following the idea illustrated in Figure 5.4b, where the horizontal/diagonal edges are reduced.

- At line 6, since the LPDS rule r and the BA transition t are dependent, $\mathcal{B}$ and $\mathcal{P}$ must transition together; therefore, we construct a BPDS rule $\langle (g, q), \gamma\rangle \hookrightarrow_{\mathcal{BP}} \langle (g', q'), \omega\rangle$;

Algorithm 5.5 BPDSRulesViaSPOR_LTL($\delta \times \Delta$)

```
$\Delta_{sync} \leftarrow \emptyset$, $\Delta_{vert} \leftarrow \emptyset$, $\Delta_{hori} \leftarrow \emptyset$, $\Delta_{diag} \leftarrow \emptyset$
for all $r = \langle g, \gamma \rangle \overset{\tau}{\hookrightarrow} \langle g', \omega \rangle \in \Delta$ do
  for all $t = q \xrightarrow{\sigma} q' \in \delta$ and $\sigma \subseteq L_{\mathcal{P}2\mathcal{B}}(\langle g, \gamma \rangle)$ and $\tau \subseteq L_{\mathcal{B}2\mathcal{P}}(q)$ do
    if  $r$ and $t$ are dependent then
      {$\mathcal{B}$ and $\mathcal{P}$ must transition together}
      $\Delta_{sync} \leftarrow \Delta_{sync} \bigcup \{\langle (g, q), \gamma \rangle \hookrightarrow_{\mathcal{BP}} \langle (g', q'), \omega \rangle\}$
    else
      {$\mathcal{P}$ transitions and $\mathcal{B}$ self-loops}
      $\Delta_{vert} \leftarrow \Delta_{vert} \bigcup \{\langle (g, q), \gamma \rangle \hookrightarrow_{\mathcal{BP}} \langle (g', q), \omega \rangle\}$
      $(ReduceDiag, ReduceHori) \leftarrow$ ReducibleBPDSRules$(t, r)$
      if $ReduceDiag$ = FALSE then
        {$\mathcal{B}$ and $\mathcal{P}$ transition together}
        $\Delta_{diag} \leftarrow \Delta_{diag} \bigcup \{\langle (g, q), \gamma \rangle \hookrightarrow_{\mathcal{BP}} \langle (g', q'), \omega \rangle\}$
      end if
      if $ReduceHori$ = FALSE or
        $\langle g, \gamma \rangle \in SensitiveSet \bigcup VisibleSet \bigcup LoopSet$ then
        {$\mathcal{B}$ transitions and $\mathcal{P}$ self-loops}
        $\Delta_{hori} \leftarrow \Delta_{hori} \bigcup \{\langle (g, q), \gamma \rangle \hookrightarrow_{\mathcal{BP}} \langle (g, q'), \gamma \rangle\}$
      end if
    end if
  end for
end for
$\Delta'_r \leftarrow \Delta_{sync} \bigcup \Delta_{vert} \bigcup \Delta_{hori} \bigcup \Delta_{diag}$
return  $\Delta'_r$
```

- At line 9, we construct a vertical rule $\langle (g, q), \gamma\rangle \hookrightarrow_{\mathcal{BP}} \langle (g', q), \omega\rangle$ to represent the asynchronous situation when $\mathcal{P}$ transitions and $\mathcal{B}$ self-loops. Since Algorithm 5.5 follows the reduction demonstrated in Figure 5.4b, all vertical BPDS rules are preserved;

- At line 10, we invoke Algorithm 5.4, i.e., ReducibleBPDSRules, to decide if the horizontal/diagonal BPDS rules are reducible candidates;

- Between line 11 and line 14, we construct a diagonal BPDS rule if necessary;

- Between line 15 and line 18, we construct a horizontal BPDS rule if necessary; Note that even if ReducibleBPDSRules returns `TRUE` for $ReduceHori$, we still have to preserve this horizontal BPDS rule if $head(r)$ belongs to $SensitiveSet$, $VisibleSet$, or $LoopSet$.

Complexity analysis. Algorithm 5.5 takes $O(|\delta| \times |\Delta|)$ time and $O(|\delta \times \Delta|)$ space, where $|\delta \times \Delta|$ denotes the size of BPDS rules that can be constructed without the reduction.

Let n_{sync} be the number of BPDS rules that are generated from dependent BA transitions and LPDS rules (at line 6), n_v be the number of BPDS rules related to visible transitions (i.e., when Algorithm 5.4 returns $ReduceDiag$ or $ReduceHori$ as `FALSE`), n_{svl} be the number of BPDS rules associated to $SensitiveSet$, $VisibleSet$, and $LoopSet$ (at line 17 when $ReduceHori$ is `TRUE`). We have $|\Delta_{hori} \bigcup \Delta_{diag}| = n_v + n_{svl}$ and $|\Delta_{sync}| = n_{sync}$. As illustrated in Figure 5.4, 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 by Algorithm 5.5 is $|\Delta'_r| = n_{sync} + \frac{|\delta\times\Delta| - n_{sync}}{3} + n_v + n_{svl} = \frac{2}{3}n_{sync} + \frac{|\delta\times\Delta|}{3} + n_v + n_{svl}$. The number of transition rules reduced is $|\Delta'| - |\Delta'_r| = \frac{2}{3}|\delta \times \Delta| - n_v - \frac{2}{3}n_{sync} - n_{svl}$. Therefore, our reduction is effective when the following criteria have small sizes:

- BPDS rules visible to φ;
- dependent transitions of $\mathcal{B}$ and $\mathcal{P}$; and
- loops in $\mathcal{P}$.

5.5.2 Correctness Argument

We prove the correctness of the reduction by two steps. First, we assume that no visible BPDS rule (including the related invisible BPDS rules) is reduced by Algorithm 5.5. More specifically, the pseudo code between line 10 and line 17 of Algorithm 5.4 is not used in this case. Based on this assumption, let the reduced BPDS model be $\mathcal{BP}'_r$. We prove that any LTL$_{-X}$ property is invariant on $\mathcal{BP}$ and $\mathcal{BP}'_r$. Second, let the reduced BPDS model without the assumption be $\mathcal{BP}_r$. We prove that any LTL$_{-X}$ property is invariant on $\mathcal{BP}'_r$ and $\mathcal{BP}_r$.

First, any LTL$_{-X}$ property is invariant on $\mathcal{BP}$ and $\mathcal{BP}'_r$. There are several concepts that can help our proof.

Definition 5.10. Given a labeling function L, two infinite paths $\phi_1 = s_0 \rightarrow s_1 \rightarrow \ldots$ and $\phi_2 = q_0 \rightarrow q_1 \rightarrow \ldots$ are *stuttering equivalent*, written as $\phi_1 \sim_{st} \phi_2$, if there are two infinite sequences of positive integers $0 = i_0 < i_1 < i_2 < \ldots$ and $0 = j_0 < j_1 < j_2 < \ldots$ such that for every $k \geq 0$, $L(s_{i_k}) = L(s_{i_k+1}) = \ldots = L(s_{i_{k+1}-1}) = L(q_{j_k}) = L(q_{j_k+1}) = \ldots = L(q_{j_{k+1}-1})$.

Definition 5.11. We define a *transition block* as a BPDS path $K = c \Rightarrow^*_{\mathcal{BP}} c'$ such that K is invisible, where for $c = \langle (g, q), \omega \rangle$, $head(\langle g, \omega \rangle) \in VisibleSet$. K can be considered as an invisible path right after a visible transition. Given two transition blocks $K = c \Rightarrow^*_{\mathcal{BP}} c'$ and $K' = c'' \Rightarrow^*_{\mathcal{BP}} c'''$, they are referred to as *corresponding transition blocks* if $c = c''$ and $c' = c'''$. Obviously, $K \sim_{st} K'$.

Lemma 5.1. *If $\mathcal{BP}$ has a transition block $K = c_0 \Rightarrow_{\mathcal{BP}} c_1 \Rightarrow_{\mathcal{BP}} \ldots \Rightarrow_{\mathcal{BP}} c$, $\mathcal{BP}'_r$ always has a corresponding transition block $K' = c_0 \Rightarrow_{\mathcal{BP}} c'_1 \Rightarrow_{\mathcal{BP}} \ldots \Rightarrow_{\mathcal{BP}} c$.*

Proof. Two types of transitions are reduced in $\mathcal{BP}'_r$: diagonal and horizontal.

First, the reduction of diagonal transitions does not affect this lemma. Given any invisible diagonal transition $t = c \Rightarrow_{\mathcal{BP}} c'$, if it is reduced by Algorithm 5.5, all transitions starting from c must be invisible. Therefore, we can always use an invisible path, $(c : \langle (g, q), \gamma v\rangle) \Rightarrow_{\mathcal{BP}} (c'' : \langle (g, q'), \gamma v\rangle) \Rightarrow_{\mathcal{BP}} (c' : \langle (g', q'), \omega v\rangle)$, to replace t, where $L_\varphi(c) = L_\varphi(c'') = L_\varphi(c')$, $\gamma \in \Gamma$, and $v, \omega \in \Gamma^*$.

Second, we prove that the reduction of horizontal transitions does not affect this lemma by mathematical induction.

Basis. When $|K| = 0$, i.e., $c = c_0$, the lemma trivially holds. When $|K| = 1$, since no horizontal edges are reduced at c_0, the lemma also holds.

Inductive step. Given $K = c_0 \Rightarrow_{\mathcal{BP}} c_1 \ldots \Rightarrow_{\mathcal{BP}} c_{i-1} \Rightarrow_{\mathcal{BP}} c'$, where $|K| = i > 0$, if $\mathcal{BP}'_r$ has a transition block $K' = c_0 \Rightarrow_{\mathcal{BP}} c'_1 \ldots \Rightarrow_{\mathcal{BP}} c'_{j-1} \Rightarrow_{\mathcal{BP}} c'$ where $|K'| = j > 0$, we show that for every invisible transition $t = c' \Rightarrow_{\mathcal{BP}} c$ of $\mathcal{BP}$, there is a transition block of $\mathcal{BP}'_r$ such that $c_0 \Rightarrow^*_{\mathcal{BP}} c$.

In K', we can always find a configuration $c'_k = \langle (g_k, q_k), \omega_k\rangle$ $(0 \le k < j)$ such that c'_k is the last configuration satisfying

$$head(\langle g_k, \omega_k\rangle) \in SensitiveSet \bigcup VisibleSet \bigcup LoopSet.$$

Thus, the path from c'_k to c' has the form of

$$(c'_k : \langle (g_k, q_k), \omega_k\rangle) \Rightarrow_{\mathcal{BP}} \langle (g_{k+1}, q_k), \omega_{k+1}\rangle \Rightarrow_{\mathcal{BP}} \ldots \Rightarrow_{\mathcal{BP}} (c' : \langle (g_j, q_k), \omega_j\rangle),$$

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, $\mathcal{BP}'_r$ cannot have a horizontal transition from c' to c. However, since the BA transitions and LPDS transitions are independent on the path from c'_k to c', we can shift the horizontal transition backward to the position right after c'_k where the horizontal transitions are not reduced. In this case, the path is

$\underline{(c'_k : \langle (g_k, q_k), \omega_k \rangle) \Rightarrow_{\mathcal{BP}} \langle (g_k, q_{k+1}), \omega_k \rangle} \Rightarrow_{\mathcal{BP}} \langle (g_{k+1}, q_{k+1}), \omega_{k+1} \rangle \Rightarrow_{\mathcal{BP}} \ldots \Rightarrow_{\mathcal{BP}} (c : \langle (g_j, q_{k+1}), \omega_j \rangle)$.

Note that, this path is invisible, because $\mathcal{BP}'_r$ does not have any visible transitions on the paths between c'_k and c. Otherwise, there must be a configuration, $\langle (g, q), \omega \rangle$ after c'_k on path K', such that $head(\langle g, \omega \rangle) \in VisibleSet$. Therefore, $\mathcal{BP}'_r$ has a transition block $c_0 \Rightarrow^*_{\mathcal{BP}} c$. □

Lemma 5.2. *Any LTL$_{-X}$ property is invariant under stuttering [22].*

Theorem 5.4. *Any LTL$_{-X}$ property is invariant on $\mathcal{BP}$ and $\mathcal{BP}'_r$.*

Proof. We prove that if $\mathcal{BP}$ has a trace ϕ, $\mathcal{BP}'_r$ always has a trace ϕ' that is stuttering equivalent to ϕ; and vice versa.

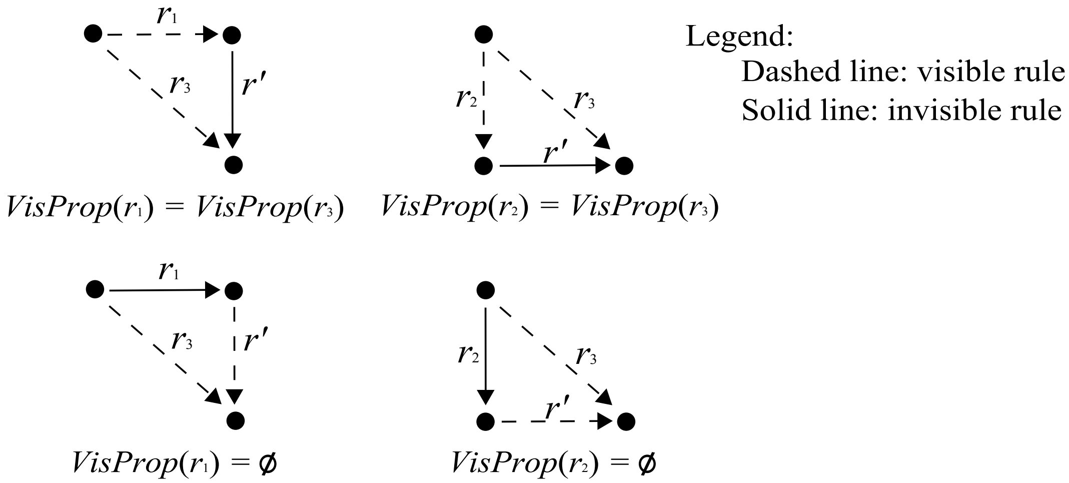
"⇒": ϕ can be written as a sequence of transition blocks such that $K_0 \Rightarrow_{\mathcal{BP}} K_1 \Rightarrow_{\mathcal{BP}} \ldots$, where only the transitions between the transition blocks are visible. Since no visible transition is reduced, $\mathcal{BP}'_r$ has the same transitions that connect these transition blocks in ϕ. Lemma 5.1 has already proven that $\forall i \geq 0$, $\mathcal{BP}'_r$ has K'_i corresponding to K_i. Therefore, $\mathcal{BP}'_r$ has a trace ϕ' such that $\phi \sim_{st} \phi'$.

"⇐": For every rule of $\mathcal{BP}'_r$, $\mathcal{BP}$ has the same rule; therefore, for every trace of $\mathcal{BP}'_r$, $\mathcal{BP}$ has the same trace. □

Second, any LTL$_{-X}$ property is invariant on $\mathcal{BP}'_r$ and $\mathcal{BP}_r$.

Lemma 5.3. *Any diagonal BPDS rule (written as $\langle (g, q), \gamma \rangle \hookrightarrow_{\mathcal{BP}} \langle (g', q'), \omega \rangle$) reduced according to Algorithm 5.4 can be replaced by a horizontal BPDS rule and a vertical BPDS rule.*

Proof. The diagonal BPDS rule can be either visible or invisible. As illustrated in Figure 5.5, a visible diagonal BPDS rule is reduced in the following four conditions:



r_1
r_3
r'
$VisProp(r_1) = VisProp(r_3)$
r_2
r_3
r'
$VisProp(r_2) = VisProp(r_3)$
Legend:
Dashed line: visible rule
Solid line: invisible rule
r_1
r_3
r'
$VisProp(r_1) = \emptyset$
r_2
r_3
r'
$VisProp(r_2) = \emptyset$

Figure 5.5: Reducible visible diagonal BPDS rules.

- For $VisProp(r_1) = VisProp(r_3)$, we know that $r' = \langle(g, q'), \gamma\rangle \hookrightarrow_{\mathcal{BP}} \langle(g', q'), \omega\rangle$ must be invisible; therefore r_3 can be replaced by r_1 and r' without affecting the stuttering equivalence between the paths of $\mathcal{BP}'_r$ and $\mathcal{BP}_r$.

- For $VisProp(r_2) = VisProp(r_3)$, we know that $r' = \langle(g', q), \gamma'\rangle \hookrightarrow_{\mathcal{BP}} \langle(g', q'), \gamma'\rangle$ must be invisible, where γ' is the top stack symbol in ω; therefore r_3 can be replaced by r_2 and r'.

- For $VisProp(r_1) = \emptyset$, given $r' = \langle(g, q'), \gamma\rangle \hookrightarrow_{\mathcal{BP}} \langle(g', q'), \omega\rangle$, we know that $VisProp(r') = VisProp(r_3)$; therefore r_3 can be replaced by r_1 and r'.

- For $VisProp(r_2) = \emptyset$, given $r' = \langle(g', q), \gamma'\rangle \hookrightarrow_{\mathcal{BP}} \langle(g', q'), \gamma'\rangle$, where γ' is the top stack symbol in ω, we know that $VisProp(r') = VisProp(r_3)$; therefore r_3 can be replaced by r_2 and r'.

As illustrated in Figure 5.6, an invisible diagonal BPDS rule is reducible in the following two conditions:

- For $VisProp(r_1) = VisProp(r_3)$ or $VisProp(r_1) = \emptyset$, we know that $r' = \langle(g, q'), \gamma\rangle \hookrightarrow_{\mathcal{BP}} \langle(g', q'), \omega\rangle$ must be invisible; therefore r_3 can be replaced by

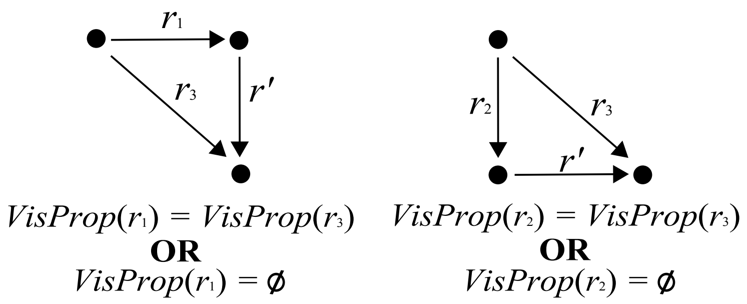

r_1
r_3
r'
r_2
r_3
r'
$VisProp(r_1) = VisProp(r_3)$
OR
$VisProp(r_1) = \emptyset$
$VisProp(r_2) = VisProp(r_3)$
OR
$VisProp(r_2) = \emptyset$

Figure 5.6: Reducible invisible diagonal BPDS rules.

r_1 and r' without affecting the stuttering equivalence between the paths of $\mathcal{BP}'_r$ and $\mathcal{BP}_r$.

- For $VisProp(r_2) = VisProp(r_3)$ or $VisProp(r_2) = \emptyset$, we know that $r' = \langle (g', q), \gamma' \rangle \hookrightarrow_{\mathcal{BP}} \langle (g', q'), \gamma' \rangle$ must be invisible, where γ' is the top stack symbol in ω; therefore r_3 can be replaced by r_2 and r'.

□

Theorem 5.5. *Any LTL_{-X} property is invariant on $\mathcal{BP}'_r$ and $\mathcal{BP}_r$.*

Proof. We prove that given a trace of $\mathcal{BP}'_r$ in the form of $\phi' = c_0 \Rightarrow_{\mathcal{BP}} c'_1 \ldots \Rightarrow_{\mathcal{BP}} c$, there is a trace of $\mathcal{BP}_r$ in the form of $\phi = c_0 \Rightarrow_{\mathcal{BP}} c_1 \ldots \Rightarrow_{\mathcal{BP}} c$, such that ϕ' and ϕ are stuttering equivalent; and (2) vice versa.

"⇒": Lemma 5.3 has demonstrated that the reduction of diagonal BPDS rules according to Algorithm 5.4 does not affect the stuttering equivalence between any traces of $\mathcal{BP}'_r$ and $\mathcal{BP}_r$. Therefore, we only need to prove that the reduction of horizontal BPDS rules does not affect the stuttering equivalence neither. In Algorithm 5.4, a horizontal BPDS rule is considered as a reducible candidate if it is either invisible or constructed from a BA transition that is horizontally visible. In both ways, the horizontal transition can be shifted backward on the BPDS trace without affecting the stuttering equivalence requirement. We prove this direction of the theorem by mathematical induction.

Basis. If $|\phi'| = 0$, i.e., $c = c_0$, our argument trivially holds. If $|\phi'| = 1$, because there is no horizontal transition reduced on the initial configuration, for any transition $c_0 \Rightarrow_{\mathcal{BP}} c$ of $\mathcal{BP}'_r$, there must be a stuttering equivalent trace of $\mathcal{BP}_r$.

Inductive step. Given a trace $\phi' = c_0 \Rightarrow_{\mathcal{BP}} c'_1 \ldots \Rightarrow_{\mathcal{BP}} c'_j \Rightarrow_{\mathcal{BP}} c'$ $(j \geq 0)$ of $\mathcal{BP}'_r$ where $|\phi'| = j + 1$, if there exists a trace $\phi = c_0 \Rightarrow_{\mathcal{BP}} c_1 \ldots \Rightarrow_{\mathcal{BP}} c_i \Rightarrow_{\mathcal{BP}} c'$ $(i \geq 0)$ of $\mathcal{BP}_r$ where $|\phi| = i + 1$, we show that for every $t = c' \Rightarrow_{\mathcal{BP}} c$ of $\mathcal{BP}'_r$, there is a trace of $\mathcal{BP}_r$ such that $c_0 \Rightarrow^*_{\mathcal{BP}} c$. Furthermore, if ϕ' and ϕ are stuttering equivalent, the new traces of $\mathcal{BP}'_r$ and $\mathcal{BP}_r$ are also stuttering equivalent.

In the trace ϕ, we can always find a configuration

$$c_k = \langle (g_k, q_k), \omega_k \rangle,\ 0 \leq k \leq i,$$

such that c_k is the last configuration satisfying

$$head(\langle g_k, \omega_k \rangle) \in SensitiveSet \bigcup VisibleSet \bigcup LoopSet.$$

Thus, the path from c_k to c' has the form of

$$(c_k : \langle (g_k, q_k), \omega_k \rangle) \Rightarrow_{\mathcal{BP}} \langle (g_{k+1}, q_k), \omega_{k+1} \rangle \Rightarrow_{\mathcal{BP}} \ldots \Rightarrow_{\mathcal{BP}} (c' : \langle (g_{i+1}, q_k), \omega_{i+1} \rangle),$$

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, $\mathcal{BP}_r$ cannot directly have the transition $(c' : \langle (g_{i+1}, q_k), \omega_{i+1} \rangle) \Rightarrow_{\mathcal{BP}} (c : \langle (g_{i+1}, q_{k+1}), \omega_{i+1} \rangle)$, i.e., the corresponding BPDS rule $\langle (g_{i+1}, q_k), \gamma_{i+1} \rangle) \hookrightarrow_{\mathcal{BP}} \langle (g_{i+1}, q_{k+1}), \gamma_{i+1} \rangle$ (γ_{i+1} is the top stack symbol of ω_{i+1}) does not exist after the reduction. However, we can shift the horizontal transition backward to the position right after c_k where the horizontal transitions are not reduced. No matter whether the transition is invisible or horizontally visible (as the two types of reducible horizontal BPDS rules according Algorithm 5.4), the paths before and after the shift operation are stuttering equivalent. We can construct the new path as

$\underline{(c'_k : \langle (g_k, q_k), \omega_k \rangle) \Rightarrow_{\mathcal{BP}} \langle (g_k, q_{k+1}), \omega_k \rangle} \Rightarrow_{\mathcal{BP}} \langle (g_{k+1}, q_{k+1}), \omega_{k+1} \rangle \Rightarrow_{\mathcal{BP}} \ldots \Rightarrow_{\mathcal{BP}} (c : \langle (g_{i+1}, q_{k+1}), \omega_{i+1} \rangle)$.

Therefore, we have proven this direction of the theorem.

"$\Leftarrow$": the other direction trivially holds because $\mathcal{BP}'_r$ has all the BPDS rules of $\mathcal{BP}_r$. □

Theorem 5.6. *Algorithm 5.5 preserves all LTL$_{-X}$ properties to be verified on $\mathcal{BP}$.*

Proof. This theorem holds, as the result of Theorem 5.4 and Theorem 5.5. □

Theorem 5.7. *$\mathcal{BP}_r$ is optimal with respect to static partial order reduction.*

Proof. Similar to the proof of Theorem 5.3, we demonstrate that any BPDS rule constructed by Algorithm 5.5 cannot be reduced without affecting the LTL$_{-X}$ property to be verified.

- At line 6, if the BPDS rule $\langle (g, q), \gamma\rangle \hookrightarrow_{\mathcal{BP}} \langle (g', q'), \omega\rangle$ is reduced, the configuration $\langle (g', q'), \omega\rangle$ may not be reachable anymore, since $\mathcal{B}$ and $\mathcal{P}$ must transition together at dependent transitions. Furthermore, any BPDS configuration that is reachable from $\langle (g', q'), \omega\rangle$ may also be affected. Since we do not know whether the reachability to $\langle (g', q'), \omega\rangle$ can affect the LTL$_{-X}$ property without going through a model checking process, the BPDS rule should not be reduced;

- At line 9, if $\langle (g, q), \gamma\rangle \hookrightarrow_{\mathcal{BP}} \langle (g', q), \omega\rangle$ is reduced, the configuration $\langle (g', q), \omega\rangle$ may not be reachable anymore. For example, in Figure 5.4b, reduce any vertical transition may affect the reachability to some BPDS configurations;

- At line 13, the diagonal rule $\langle (g, q), \gamma\rangle \hookrightarrow_{\mathcal{BP}} \langle (g', q'), \omega\rangle$ is constructed only in two possibilities as illustrated in Figure 5.7. In a general point of view, the BPDS rule cannot be reduced in either way (see Section 5.6.2 for further discussion);

- At line 17, the BPDS rule $\langle (g, q), \gamma\rangle \hookrightarrow_{\mathcal{BP}} \langle (g, q'), \gamma\rangle$ is not reduced. When $ReduceHori$ = FALSE, reducing the rule may eliminate the traces of $\mathcal{BP}_r$

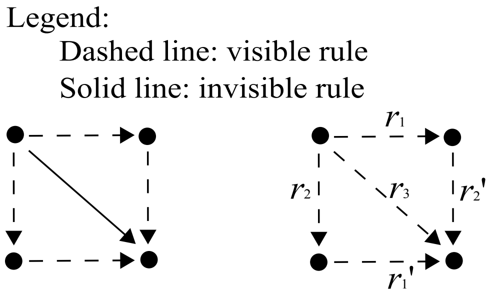


Figure 5.7: Irreducible diagonal BPDS rules.

> that are stuttering equivalent with a trace of $\mathcal{BP}'_r$; when the BPDS rule is related to *SensitiveSet*, reducing the rule can affect the reachability to $\langle (g, q'), \gamma \rangle$; when the BPDS rule is related to *VisibleSet*, reducing the rule can either affect invisible paths after visible transitions or eliminate all horizontally visible transitions after visible transitions; when the BPDS rule is related to *LoopSet*, reducing the rule can simply remove all BPDS traces, because the BPDS loop constraint may not be satisfied.

With the information available in static partial order reduction, we cannot be sure to reduce any rule constructed by Algorithm 5.5 without affecting the LTL$_{-X}$ property to be verified; therefore, we have proven that $\mathcal{BP}_r$ is optimal. □

5.6 SYMBOLIC ALGORITHMS

A system design can have an enormous number of states; therefore, it is almost impossible to specify the design explicitly in practice, where the transition relation between every two states is described by a separate rule. It is also inefficient to analyze an explicit representation, because most analysis algorithms need to explore all the transition rules.

Symbolic representation is a compact way to specify system designs. A symbolic rule describes the transition relation between two sets of states. In a general point of view, we can consider a hardware transaction function in modelC (see Chapter 3) as a symbolic rule of BA or a C statement in software programs as a symbolic rule of LPDS, because both of them describe the transition relation between two sets of states. Therefore, we can apply our static partial order reduction algorithms directly on the programs specified using C, modelC, etc. On the other hand, data structures such as BDD can be utilized to encode the transition rules of BPDS during model checking. Symbolic model checking that operates on these symbolic BPDS rules are more efficient than model checking on explicit BPDS rules. In this section, we will discuss the symbolic algorithms for the first type of symbolic representation, where the algorithms work directly on the programs in order to construct reduced BPDS models.

5.6.1 Reduction Algorithm for Reachability Analysis

Given the symbolic representations of BPDS discussed in Section 4.4, we present Algorithm 5.6 as the symbolic version of Algorithm 5.3. Algorithm 5.6 is similar to Algorithm 5.3, except that Algorithm 5.6 operates on symbolic rules of BA and LPDS in order to construct symbolic rules of BPDS. We have two observations on Algorithm 5.6:

- $\mathcal{B}$ and $\mathcal{P}$ need to transition together only when their transitions are dependent; and
- $\mathcal{B}$ and $\mathcal{P}$ can transition in an interleaved manner when their transitions are independent.

The two observations tell us how $\mathcal{BP}$ can be constructed from $\mathcal{B}$ and $\mathcal{P}$ so that only the necessary BPDS rules are included. Since a modelC program can be considered as the symbolic representation of BA and a C program can be considered as the symbolic representation of LPDS, we can construct a BPDS model by instrument the C program using the modelC program. If we drop the acceptance condition

Algorithm 5.6 SymbolicBPDSRulesViaSPOR($\delta \times \Delta$)

```
$\Delta_{sync} \leftarrow \emptyset$, $\Delta_{vert} \leftarrow \emptyset$, $\Delta_{hori} \leftarrow \emptyset$
for all $\mathcal{R} = \langle g, \gamma \rangle \overset{\tau}{\underset{R}{\hookrightarrow}} \langle g', \omega \rangle \subseteq \Delta$ do
  for all $U = Q \times \{\sigma\} \times Q \subseteq \delta$ and $\sigma \subseteq L'_{\mathcal{P}2\mathcal{B}}(\langle g, \gamma \rangle)$ and $\tau \subseteq L'_{\mathcal{B}2\mathcal{P}}(U)$ do
    if $\mathcal{R}$ and $U$ are dependent then
      {$\mathcal{B}$ and $\mathcal{P}$ must transition together}
      $\Delta_{sync} \leftarrow \Delta_{sync} \bigcup \langle g, \gamma \rangle \underset{R'}{\hookrightarrow}_{\mathcal{BP}} \langle g', \omega \rangle$, where $R' = R \times U$
    else
      {Vertical edges (see Figure 5.4b), when $\mathcal{P}$ transitions and $\mathcal{B}$ self-loops}
      $\Delta_{vert} \leftarrow \Delta_{vert} \bigcup \langle g, \gamma \rangle \underset{R'}{\hookrightarrow}_{\mathcal{BP}} \langle g', \omega \rangle$, where $R' = R \times U_{loop}$
      if $\langle g, \gamma \rangle \in SensitiveSet$ then
        {Horizontal edges (see Figure 5.4b), when $\mathcal{B}$ transitions $\mathcal{P}$ self-loops}
        $\Delta_{hori} \leftarrow \Delta_{hori} \bigcup \langle g, \gamma \rangle \underset{R'}{\hookrightarrow}_{\mathcal{BP}} \langle g, \gamma \rangle$, where $R' = R_{loop} \times U$
      end if
    end if
  end for
end for
$\Delta'_r \leftarrow \Delta_{sync} \bigcup \Delta_{vert} \bigcup \Delta_{hori}$
return $\Delta'_r$
```

of the BPDS model, the resulting program actually corresponds to the PDS verification model, $\mathcal{P}'_r$; therefore, we can utilize existing model checkers to solve our reachability problems of BPDS. Chapter 6 will discuss the details regarding the implementation aspect of Algorithm 5.6.

5.6.2 Reduction Algorithm for LTL Checking

As discussed in Section 4.4, a symbolic LPDS rule, $\langle g, \gamma \rangle \overset{\tau}{\underset{R}{\hookrightarrow}} \langle g', \omega \rangle$, describes a set of LPDS rules that are not only labeled by the same input symbol but also have the same state transition with respect to the control flow. The transition relation R describes a set of data-flow transitions with respect to the same control-flow transition. It is inefficient (also unnecessary) to specify an LTL property on both the control flow and the data flow of LPDS; otherwise, all symbolic BPDS rules can be visible (due to some visible data-flow transition). Without loss of generality, we assume that the labeling function L_φ is defined based on the BA states and the LPDS states that are only related to the control flow. Furthermore, we extend the function $VisProp$ to take symbolic BPDS rules as the input. Algorithm 5.7 and Algorithm 5.8 are the symbolic version of Algorithm 5.4 and Algorithm 5.5 respectively. We have the following observations:

- A symbolic BA transition rule describes a set of BA transitions. There may exist some visible BPDS rules that are constructed from such BA transitions. Assuming that we are not allowed to reduce any visible BPDS rule, in the worst case, if every symbolic BA transition rule describes some BA transitions that are horizontally visible, we will not be able to reduce any of the horizontal symbolic BPDS rules. This is the motivation for us to reduce visible BPDS rules based on how the property is specified.

- As illustrated in Figure 5.7, diagonal BPDS rules are irreducible only if the related horizontal and vertical BPDS rules are all visible.

Algorithm 5.7 ReducibleSymbolicBPDSRules($U \subseteq \delta, \mathcal{R} \subseteq \Delta$)

```
Require: $U$ and $\mathcal{R}$ are independent.
ReduceDiag ← FALSE, ReduceHori ← FALSE
Let $U = Q \times \{\sigma\} \times Q$
    $\mathcal{R} = \langle g, \gamma \rangle \overset{\tau}{\underset{R}{\hookrightarrow}} \langle g', \omega \rangle$
    $\mathcal{R}_1 = \langle g, \gamma \rangle \underset{R'}{\hookrightarrow}{}_{\mathcal{BP}} \langle g, \gamma \rangle$, where $R' = R_{loop} \times U$ {Horizontal BPDS rules}
    $\mathcal{R}_2 = \langle g, \gamma \rangle \underset{R'}{\hookrightarrow}{}_{\mathcal{BP}} \langle g', \omega \rangle$, where $R' = R \times U_{loop}$ {Vertical BPDS rules}
    $\mathcal{R}_3 = \langle g, \gamma \rangle \underset{R'}{\hookrightarrow}{}_{\mathcal{BP}} \langle g', \omega \rangle$, where $R' = R \times U$ {Diagonal BPDS rules}
if $VisProp(\mathcal{R}_1) = \emptyset$ and $VisProp(\mathcal{R}_2) = \emptyset$ and $VisProp(\mathcal{R}_3) = \emptyset$ then
  {If $\mathcal{R}_1$, $\mathcal{R}_2$, and $\mathcal{R}_3$ are all invisible}
  ReduceDiag ← TRUE, ReduceHori ← TRUE
else
  if $VisProp(\mathcal{R}_1) = VisProp(\mathcal{R}_3)$ or $VisProp(\mathcal{R}_2) = VisProp(\mathcal{R}_3)$ or
      $VisProp(\mathcal{R}_1) = \emptyset$ or $VisProp(\mathcal{R}_2) = \emptyset$ then
    ReduceDiag ← TRUE
  end if
  if $\mathcal{R}_1$ is invisible or $U$ is horizontally visible then
    ReduceHori ← TRUE
  end if
end if
return (ReduceDiag, ReduceHori)
```

Algorithm 5.8 $\textsc{SymbolicBPDSRulesViaSPOR\_LTL}(\delta \times \Delta)$

```
$\Delta_{sync} \leftarrow \emptyset$, $\Delta_{vert} \leftarrow \emptyset$, $\Delta_{hori} \leftarrow \emptyset$, $\Delta_{diag} \leftarrow \emptyset$
for all $\mathcal{R} = \langle g, \gamma \rangle \xhookrightarrow[R]{\tau} \langle g', \omega \rangle \subseteq \Delta$ do
    for all $U = Q \times \{\sigma\} \times Q \subseteq \delta$ and $\sigma \subseteq L'_{\mathcal{P}2\mathcal{B}}(\langle g, \gamma \rangle)$ and $\tau \subseteq L'_{\mathcal{B}2\mathcal{P}}(U)$ do
        if $\mathcal{R}$ and $U$ are dependent then
            {$\mathcal{B}$ and $\mathcal{P}$ must transition together}
            $\Delta_{sync} \leftarrow \Delta_{sync} \bigcup \langle g, \gamma \rangle \xhookrightarrow[R']{}_{\mathcal{BP}} \langle g', \omega \rangle$, where $R' = R \times U$
        else
            {$\mathcal{P}$ transitions and $\mathcal{B}$ self-loops}
            $\Delta_{vert} \leftarrow \Delta_{vert} \bigcup \langle g, \gamma \rangle \xhookrightarrow[R']{}_{\mathcal{BP}} \langle g', \omega \rangle$, where $R' = R \times U_{loop}$
            $(ReduceDiag, ReduceHori) \leftarrow \textsc{ReducibleBPDSRules}(U, \mathcal{R})$
            if $ReduceDiag = \text{FALSE}$ then
                {$\mathcal{B}$ and $\mathcal{P}$ transition together}
                $\Delta_{diag} \leftarrow \Delta_{diag} \bigcup \langle g, \gamma \rangle \xhookrightarrow[R']{}_{\mathcal{BP}} \langle g', \omega \rangle$, where $R' = R \times U$
            end if
            if $ReduceHori = \text{FALSE}$ or
                $\langle g, \gamma \rangle \in SensitiveSet \bigcup VisibleSet \bigcup LoopSet$ then
                {$\mathcal{B}$ transitions and $\mathcal{P}$ self-loops}
                $\Delta_{hori} \leftarrow \Delta_{hori} \bigcup \langle g, \gamma \rangle \xhookrightarrow[R']{}_{\mathcal{BP}} \langle g, \gamma \rangle$, where $R' = R_{loop} \times U$
            end if
        end if
    end for
end for
$\Delta'_r \leftarrow \Delta_{sync} \bigcup \Delta_{vert} \bigcup \Delta_{hori} \bigcup \Delta_{diag}$
return $\Delta'_r$
```

- The first type of irreducible diagonal BPDS rule, as demonstrated on the left side of Figure 5.7 requires the LTL properties being specified on an explicit BPDS configuration. However, in symbolic representations, properties are specified on control-flow locations, where each control-follow location corresponds to a set of BPDS configurations. If a vertical transition is visible, the related diagonal transition should also be visible. Therefore, such irreducible diagonal BPDS rules do not exist in symbolic representations.

- In practice, the second type of irreducible diagonal BPDS rule, as demonstrated on the right side of Figure 5.7, is reducible under certain conditions. The rule r_3 can be replaced by r_1 and r'_2 if

  1. $VisProp(r_1) \bigcup VisProp(r'_2) = VisProp(r_3)$; and
  2. the propositional variables respectively from $VisProp(r_1)$ and $VisProp(r'_2)$ do not occur in the same Boolean expression (i.e., excluding the temporal operators) of the LTL property.

  r_3 can be replaced by r_2 and r'_1 if

  1. $VisProp(r_2) \bigcup VisProp(r'_1) = VisProp(r_3)$; and
  2. the propositional variables respectively from $VisProp(r_2)$ and $VisProp(r'_1)$ do not occur in the same Boolean expression of the LTL property.

Chapter 6

IMPLEMENTATION

In our approach, hardware and software can be specified using different languages such as C or SystemC. These specification languages need to be converted to a uniform format acceptable by the model checking engine. A straightforward conversion preserves the state space of the specification; however, it usually suffers from the state explosion problem. Therefore, a counterexample-guided abstraction refinement process is commonly applied to alleviate this problem, where the process starts with a highly abstracted conversion and then asymptotically introduces more details to the abstraction based on infeasible counterexamples given by the model checking engine.

For safety property verification, counterexample-guided abstraction refinement has been widely applied to software implementations such as C programs. According to the discussions in Chapter 5, BPDS models can be specified in C/modelC programs and then converted into a C program for checking safety properties. Therefore, the SLAM verification engine can be utilized to solve our verification problems. SLAM accepts properties specified in SLIC, a property specification language designed for software. As for our co-verification framework, property specification on hardware behaviors also is desired. We demonstrate how SLIC can be adapted to specify hardware properties.

For liveness property verification, one major challenge is loop computation, i.e., checking whether or not there exists a loop in the design that may not terminate. Loop computation is often inefficient in counterexample-guided abstraction refinement, because a loop needs to be completely unrolled in order to check its termination properties. Such unrolling itself may not even terminate when the specification language has the power to describe a Turing machine. Therefore, verification of liveness properties on C programs requires different techniques than safety properties. In this chapter, we will discuss verification of liveness properties on BPDS models specified using Boolean programs. The general concept of our approach also is applicable to BPDS models specified using C/modelC programs when a liveness verification engine for C programs (such as Terminator [25]) is available.

Considering a hardware BA model and a software LPDS model, we discuss the implementation in three steps: First, we need to construct a BPDS model from the BA and LPDS models. Second, we want to specify the properties that should be observed on the target model in verification. Third, we should apply our reduction algorithms at a proper phase of our implementation so that the size of the BPDS model can be reduced with a low cost. The rest of this chapter is organized as follows: Section 6.1 and Section 6.2 discuss the implementation of reachability analysis and LTL checking for co-verification respectively. Section 6.3 discusses our co-verification tool, CoVer.

6.1 REACHABILITY ANALYSIS

As discussed in Chapter 4, a symbolic BPDS model can be constructed directly from a symbolic BA model and a symbolic LPDS model. In co-specification, we have designed a language, modelC, to formally specify the device behaviors from the view of a driver, namely a *formal device model* (also referred to as a *hardware interface model* [49]). A Formal Device Model (FDM) includes both the HW/SW interface specification and hardware specification (see Chapter 3). Conceptually, the hardware behaviors described by a FDM can be represented by a symbolic BA model, where a hardware transaction function describes a set of BA transitions labeled by the same input symbol. Our goal is to verify a driver implementation with its FDM, where the driver's C code can be considered as a symbolic LPDS model and an atomic software statement describes a set of LPDS rules labeled by the same input symbol. We can then construct a symbolic BPDS model from the symbolic BA model and the symbolic LPDS model. The Cartesian product is carried out via code instrumentation, i.e., instrument the driver's C code with the device's modelC code. The symbolic BPDS model, as the result of the product, is actually a C program with non-determinism and a Büchi constraint. We can safely drop the Büchi constraint, since it is irrelevant to reachability analysis. Therefore, the non-deterministic C program can be verified by the SLAM engine [4] for reachability properties.

SLAM takes SLIC [10] as the property specification language. Since SLIC was designed for software verification, the language constructs of SLIC mainly focus on the control flow of C programs. This is different from hardware designs, where the data flow is more interesting. We demonstrate that the properties regarding hardware behaviors can also be specified using the SLIC language within our co-verification framework.

A straightforward product of the BA and LPDS will construct BPDS rules that are unnecessary for reachability analysis. In Chapter 5, we presented a static partial order reduction algorithm to reduce BPDS rules while constructing the BPDS model. Since the reduction is applied during the compilation phase of co-verification, no modification is necessary to the model checker. This is very helpful in practice, because verification engines with industrial strength, such as SLAM, can thus be readily utilized.

6.1.1 Cartesian Product via Code Instrumentation

There are two types of BPDS rules. First, synchronous BPDS rules are constructed from dependent BA transitions and LPDS rules. Second, asynchronous BPDS rules are constructed from independent BA transitions and LPDS rules.

In co-specification, the synchronous BPDS rules are specified in HW/SW interface. For example, when a driver invokes a function, `WRITE_REGISTER_UCHAR` (see Figure 3.1), to update device interface registers, the corresponding hardware transaction functions, e.g., `atWritePortA`, are invoked subsequently[1]. This sequence of operations can be understood as a synchronous (a.k.a., dependent) transition of the driver and device, where the driver executes the function, `WRITE_REGISTER_UCHAR`, and at the same time the device executes the hardware transaction function, `atWritePortA`. In the other direction, when device raises an interrupt, the driver should invoke ISR to service the interrupt. The function `RunIsr`, illustrated in Figure 3.4, models such a process. There are two atomic blocks in `RunIsr`. The first atomic block checks the states of both device and driver to decide if an ISR should be invoked; and the second atomic block sets the device and driver to the proper state after the ISR returns. The two atomic blocks should be considered as two synchronous transitions of the device and driver.

With respect to asynchronous BPDS rules, there are three types of asynchronous transitions, i.e., BA transitions and LPDS self-loops, LPDS transitions and BA self-loops, and BA and LPDS transition together. The three types of asynchronous BPDS rules can be modeled as interleaved executions between the driver statements and the hardware transaction function of the hardware model.

Hardware instrumentation function. As illustrated in Figure 6.1, a hardware instrumentation function implements a non-deterministic loop to invoke `atRun_DIO` and `RunIsr` in sequence. If an interrupt is raised due to a hardware state transition by executing `atRun_DIO`, the context-switch to the ISR is modeled as a function call, where the execution switches back to the interrupted thread only after the ISR returns. This approach is correct to simulate the context-switches because

[1] In verification, the implementation of `WRITE_REGISTER_UCHAR` is replaced by the glue code illustrated in Figure 3.3. The replacement is carried out automatically by Static Driver Verifier (SDV) [4], the working environment of SLAM.

```
VOID HWInstr () {
    while( choice() ) { // Non-deterministic choices
        atRun_DIO();    // Run hardware transaction function
        RunIsr();    // If interrupt has been raised
    }
}
```

Figure 6.1: The hardware instrumentation function.

ISRs are relatively atomic to other driver routines.

Code instrumentation. We insert the hardware instrumentation function, `HWInstr`, after every atomic driver statement to construct asynchronous BPDS rules. The idea is based on the concept of relative atomicity as illustrated in Figure 3.7. The non-deterministic while-loop simulates the delays of either software or hardware, i.e., BA transitions and LPDS self-loops or LPDS transitions and BA self-loops. The situation when hardware (i.e., BA) and software (i.e., LPDS) transition together can be replaced by continuous executions of a driver statement and the hardware transaction function, `atRun_DIO`.

6.1.2 Specification of SLIC rules

Hardware and software are different in nature. When specifying the properties to be verified on hardware and/or software, their differences must be explored to ensure that the unique behaviors of hardware and software can be precisely captured. In general, hardware is data-flow-centric, where the state change of registers by hardware transactions is interested; software is control-flow-centric, where the execution sequences of program statements are interested. These design features must be considered in the property specifications of co-verification.

Control flow refers to the order in which individual statements, instructions, or

```
// InvalidRead: the driver should never complete an I/O read request using
// STATUS_SUCCESS without actually reading any data from the device.
// Declare the state variable used by this rule
state { enum { INIT, DPCSch } s = INIT; }

[ atReadPortA, atReadPortB, atReadPortC ].entry {
    halt; // Stop the current execution if any data is read from hardware
}
WdfInterruptQueueDpcForIsr.entry {
    s = DPCSch; // DPC is scheduled in the ISR
}
DioDpc.entry {
    // Stop the current execution if DPC is not scheduled in the ISR
    if ( s == INIT ) halt;
}
WdfRequestCompleteWithInformation.entry {
    // If the I/O request is completed with STATUS_SUCCESS but no data
    // is actually read, raise an alarm.
    if( (s == DPCSch) && ($2 == STATUS_SUCCESS) )
        abort "Input request is successfully completed with no read operation.";
}
```

Figure 6.2: The SLIC rule `InvalidRead` for the PIO-24 digital I/O card driver. The driver source code is discussed in Section 2.5. We implemented a test harness, as illustrated in Figure 6.3, to model the OS environment on how the driver should be called.

function calls of an imperative or a declarative program are executed or evaluated. The SLIC language allows temporal properties to be specified on the order of function calls/returns. Commonly, there are two types of events that can be specified on a function: `entry` and `exit`. The two events identify the program points in the function immediately before its first statement and immediately before it returns control to the caller. Meanwhile, the states of the program can be specified by referring to function parameters and global variables at the events. The value of the n^{th} formal parameter in a function is referred to as $\$n$. The return value of a function is referred to as `$return`.

Figure 6.2 illustrates an example of a SLIC rule that checks whether the driver will ever complete an I/O read request using `STATUS_SUCCESS` without actually reading any data from the device. The `halt` statement signals that the analysis of the current execution path should stop. We halt the verification when a port (A, B, or C) is read by the driver, which satisfies the rule immediately; otherwise when the function, `WdfRequestCompleteWithInformation`, is invoked with the second formal parameter equal to `STATUS_SUCCESS`, we raise an alarm using the statement, `abort`.

Figure 6.3 illustrates the test harness that models the OS environment for invoking the driver. Instead of directly invoking the dispatch routines that are provided by the PIO-24 digital I/O card driver, we invoke the role type functions. A role type function corresponds to those dispatch routines that service the same type of request in Windows. Such dispatch routines should have the same function type; however, they may be defined under different names in various driver implementations. Therefore, role type functions help us to attain the portability of verification over different driver implementations. The tool that matches role type functions to driver dispatch routines is provided by Static Driver Verifier (SDV) [4]. For example, the role type function, `fun_WDF_IO_QUEUE_IO_DEVICE_CONTROL`, corresponds to the dispatch routine, `DioEvtDeviceControl`, in the PIO-24 driver and the role

```
void main() {
    // Non-deterministically invoke the role type functions for different requests
    switch( choice() ) {
        case 0: fun_WDF_IO_QUEUE_IO_READ( . . . ); break;
        case 1: fun_WDF_IO_QUEUE_IO_WRITE( . . . ); break;
        default: fun_WDF_IO_QUEUE_IO_DEVICE_CONTROL( . . . ); break;
    }
    // Invoke DPC to complete the request
    fun_WDF_DPC(. . . );
}
```

Figure 6.3: The test harness for `InvalidRead`.

type function, `fun_WDF_DPC`, corresponds to the dispatch routine, `DioDpc`, in the driver. As an execution scenario, the two dispatch routines `DioEvtDeviceControl` and `DioDpc` can be invoked in sequence by the harness. Meanwhile, the hardware instrumentation function, `HWInstr`, is invoked after every driver statement due to the code instrumentation. If the device model raises an interrupt, the ISR routine, `DioIsr`, will be invoked via `HWInstr`. At last, although the test harness invokes the DPC routine, `DioDpc`, it will not be executed in verification unless `DioIsr` has requested for a DPC routine (at the line `P3` of Figure 2.8b). This is guaranteed by the SLIC rule, `InvalidRead`, where the rule halts verification at the entry of `DioDpc` if no DPC routine was requested by `DioIsr`.

Data flow refers to the order in which the values of variables are changed. For example, in a clock-driven hardware design, the values of registers are updated along with every clock cycle. How the registers should be updated depends on the current state of the registers and the design of the hardware (i.e., the transition rules specified). Property specifications based on data flow usually monitor the changes of register values along with clock cycles. In our approach, clock cycles are abstracted away. Instead, we use hardware transaction functions to describe the state transition rules of hardware. Because hardware transaction functions are atomic in the view of software, we do not need to monitor the intermediate hardware state within a transaction. Because software cannot directly update the hardware state without going through hardware transaction functions, we do not need to monitor the hardware state when a program statement is executed. As a result, we monitor the hardware states at the exits of hardware transaction functions, because conceptually these exit events occur when hardware states are updated. Figure 6.4 illustrates a SLIC rule that checks whether the hardware model of the PIO-24 digital I/O card will ever raise an interrupt when its interrupt status is disabled. This rule is useful to validate the correctness of our formal device/driver models in co-specification.

```
// InvalidHWInterrupt: formal device model should not raise an interrupt when
// it is in an interrupt disabled state.

// Check the hardware state at the exit of a hardware transaction function
atRun_DIO.exit {
    // If hardware raises an interrupt when it is in an interrupt disabled state
    if( ($g_DIORegs.IRQST.IRQST1==1) && ($g_DIORegs.IRQ.IRQENn!=1) ) {
        abort "Interrupt is raised when the Interrupt Enable (IE) register is 0.";
}
```

Figure 6.4: The SLIC rule `InvalidHWInterrupt` for the PIO-24 digital I/O card device model.

6.1.3 Reduction

In Chapter 5, we demonstrated an approach to efficient reachability analysis of BPDS models. The process that reduces a BPDS model $\mathcal{BP}$ is presented in Algorithm 5.3 and the symbolic version is presented in Algorithm 5.6. As the key idea of the reduction, we observe that the BA $\mathcal{B}$ and the LPDS $\mathcal{P}$ can run separately when their state transitions are independent. This allows the reduction of many transition rules of $\mathcal{BP}$ without affecting the verification result. Following the concept of static partial order reduction, these reducible transition rules need not be included when constructing the BPDS model.

Software synchronization points. With respect to static partial order reduction, a key concept is *SensitiveSet*, defined to identify the BPDS rules that are necessary in reachability analysis. As the concrete counterpart of the *SensitiveSet* concept in implementation, 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;

3. those points where hardware interrupts may affect the software execution.

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. For example, in Figure 2.8b, the program reads hardware interface registers by `READ_REGISTER_UCHAR`. There is a

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

software synchronization point right after the function call. There is another software synchronization point right before the statement `P1` of Figure 2.8a, because a global variable `CurrentRequest` is accessed in the previous statement.

To construct the reduced BPDS model, $\mathcal{BP}_r$, according to Algorithm 5.6, we instrument the driver code by `HWInstr` in such a way that `HWInstr` is invoked at every software synchronization point. Conceptually, the instrumentation lets hardware run after every HW/SW synchronous transition. Compared to the trivial approach that inserts `HWInstr` after every software statement to simulate the concurrent state transitions of hardware and software, our algorithm can significantly reduce the complexity of the verification model, because the number of software synchronization points are usually very small in common applications.

6.2 LTL CHECKING

We have implemented the LTL checking algorithm for BPDS, where the LPDS $\mathcal{P}$ is specified using Boolean programs and the BA $\mathcal{B}$ is specified using Boolean programs with the semantic extension of relative atomicity, i.e., hardware transitions are modeled as atomic to software. In this section, we first present an example of a BPDS model specified in Boolean programs. Second, we illustrate how we specify LTL properties on such a BPDS model. Third, we elaborate on how we generate a reduced BPDS model for the verification of an LTL$_{-X}$ formula.

6.2.1 A BPDS Model specified using Boolean programs

We specify $\mathcal{B}$ and $\mathcal{P}$ using our co-specification approach as described in Chapter 3. Figure 6.5 demonstrates such an example. The states of $\mathcal{B}$ are represented by global variables. All the functions labeled by the keyword `__atomic` are hardware transaction functions that describe the state transitions of $\mathcal{B}$. The function `main` is the program entry of $\mathcal{P}$, where `main` has three steps:

```
void main() begin
    decl v0,v1,v2 := 1,1,1;
    reset();
    // wait for the reset to complete
    v1,v0 := status();
    while(!v1|v0) do v1,v0 := status(); od
    // wait for the counter to increase
    v2,v1,v0 := rd_reg();
    while(!v2) do v2,v1,v0 := rd_reg(); od
    // if the return value is valid
    if (v1|v0) then
        error: skip;
    fi
    exit: return;
end

__atomic void inc_reg()
begin
    if (!c0) then c0 := 1;
    elsif (!c1) then c1,c0 := 1,0;
    elsif (!c2) then
        c2,c1,c0 := 1,0,0; fi
end
```

```
// represent hardware registers
decl c0, c1, c2, r, s;

__atomic void reset()
begin reset_cmd: r := 1; end

__atomic bool<3> rd_reg()
begin return c2,c1,c0; end

__atomic bool<2> status()
begin return s,r; end

// hardware instrumentation function
void HWInstr() begin
    while(∗) do HWModel(); od
end

// asynchronous hardware model
__atomic void HWModel() begin
    if (r) then
      reset_act: c2,c1,c0,r,s := 0,0,0,0,1;
    elsif(s) then inc_reg(); fi
end
```

Figure 6.5: An example of BA $\mathcal{B}$ and LPDS $\mathcal{P}$ both specified in Boolean programs.

1. resets the state of $\mathcal{B}$ by invoking the function `reset`;
2. waits for the reset to complete;
3. waits for the counter of $\mathcal{B}$ to increase above 4, i.e., `v2==1`.

When a hardware transaction function, such as `reset` or `rd_reg`, is invoked from $\mathcal{P}$, it represents a dependent (a.k.a., synchronous) transition between $\mathcal{B}$ and $\mathcal{P}$. On the other hand, the hardware transaction function `HWModel` represents independent (a.k.a., asynchronous) transitions of $\mathcal{B}$ with respect to $\mathcal{P}$. In this example, since the dependent transitions of $\mathcal{B}$ and $\mathcal{P}$ are already specified as direct function calls, the rest of the Cartesian product is to instrument $\mathcal{P}$ with the independent transitions of $\mathcal{B}$, i.e., add function call to `HWInstr` after each statement in `main`. Such instrumentation only models two types of asynchronous BPDS rules when $\mathcal{B}$ transitions and $\mathcal{P}$ self-loops or $\mathcal{P}$ transitions and $\mathcal{B}$ self-loops. The BPDS rules when $\mathcal{B}$ and $\mathcal{P}$ transition together are not directly modeled by code instrumentation. Sometimes, these types of BPDS rules are not necessary to the checked LTL property; therefore, they can be replaced by interleaved transitions of $\mathcal{B}$ and $\mathcal{P}$ (Note that the transition order between $\mathcal{B}$ and $\mathcal{P}$ does not matter here). Otherwise, when these types of BPDS rules may affect the LTL property, we need to apply some restrictions on how the propositional variables are evaluated during verification, in order to reduce the BPDS rules. The next sub-section will discuss the details about what LTL properties require the asynchronous BPDS rules for $\mathcal{B}$ and $\mathcal{P}$ to transition together as well as how we can satisfy such requirements in verification.

6.2.2 Specification of LTL Properties

Without loss of generality, we specify LTL properties on the statement labels of Boolean programs. Formally, such labels are considered as propositional variables that evaluate to true at those BPDS configurations right after the execution of the labeled statements. For example, we write an LTL formula, ***F*** *exit*, which asserts that the function `main` always terminates. This property is asserted on a common scenario: when software waits for hardware to respond, the waiting thread should not hang. As illustrated in Figure 6.5, the hardware transaction function, `HWModel`, describes a hardware model that responds to software reset immediately; therefore, the first while-loop in `main` will not loop for ever. Since hardware increments its register after reset, the second while-loop in `main` also will terminate. Therefore, ***F*** *exit* holds. Note that the non-deterministic while-loop in `HWInstr` will repeatedly call `HWModel`, which is guaranteed by the BPDS loop constraint and the fairness between hardware state transitions (i.e., transitions specified by `HWModel` should not be starved by self-loop transitions introduced when constructing a BPDS).

There may exist a hardware design that cannot guarantee immediate responses to software reset commands. Therefore, delays should be represented in the hardware model. Figure 6.6 illustrates a hardware transaction function, `HWModelSlow`, which describes a hardware design that cannot guarantee immediate responses to reset commands. The property, ***F*** *exit*, fails on the BPDS model that uses `HWModelSlow` for hardware, since the hardware can delay the reset operation infinitely. In practice, design engineers may want to assume that hardware can delay the reset operation; therefore, software should wait for reset completion; however hardware should not delay the reset operation for ever. Such assumptions also

```
__atomic void HWModelSlow() begin
    if (r) then
        if (∗) then reset_act: c2,c1,c0,r,s := 0,0,0,0,1; fi
    elsif(s) then inc_reg(); fi
end
```

Figure 6.6: Hardware does not respond to reset immediately.

can be specified as LTL formulae. Under the assumption, ***G*** *(reset_cmd →* ***(F*** *reset_act))*, the property, ***F*** *exit*, will hold on the BPDS model.

As another example, we write an LTL formula, ***G*** *!error*, asserting that the labeled statement, `error`, in `main` is not reachable. The verification of ***G*** *!error* fails on the BPDS model in Figure 6.5. Since hardware is asynchronous with software when incrementing the register, it is impossible for software to control how fast the register is incremented. Therefore, when software breaks from the second while-loop, the hardware register may have already been incremented to 5, i.e., `(v2==1)&&(v1==0)&&(v0==1)`.

$\mathcal{B}$ and $\mathcal{P}$ transition together in an asynchronous BPDS rule. The property ***G*** *!(error* && *reset_ack)* specifies that BPDS does not contain states such that the hardware model acknowledges the reset command at the same time that $\mathcal{P}$ is executing the software statement labeled by `error`. Despite the usefulness of such kind of rule, they put a requirement on how the propositional variables should be evaluated during verification. In this case, the BPDS rules for $\mathcal{B}$ and $\mathcal{P}$ to transition together cannot be easily reduced; otherwise, the propositional variables, $error$ and $reset\_ack$, will not be evaluated as true at the same time. To solve this problem, we let the propositional variable, $error$, stay true when $\mathcal{B}$ executes and $\mathcal{P}$ self-loops; therefore, $error$ and $reset\_ack$ can be evaluated as true at the same time even when the asynchronous BPDS rules for $\mathcal{B}$ and $\mathcal{P}$ to transition together are reduced.

6.2.3 Reduction

In order to construct the Cartesian product of $\mathcal{B}$ and $\mathcal{P}$, we need to add a function call to `HWInstr` after every software statement. As discussed in Chapter 5, some BPDS rules are unnecessary to be generated for such a product. In other words, it is only necessary to call `HWInstr` after certain software statements in order to verify an LTL$_{-X}$ property. There are three types of program locations of $\mathcal{P}$ necessary for instrumentation. Except for the *software synchronization points* as defined in Section 6.1, we define the other two types of program locations:

Software visible points. Corresponding to $VisibleSet$, we define software visible points as a set of program locations right after the program statements whose labels are used in the LTL property. For example, in Figure 6.5 the program location right after the statement, *error*, can be a software visible point. However, the location right after the statement, *reset_act*, cannot be a software visible point, since this statement is in a hardware transaction function of $\mathcal{B}$.

Software loop points. Corresponding to *LoopSet*, we define software loop points as a set of program locations involved in program loops. The precise detection of those loops needs to explore the program's state graph, which is inefficient. Therefore, we try to identify a super set $LoopSet' \supseteq LoopSet$ using heuristics. A program location is included into the super set if it is at (1) the point right before the first statement of a while loop; (2) the point right before a backward goto statement; or (3) the entry of a recursive function, which can be detected by analyzing the call graph between functions.

As for implementation, we first detect the software synchronization points, visible points, and loop points in the Boolean program of $\mathcal{P}$ and then inserts function calls to `HWInstr` only at those detected points. Conceptually, the instrumentation lets hardware run for all the possibilities at those instrumentation points. Note that some transitions described by `HWModel` (called via `HWInstr`) may be visible when a statement label in `HWModel` is used in the LTL formula, e.g., ***F*** *reset_act*. However, such BA transitions are horizontally visible, since `reset_act` is not affected by any transition of $\mathcal{P}$. This is why function calls to `HWInstr` can be reduced without affecting the LTL properties even if `HWModel` describes visible transitions. Compared to the trivial approach that inserts `HWInstr` after every software statement to simulate the concurrent state transitions of hardware and software, our reduction can significantly reduce the complexity of the model to be verified, since the number of the instrumentation points are usually very small in common applications.

6.3 CO-VERIFICATION TOOL, COVER

We have created a co-verification tool, CoVer, which provides two options for reachability analysis and LTL property verification respectively.

Reachability analysis. Figure 6.7 illustrates the implementation for reachability analysis. CoVer has two steps. First, the frontend automatically instruments the driver with the formal device model to generate the verification model, a C program. Static partial order reduction is applied during this step in order to reduce function calls to the hardware instrumentation function, `HWInstr`. Second, the SLAM engine checks the reachability property (in the form of a SLIC rule) on the C program. As proven in Chapter 5, the reachability properties satisfied/disatisfied on the verification model will also be satisfied/disatisfied on the original device/driver model.

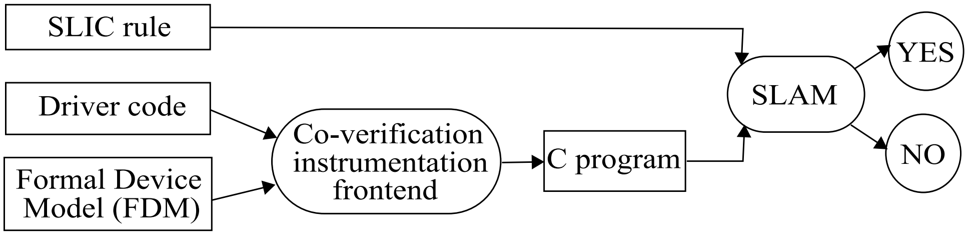

SLIC rule
Driver code
Formal Device Model (FDM)
Co-verification instrumentation frontend
C program
SLAM
YES
NO

Figure 6.7: CoVer implementation for reachability analysis.

It is important to note that our approach is not restricted by the verification engine, SLAM. Any verification engine that supports: (1) the verification of C programs; (2) non-determinism; and (3) property specification languages similar to SLIC, can be readily utilized in our co-verification approach.

LTL property verification. As illustrated in Figure 6.8, we have realized the LTL checking algorithm for BPDS as well as the static partial order reduction algorithm in our co-verification tool, CoVer. The implementation is based on the Moped model checker [77]. CoVer takes three inputs: First, the LTL assertions and assumptions. Second, the software LPDS model specified using Boolean programs. Third, the hardware BA model specified using Boolean programs with relative atomicity.

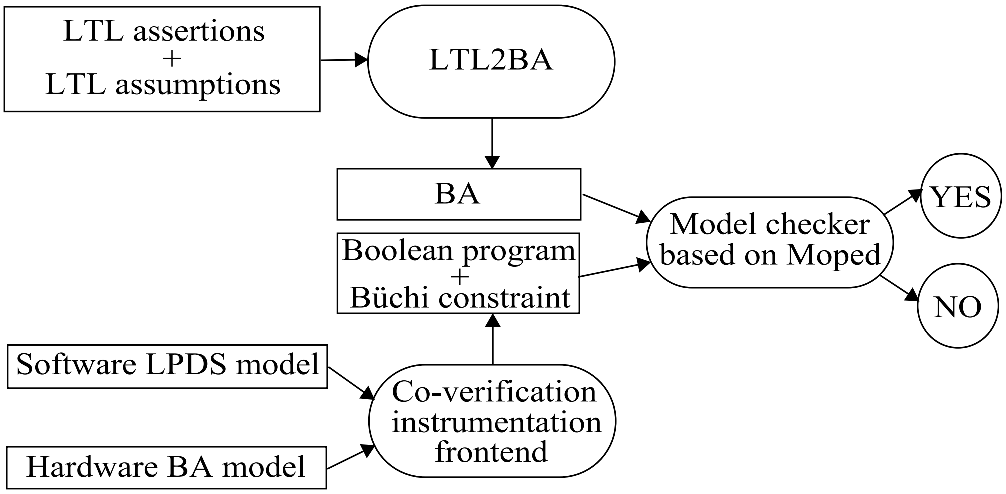


Figure 6.8: CoVer implementation for LTL checking.

There are three steps in verification: First, the LTL formulae are converted into a BA using the LTL2BA tool [31]. Second, the software LPDS model is instrumented with the hardware BA model to generate a Boolean program with the Büchi constraint. Third, this Boolean program is verified for the LTL formulae using the model checker implemented based on Moped. The static partial order reduction is implemented in the second step, and the Moped model checker is extended in order to support the BPDS loop constraint.

Chapter 7

EVALUATION

In practice, our approach has two phases: First, we need to formally specify the HW/SW interface protocols, i.e., co-specification. Second, we can utilize the formal models, as constructed in the co-specification process, in co-verification of driver implementations. Since our specifications closely resemble the implementation semantics of HW/SW interfaces, the formal models can be used, without any modification, as the test harness in co-verification. When a formal model is used as the test harness for a driver implementation, we refer to such a test harness as a Formal Device Model (FDM), because it describes the device behaviors in the view of the driver.

We applied our approach to four device/driver frameworks. One of the device/driver frameworks is still under development, while the other three have existed for many years. Following the mechanized process presented in Chapter 3, we constructed four formal models from the English documents of the device/driver frameworks. Although the quality of the English documents varies, the formal models are specified under the same criteria. For example, hardware behaviors visible to software should be clearly specified, and vice versa. We also applied automatic tools, such as CoVer, to validate our formal models. This is a significant benefit of formal models, because they can be analyzed by automatic tools. In total, there are **fifteen specification issues** in the English documents discovered during our formal specification process. Such specification issues can mislead development engineers and cause product failures. Given the fact that some of the English documents have existed for many years and been revised several times, our formalization approach is rather effective.

Co-verification is evaluated in reachability analysis and LTL checking respectively. For reachability analysis, CoVer is able to co-verify driver implementations with their FDMs. Both the driver implementations and the FDMs are directly used without any modification. There are five Windows drivers developed for the four device/driver frameworks: one Microsoft in-house driver, one Open Systems Resources (OSR) sample driver published by OSR online [67], and three drivers published in Microsoft Windows Driver Kit (WDK) as the sample drivers [59, 61, 63]. Except for the Microsoft in-house driver, which is a prototype currently under development, all other drivers are fully functional and well tested; however, utilizing our co-verification tool, CoVer, we have still discovered **twelve real bugs**. All of these bugs, which could cause serious system failures including data loss, interrupt storm, device hang, etc., were previously unknown to the driver developers. For LTL checking, we have designed a synthetic BPDS template to generate BPDS models with various complexities. The template mimics the common scenarios of HW/SW interactions. The evaluation illustrates that our reduction algorithm is very effective in both reachability analysis and LTL checking. The average reduction of the verification cost is 70% in time usage and 30% in memory usage.

7.1 CO-SPECIFICATION

As discussed in Chapter 3, the development process of a device/driver framework contains three stages: design, development, and certification. We have applied our approach to the first two stages.

First, for the design stage, we have applied our approach to the next generation of a pervasively used industry standard. Our approach has led to the detection of five issues in the draft English HW/SW interface document. One of the issues is a spec-inconsistency in an algorithm pseudo-code that describes the hardware-side interface protocol. This finding has triggered a discussion between two companies who participated in the design of this HW/SW interface protocol. Our formal model has 4781 lines of modelC code that covers about 277 pages of the English document. Therefore, the Model-Doc ratio is 17.26, which indicates that the draft English document is considerably elaborate compared with the other case studies (see below). The Model-Doc ratio is an important criteria to compare the formal model with its document. Specifically, Model-Doc is the ratio between the size of the formal model and the size of the document portion that is actually modeled.

Second, for the development stage, we have applied our approach to three long-existing device/driver frameworks:

- the Sealevel PIO-24 digital I/O device/driver framework, a.k.a., PIO-24 [78];
- the Intel 8255x 10/100Mbps Ethernet controller device/driver framework, a.k.a., Ethernet controller [39]; and
- the USB 2.0 device/driver framework, a.k.a., USB 2.0 [23, 57].

Our HW/SW interface formalization process (i.e., co-specification) has led to the detection of ten issues in the English documents.

PIO-24 device/driver framework. We use two sets of tables to present the evaluation of our formalization process. Table 7.1 illustrates the overall statistics about the formalization for the PIO-24 device/driver framework. The statistics are gathered before and after the formalization respectively. We require the specification engineer[1] to give an estimation of the manual effort necessary for formalization, so that we can compare how well interface documents with different complexities can be handled by an engineer. We also take the specification engineer's experience into consideration, where three areas of the experience may largely affect the

[1] The author is the specification engineer in this dissertation research.

Table 7.1: Formalization of the PIO-24 device/driver framework.

Gathered before the formalization process	
HW/SW interface doc. (document) size (pages)	20
The portion of the doc. for the HW/SW interface protocol (pages)	10
The portion of the doc. that cannot be modeled (pages)	10
Specification engineer's experience in driver development (years)	2
Specification engineer's experience in hardware design (years)	1
Specification engineer's experience in formal verification (years)	3
Specification engineer estimated manual effort (person-day)	7
Gathered after the formalization process	
The actual manual effort (person-day)	3
Specification issues found in the English document	2
Size of the modelC code in formal model (lines)	773
Size of the comments in formal model (lines)	577
Model-Doc ratio as (lines of the modelC code)/(pages of the modeled doc.)	773/10 = 77.3

result of the formalization. Two specification issues have been discovered in the HW/SW interface document for the PIO-24 device/driver framework: one spec-inconsistency and one spec-incompleteness. Taking the spec-incompleteness issue as an example, the document does not mention the default value of the interrupt pending register (which is usually disabled by default in many English documents for HW/SW interface specifications); therefore, we assign a non-deterministic initialization value to this register in our formal specification. Coincidentally, the Windows driver of this device does not clear the interrupt pending register during the driver initialization. This uninitialized register affects the driver's interrupt handling process, which can lead to data loss (see rule ProperISR2 in Table 7.7 for

Table 7.2: Formal model of the PIO-24 device/driver framework. (Com.: comments, Doc.: document)

File name	# of lines		Doc.	Description
	Com.	Code	pages	
DIODefs.h	63	151	2	Data structures
DIO.c	210	192	1	Hardware transaction function
DIODrv.c	37	76	1	software-side protocol
Global∼.c	21	15	N/A	Global variables for both hardware and software models
DIORegs.c	146	270	3	Registers, HW/SW interface events
Environ∼.c	100	69	3	Simulate inputs to Port A, B, and C

more details about this driver bug). We consider this driver bug partially caused by the spec-incompleteness issue, because the document should at least warn driver developers that the interrupt pending register is not initialized by default.

Table 7.2 illustrates the detailed statistics about the formal model for the PIO-24 HW/SW interface protocol. The formal model, as implemented in six files, has 577 lines of comments and 773 lines of modelC code. This corresponds to 10 pages of the English document. In the form of comments, we have added references that point to the corresponding document positions; therefore, the formal model can be related back to the original document. The file "Global∼.c" defines all the global variables that represent hardware and software states; therefore, we are not able to determine the exact number of corresponding pages in the document.

Ethernet controller device/driver framework. The statistics about formalizing the Ethernet controller device/driver framework are presented in Table 7.3 and Table 7.4 respectively. Compared to the English document of the PIO-24 device/driver framework, the English document of Ethernet controller is more

Table 7.3: Formalization of the Ethernet controller device/driver framework.

Gathered before the formalization process	
HW/SW interface doc. (document) size (pages)	175
The portion of the doc. for the HW/SW interface protocol (pages)	136
The portion of the doc. that cannot be modeled (pages)	39
Specification engineer's experience in driver development (years)	2
Specification engineer's experience in hardware design (years)	1
Specification engineer's experience in formal verification (years)	3
Specification engineer estimated manual effort (person-day)	14
Gathered after the formalization process	
The actual manual effort (person-day)	21
Specification issues found in the English document	6
Size of the modelC code in formal model (lines)	2370
Size of the comments in formal model (lines)	1446
Model-Doc ratio as (lines of the modelC code)/(pages of the modeled doc.)	2370/136 = 17.43

elaborate. This can be inferred from the major difference between their Model-Doc ratios, where the Model-Doc ratio of PIO-24 is much higher. Because the semantics of formal models closely resemble the HW/SW implementation semantics, necessary details must be specified. Therefore, the size of formal models can be considered as a standard measurement of the HW/SW interface complexities. During our formalization process, we have detected six specification issues in the Ethernet controller English document. One example of the issues is already illustrated in Figure 3.8. Given that this document has been published for seven years and revised three times, we were surprised. We have also observed an interesting difference between the manual effort estimations: it is clear that engineers have a

Table 7.4: Formal model of the Ethernet controller device/driver framework. (Com.: comments, Doc.: document)

File name	**# of lines**		**Doc.**	**Description**
	Com.	**Code**	**pages**	
E100Defs.h	203	768	14	Data structures
E100.c	182	197	15	Hardware transaction function
E100Drv.c	48	182	9	software-side protocol
Global∼.c	20	15	N/A	Global variables for both hardware and software models
E100Regs.c	173	492	35	Registers, HW/SW interface events
Port.c	170	151	5	Handle software commands to PORT interface registers
CmdUnit.c	410	329	26	Process the Command Unit (CU)
RcvUnit.c	133	134	25	Process the Receive Unit (RU)
Environ∼.c	107	102	7	Simulate the inputs to the device

better control over English documents that are less complicated.

USB 2.0 device/driver framework. The USB 2.0 device/driver framework is different from the previous device/driver frameworks such as PIO-24 and Ethernet controller in the sense that USB 2.0 devices use the USB bus instead of the PCI bus. Therefore, their HW/SW interfaces are quite different. Nevertheless, our approach has also been successfully applied to the USB 2.0 device/driver framework. The statistics are presented in Table 7.5 and Table 7.6 respectively. The formal model has 2304 lines of modelC code, which corresponds to 60 pages of the USB 2.0 document [23] and 70 pages (by estimation) of the Microsoft online document [57]. Therefore, the Model-Doc ratio is 17.72. We have discovered two spec-incompleteness problems in the Microsoft online document. Windows

Table 7.5: Formalization of the USB 2.0 device/driver framework.

Gathered before the formalization process	
HW/SW interface doc. (document) size (pages)	650 + 120 = 770
The portion of the doc. for the HW/SW interface protocol (pages)	60 + 70 = 130
The portion of the doc. that cannot be modeled (pages)	640
Specification engineer's experience in driver development (years)	2
Specification engineer's experience in hardware design (years)	1
Specification engineer's experience in formal verification (years)	3
Specification engineer estimated manual effort (person-day)	16
Gathered after the formalization process	
The actual manual effort (person-day)	20
Specification issues found in the English document	2
Size of the modelC code in formal model (lines)	2304
Size of the comments in formal model (lines)	1016
Model-Doc ratio as (lines of the modelC code)/(pages of the modeled doc.)	2304/130 = 17.72

provides a set of programming interfaces for operating USB devices. However, some programming rules are not specified, which has confused driver developers. We have discovered such programming problems in driver implementations using CoVer. For example, one of the problems is caused by redundant function calls from driver to stop a USB device[2].

Because formal models are manually specified, it is impossible to guarantee that

[2] Note that such problems are not reported as bugs in co-verification statistics; however, they can also be considered as bugs in a stricter standard.

Table 7.6: Formal model of the USB 2.0 device/driver framework. (Com.: comments, Doc.: document)

File name	# of lines		Doc. pages	Description
	Com.	Code		
USBDef.h	52	128	20	Data structures
USB.c	186	178	20	Hardware transaction function
USBDrv.c	112	140	20	software-side protocol
Global∼.c	9	8	N/A	Global variables for both hardware and software models
wdfintfs.c	393	1394	50	Registers, HW/SW interface events
device.c	244	445	15	USB device state machine
Environ∼.c	20	11	5	Simulate the inputs to USB devices

no error is made by the specification engineer. However, we are able to validate our formal models using automatic tools. For example, a C compiler has helped discover quite a few specification inconsistencies in our formal models, because most inconsistencies fail the syntax/semantic checking right away. Furthermore, CoVer has helped discover thirteen errors in our formal models. The errors are mostly introduced by code copy-paste and misunderstandings of the English specifications. In our approach, the ability to utilize automatic tools in formal HW/SW interface specifications is a significant advantage over English specifications.

7.2 CO-VERIFICATION

Co-verification is evaluated in reachability analysis and LTL checking respectively, where real driver programs are verified in reachability analysis and synthetic BPDS models are used as the benchmark in LTL checking. All evaluation experiments run on a Lenovo ThinkPad notebook with Dual Core 2.66GHz CPU and 4GB memory.

The timeout threshold is set as 3000 seconds for both reachability analysis and LTL checking. For reachability analysis, the spaceout threshold is set as 2000MB, which is enforced by the SLAM engine. For LTL checking, the spaceout threshold is not explicitly specified, i.e., a maximum of 4000MB memory may be used.

7.2.1 Reachability Analysis

In reachability analysis, the properties to be verified can be classified into two categories:

1. whether a driver callback function[3] accesses the hardware interface registers in correct ways, e.g., a command should not be issued when hardware is busy;

2. whether a driver callback function can cause an out-of-synchronization between the driver and device. For example, we check if the return value of a driver callback function correctly indicates the current hardware state.

We have applied CoVer to co-verification of a Microsoft in-house driver with its FDM developed in co-specification. This in-house driver is a prototype with the functionalities partially implemented. However, CoVer can still be applied to analyze the implemented portion of the driver. As a result, two real bugs were discovered. This is an advantage over runtime validation where most functionalities of the driver need to be implemented before any comprehensive test can be conducted.

We have also applied CoVer to four fully functional Windows device drivers with their FDMs:

- OSR PIO-24 driver [67];

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

- Microsoft Ethernet controller driver [61];
- OSR USB 2.0 OSRUSBFX2 driver [59]; and
- Microsoft USB 2.0 USBSAMP driver [63].

Because the source code of the drivers has been provided to public as samples for years, we did not expect to find many bugs. However, utilizing CoVer, we discovered ten real bugs. All of these bugs, which could cause serious system failures including data loss, interrupt storm, device hang, etc., were previously unknown to the driver developers.

PIO-24 driver by OSR. Table 7.7 presents the statistics on the verification of the PIO-24 driver with its FDM. We discovered four bugs and proved two properties of the driver using CoVer. For example, the code excerpt in Figure 2.8

Table 7.7: Statistics on the co-verification of the PIO-24 device/driver.

Size of the driver (# of lines)	1724
Size of the formal device model (# of lines)	1237

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 the 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 device interrupt-pending status before return.	58.9	58	35.2	43	Failed
`ProperISR2`	ISR will not acknowledge the interrupt raised by other devices.	74.1	62	28.7	37	Failed

contains one bug, which violates the rule InvalidRead (illustrated in Figure 6.2) and will cause the driver return invalid data to user applications. This "invalid read" bug occurs when the ISR routine DioIsr interrupts the device driver control routine DioEvtDeviceControl at P1, where the variables CurrentRequest and AwaitingInt become inconsistent. DioIsr will not execute the if block at P2 because AwaitingInt is FALSE. Later the DPC routine DioDpc is requested at P3. After both DioIsr and DioEvtDeviceControl have returned, DioDpc starts to run. At P4, the data is read from PortAValueAInt which has never been written in DioIsr; therefore, the data is invalid. However, DioIsr still sends the invalid data back to user application with STATUS_SUCCESS at P5.

Another serious bug (discovered using the rule ProperISR1) of this driver can cause an interrupt storm. The design of the device allows interrupts being repeatedly generated in certain configuration; however the driver does not handle the interrupts correctly which will cause interrupts being raised more frequently than that can be consumed, i.e., interrupt storm. This bug also reveals a problem of the device document. Since the assumption on device input is not well defined in the document, our formal model has to simulate all possible input. On the other hand, the driver fails to handle one of the possibilities. As a solution to fix this bug, the driver can disable the interrupt in ISR first and re-enable it later after interrupt processing is completed.

Ethernet controller driver by Microsoft. Table 7.8 presents the statistics on the verification of the Intel 82557/82558 based Ethernet controller driver with its FDM. We discovered three bugs and proved five properties of the driver using CoVer. For example, CoVer helps discover 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

Table 7.8: Statistics on the co-verification of the Ethernet controller device/driver.

Size of the driver (# of lines)	14406
Size of the device formal model (# of lines)	3586

Rule	**Description**	**No reduction**		**Reduction**		**Result**
		Time	**Mem.**	**Time**	**Mem.**	
		(Sec)	(MB)	(Sec)	(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 the hardware registers.	64.1	99	39.9	79	Passed
`ProperISR1`	ISR will clear device interrupt-pending status before return.	48.9	59	32.6	52	Passed
`ProperISR2`	ISR will not acknowledge the interrupt raised 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

device to become 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 to become permanently unaccessible.

Another bug that violates the rule `DoubleCUC` is illustrated in Figure 3.16b, where the function `D100IssueScbCommand` waits before issuing a new command only if the function parameter `WaitForScb` is `TRUE`. This kind of design is due to a performance optimization. Since there are some program locations where the driver knows that the device command register is free, it is unnecessary to check

Table 7.9: Statistics on the co-verification of the USB 2.0 OSRUSBFX2 device/driver.

Size of the driver (# of lines)	2892
Size of the device formal model (# of lines)	3068

Rule	**Description**	**No reduction**		**Reduction**		**Result**
		Time (Sec)	**Mem.** (MB)	**Time** (Sec)	**Mem.** (MB)	
`StopIO`	I/O on interrupt pipe should be stopped during powering down	Timeout	N/A	2755.6	340	Passed
`ResetDevice`	All I/O on all pipes should be stopped before resetting the device.	318.0	150	126.2	82	Failed
`ResetPipe`	Driver must stop the pipe before resetting it.	0.9	28	0.6	28	Passed
`DevIORead`	A read request should fail if the device is in an invalid state.	221.4	133	54.6	60	Passed
`DevIOWrite`	A write request should fail if the device is in an invalid state.	200.3	132	87.6	71	Passed

the register before issuing a new command. However, CoVer has demonstrated that in some program execution path, a command is issued by the driver even when the device command register is busy. This is a typical example of performance optimization creating bugs. Since optimized code is often more complex than the original code, it very important to use automatic tools, such as CoVer, in order to ensure the correctness of the optimization.

USB 2.0 device drivers by Microsoft and OSR. Table 7.9 presents the statistics on co-verification of the OSR OSRUSBFX2 driver implementation. We discovered one real bug in this driver using CoVer. The SLIC rule `ResetDevice` checks that I/O on all pipes should be stopped before a resetting command; however, the driver fails to follow this rule in certain execution paths. As for the SLIC rule `ResetPipe`, the verification cost is very low. Because CoVer (actually, SLAM) decides that the error routine (i.e., function that contains the reachability label)

Table 7.10: Statistics on the co-verification of the USB 2.0 USBSAMP device/driver.

Size of the driver (# of lines)	3969
Size of the device formal model (# of lines)	3068

Rule	**Description**	**No reduction**		**Reduction**		**Result**
		Time (Sec)	**Mem.** (MB)	**Time** (Sec)	**Mem.** (MB)	
`StopIO`	I/O on interrupt pipe should be stopped during powering down	105.8	81	98.2	72	Passed
`ResetDevice`	All I/O on all pipes should be stopped before resetting the device.	200.1	100	110.3	65	Failed
`ResetPipe`	Driver must stop the pipe before resetting it.	54.2	51	31.6	40	Failed
`DevIORead`	A read request should fail if the device is in an invalid state.	70.1	63	38.4	48	Passed
`DevIOWrite`	A write request should fail if the device is in an invalid state.	68.5	63	34.7	48	Passed

is not reachable in the call graph of the instrumented program, verification stops with a rule pass right after compilation; therefore, no model checking is necessary for `ResetPipe`. Table 7.10 presents the statistics on co-verification of the Microsoft USBSAMP driver implementation. We discovered two real bugs in this driver using CoVer. Other than `ResetDevice`, the SLIC rule `ResetPipe` is also violated by USBSAMP driver, where the driver does not stop the I/O on a pipe before resetting. Such bug may cause data loss in I/O operations. Note that verification of the OSRUSBFX2 driver costs much more time and memory than that of the USBSAMP driver, because OSRUSBFX2 implements iterations on operating multiple USB device pipes. These iterations need to be fully unrolled in verification.

Effectiveness of reduction. We have also compared the differences of co-verification performance on whether our reduction algorithm is applied. It is clear that our reduction algorithm can significantly scale co-verification, especially when the target system is complex. For example, when no reduction is applied, there is only one non-useful result in Table 7.7, however half of the verification cannot complete in Table 7.8. This is because the Ethernet controller device/driver have more comprehensive functionalities and implementation than the PIO-24 device/driver.

7.2.2 LTL Checking

We designed a synthetic BPDS template `BPDS<N>` for $N > 0$ to evaluate our algorithms. As illustrated in Figure 7.1, this template is similar to the BPDS model in Figure 6.5. The major difference is between the models of $\mathcal{P}$. `BPDS<N>` has two function templates `level<N>` and `gcd<N>` for $\mathcal{P}$, where each of the function templates has N instances. For $0 < i \leq N$, `level<i>` calls `gcd<i>` which is the i^{th} instance of `gcd<N>` that computes the greatest common divisor (implementation of `gcd<N>` is omitted). For $0 < j < N$, the instance of `<stmt>` in the body of the function `level<j>` is replaced by a call to `level<j+1>`. The instance of `<stmt>` in the body of `level<N>` is replaced by `skip`. The design of `BPDS<N>` mimics the common scenarios in co-verification: since hardware and software are mostly asynchronous, there are many software statements independent with hardware transitions.

Table 7.11 presents the statistics for the verification of five LTL formulae on the BPDS models generated from `BPDS<N>`, where some of the LTL formulae are discussed as the examples in Chapter 6.

Table 7.12 presents the statistics for the verification of BPDS models generated from `BPDS_Slow<N>`, a template that differs from `BPDS<N>` only in the hardware model. `BPDS_Slow<N>` uses the hardware model illustrated in Figure 6.6. As discussed in Chapter 6, verification of the properties $\mathcal{A}1$ and $\mathcal{A}2$ will fail on the BPDS models generated from `BPDS_Slow<N>`, since the hardware cannot guarantee an immediate response to the software reset command. However, by assuming $\mathcal{A}2$, the

```
decl c0, c1, c2, r, s; // hardware registers
decl g; // software global variable
void main() begin
    decl v0,v1,v2 := 1,1,1;
    reset();
    v1,v0 := status();
    while(!v1|v0) do v1, v0 := status(); od

    // call the first level
    level<1>();

    v2,v1,v0 := rd_reg();
    while(!v2) do v2,v1,v0 := rd_reg(); od
    if (v1|v0) then error: skip; fi
    exit: return;
end
```

```
void level<i>()
begin
    decl v0,v1,v2,v3,v4,v5;
    v2,v1,v0 := rd_reg();
    v5,v4,v3 := rd_reg();
    v2,v1,v0 :=
      gcd<i>(v5,v4,v3,v2,v1,v0);

    if(*) then reset(); fi

    if(g) then
        g := (v3 != v0);
        <stmt>;
    fi
end
```

Figure 7.1: The BPDS template `BPDS<N>` for evaluation.

verification of $\mathcal{A}1$ should pass. Obviously, the verification of this property, denoted by φ (including both $\mathcal{A}1$ and $\mathcal{A}2$), costs more time and memory compared to other properties, because φ is more complex than other properties.

We can infer from the two tables that our reduction algorithm is very effective in reducing the verification cost. For example, without the reduction, verification of the property φ gets a spaceout failure for $N = 2000$, i.e., CoVer fails to allocate more memory from the Operating System. The statistics suggest that our reduction algorithm can reduce the verification cost by 80% in time usage and 35% in memory usage on average.

Table 7.11: Statistics on the LTL checking of `BPDS<N>`. (NoR.: No Reduction. Red.: Reduction)

LTL Property		***N***		
		500	**1000**	**2000**
F *exit*	**NoR.**	177.9sec/49.1MB	606.8sec/98.1MB	1951.5sec/196.3MB
	Red.	55.6sec/27.8MB	100.9sec/55.6MB	231.5sec/111.2MB
G(*reset_cmd* → (***F*** *reset_act*))	**NoR.**	100.8sec/51.1MB	439.0sec/102.1MB	1742.1sec/204.3MB
	Red.	19.2sec/31.6MB	37.2sec/63.2MB	115.0sec/126.5MB
F *level_N*	**NoR.**	165.3sec/49.1MB	524.1sec/98.1MB	1934.1sec/196.3MB
	Red.	52.9sec/27.8MB	99.8sec/55.6MB	230.7sec/111.2MB
G *!level_N*	**NoR.**	94.8sec/43.4MB	404.0sec/86.2MB	1728.9sec/172.5MB
	Red.	10.7sec/25.0MB	22.3sec/49.9MB	84.5sec/99.9MB
G *!error*	**NoR.**	96.6sec/42.4MB	402.6sec/84.8MB	1719.9sec/169.8MB
	Red.	10.1sec/24.8MB	21.2sec/49.2MB	81.5sec/98.5MB

7.3 SUMMARY

Summary of the bug discovery by co-verification. Consider the twelve bugs discovered using co-verification in Windows driver implementations:

- All the bugs involve interactions between drivers and devices.
- One bug happens when a driver does not initialize its device correctly, i.e., a default device state is not considered during the initialization process.
- Three bugs happen when devices interrupt their drivers. It is a restricted version of concurrency checking.
- Four bugs are due to the out-of-synchronization between drivers and devices. For example, a driver issues a command while its device is busy.

Table 7.12: Statistics on the LTL checking of `BPDS_Slow<N>` which uses the hardware model of Figure 6.6. (NoR.: No Reduction. Red.: Reduction)

LTL Property		***N***		
		500	**1000**	**2000**
$\mathcal{A}1$:***F*** *exit*	**NoR.**	186.5sec/49.1MB	576.4sec/98.1MB	1913.5sec/196.3MB
	Red.	38.1sec/27.8MB	98.5sec/55.6MB	207.1sec/111.2MB
$\mathcal{A}2$:***G****(reset_cmd → (**F** reset_act))*	**NoR.**	143.1sec/61.0MB	587.1sec/122.0MB	1778.7sec/203.5MB
	Red.	28.3sec/35.5MB	64.3sec/71.0MB	164.1sec/142.0MB
$\mathcal{A}1$ using $\mathcal{A}2$ as the assumption	**NoR.**	1264.0sec/223.4MB	3750.3sec/446.7MB	N/A/spaceout
	Red.	255.8sec/109.5MB	565.6sec/218.9MB	1260.8sec/437.7MB
F *level_N*	**NoR.**	181.9sec/49.1MB	588.6sec/98.1MB	1908.4sec/196.3MB
	Red.	42.2sec/27.8MB	90.8sec/55.6MB	198.6sec/111.2MB
G *!level_N*	**NoR.**	96.7sec/43.4MB	414.6sec/86.2MB	1679.7sec/172.5MB
	Red.	12.1sec/25.0MB	26.9sec/49.9MB	91.5sec/99.9MB
G *!error*	**NoR.**	95.0sec/42.5MB	414.2sec/84.8MB	1672.6sec/169.8MB
	Red.	11.5sec/24.8MB	25.3sec/49.2MB	88.9sec/98.5MB

- Four bugs happen when drivers mishandle their device failures. For example, a driver returns SUCCESS when its device actually fails.

Summary of evaluation. We have five observations through evaluation:

- First, our co-specification approach is very effective in detecting the specification issues of English documents. We have discovered **fifteen specification issues** in four English documents, where some of the issues have existed for many years.

- Second, the formal models developed in co-specification can precisely capture the HW/SW interface behaviors regardless of the English documents' quality.

- Third, the correctness of the formal models can be easily analyzed by automatic tools.

- Fourth, our co-verification algorithm is effective in discovering sophisticated bugs of HW/SW interface implementations in driver programs. Utilizing CoVer, we have discovered **twelve real bugs** in five Windows driver implementations. All these bugs are previously unknown to driver developers, even after comprehensive testing.

- Fifth, our reduction algorithm is efficient in alleviating the verification cost. For both reachability analysis and LTL checking, the average reduction of the verification cost is 70% in time usage and 30% in memory usage.

Chapter 8

CONCLUSION AND FUTURE RESEARCH

8.1 CONCLUSION

HW/SW interfaces exist in all kinds of computer systems ranging from embedded systems to personal computers. These systems are often expected to be reliable. However, the intrinsic complexity in HW/SW interface designs have always been a challenge to this goal. It is challenging to specify HW/SW interface protocols in a manner that is clear and precise to both hardware and software engineers; it is challenging to synthesize a unifying formal model for HW/SW interfaces, since hardware and software have different implementation semantics; it is also challenging to develop effective tools for HW/SW co-verification, where the design features of HW/SW interfaces are well exploited.

Throughout this dissertation, we have demonstrated that co-verification of HW/SW interface protocols can be effectively achieved via formal specification and model checking.

Co-specification. It is possible to formally specify HW/SW interface protocols in such a way that closely resembles the implementation semantics of hardware and software. Our specification language, modelC, is designed based on the C semantics with three restrictions to achieve finite state and two extensions to support non-determinism and relative atomicity. The hardware behaviors are specified using TLM, a common approach in hardware specification. In order to specify a hardware-side interface protocol in modelC, one should model the hardware states using global variables; and describe the hardware behaviors using hardware transaction functions. A hardware transaction function is an atomic C function that describes the transition rule with respect to the state change of (hardware) global variables. The concurrency in a hardware design is modeled via interleaved executions of hardware transaction functions and non-deterministic choices made on the control flow of each hardware transaction function. On the other hand, it is straightforward to specify a software-side interface protocol using modelC. Different from hardware, software states are maintained by both (software) global variables and local variables. In software specification, an atomic program statement describes a set of software state transitions.

Except for the purpose of co-verification, formal models constructed by co-specification can also be utilized in the development process of devices and drivers, as the formal HW/SW interface specifications. Compared with English specifications, formal models are clear, precise, and easy for development engineers to understand. Furthermore, formal models can serve as the basis of a uniform platform for co-verification, co-simulation, conformance testing, etc. Section 8.2 will discuss how to apply the formal models to co-simulation and conformance testing respectively.

Co-specification is very effective to help identify specification issues of HW/SW interface protocols. As discussed in Chapter 7, the formalization process of four device/driver interface protocols has led to the detection of fifteen specification issues, given the fact that some of the specifications have existed as the industry standards for many years.

Co-verification model. BPDS is a suitable formal model for HW/SW interfaces. A BPDS model is the Cartesian product of a BA and an LPDS, where BA is a suitable representation for hardware which is finite state; and LPDS is a suitable representation for software which is often infinite state. The input alphabets of BA and LPDS are induced on the states of each other, so that BA transitions and LPDS rules can be combined into BPDS rules. BPDS has a synchronous execution mode, i.e., both the BA and LPDS must transition at the same time in order to make one BPDS transition. In synchronous execution mode, it is straightforward to model the situation when hardware and software transition simultaneously. However, they may also be interleaving, which is modeled by introducing self-loop transitions to both BA and LPDS.

Co-verification algorithms. The verification problem of BPDS for either safety properties or liveness properties is solvable in cubic time and space with respect to the size of the BPDS model and the property to be checked. For reachability analysis (i.e., safety properties only), BPDS is converted into PDS so that existing model checkers for PDS can be readily utilized to solve the problem. For LTL checking (including safety properties and liveness properties), an LTL property is first negated and then represented as a BA. The BA is combined with BPDS in such a way that the BA monitors the state transitions of the BPDS. The LTL property fails if the BA has an accepting run on the BPDS; otherwise, the LTL property passes.

The verification cost can be greatly alleviated via reducing the size of BPDS. Since hardware and software are mostly asynchronous, their transition orders are often unnecessary to be explored during verification. Therefore, many BPDS rules can be pruned in the compilation phrase while constructing a BPDS from a BA and an LPDS. Such reduction is very useful in practice, since it does not require any modification to the model checker. Therefore, model checkers with industry strength, such as SLAM, can be readily utilized. Interestingly, our reduction algorithm is also useful as the formal foundation for those reductions applied with runtime techniques. For example, the reduction method used in Device Driver Tester (DDT) [46] is actually one kind of static partial order reduction for HW/SW interfaces. The motivation and correctness of such reduction have been thoroughly discussed in this dissertation.

Our co-verification tool, CoVer, has been applied to five Windows drivers with their Formal Device Models (FDMs). Some of the drivers are fully functional, well tested, and used as sample drivers for many years. However, utilizing CoVer, we have still discovered real bugs in each of the drivers and the total bug count is twelve. All of these bugs, which could cause serious system failures including data loss, interrupt storm, device hang, etc., were previously unknown to the driver developers. Furthermore, evaluation suggests that the average reduction of verification cost is 70% in time usage and 30% in memory usage.

8.2 FUTURE RESEARCH

This dissertation has presented a useful approach to improve the reliability of HW/SW interface implementations; however, it is only the tip of the iceberg. There are other interesting research that needs to be explored.

8.2.1 Co-verification of Liveness Properties on Driver Code

We demonstrate the verification of liveness properties on BPDS models specified using Boolean programs. In practice, it is desired that co-verification of liveness properties can be applied to driver implementations. For example, developers may want to know whether their drivers may hang on device operations. Co-verification of liveness properties on driver implementations can be realized based on liveness verification engines for C programs, such as Terminator [24, 25].

As illustrated in Figure 8.1, given a liveness property, a driver implementation, and a FDM, we can implement a co-verification frontend that converts the input into a C program with some liveness constraints, where the idea of the conversion is presented in Algorithm 5.8. Therefore, the verification problem can be solved

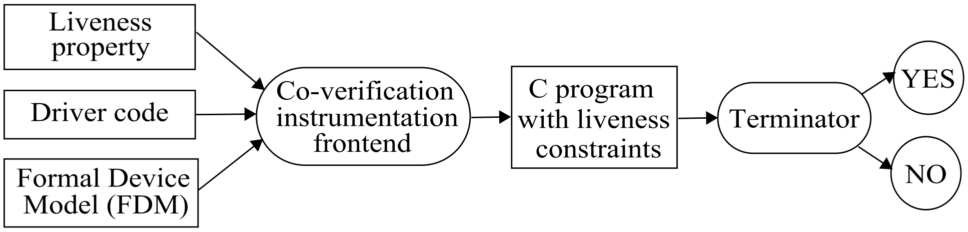

Liveness property
Driver code
Formal Device Model (FDM)
Co-verification instrumentation frontend
C program with liveness constraints
Terminator
YES
NO

Figure 8.1: Co-verification of liveness properties on driver implementations.

by Terminator.

8.2.2 Co-simulation

Although we can discover sophisticated bugs using co-verification, co-simulation, i.e., simulating a driver with its device model, is also highly desired in practice. Simulation can help discover shallow bugs with a low cost and is often used to evaluate the efficiency of implementations.

As illustrated in Figure 8.2, a FDM constructed by co-specification can also be used in co-simulation, where the *FDM interface* is a thin layer that adapts

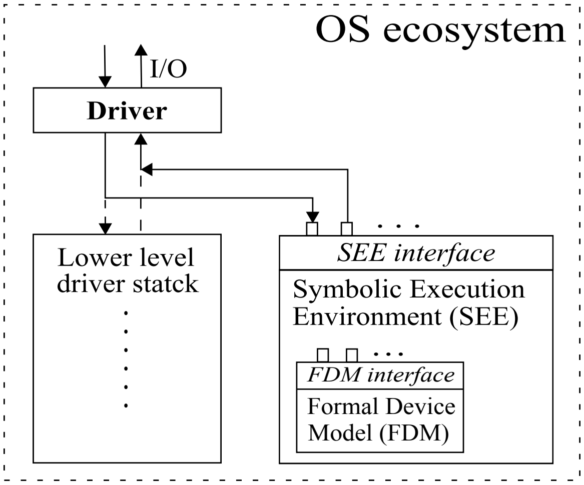

OS ecosystem
I/O
Driver
Lower level driver statck
SEE interface
Symbolic Execution Environment (SEE)
FDM interface
Formal Device Model (FDM)

Figure 8.2: Co-simulation using formal device model.

the interface of the FDM to simulation environment. One major challenge to co-simulation is how to support relative atomicity and non-determinism without changing the FDM. We need to implement two modules: a *Symbolic Execution Environment (SEE)* and a *SEE interface*.

Symbolic Execution Environment executes a FDM via the FDM interface. Note that non-determinism can be easily supported by symbolic execution.

SEE interface has three functions:

- First, it intercepts the communications between the driver and its underline stack in order to reroute the I/O to SEE.
- Second, it ensures the relative atomicity. For example, hardware transaction functions should be atomic to each other; and some driver operations such as kernel API calls should be atomic to hardware transaction functions.
- Third, it serves as the boundary between symbolic execution and concrete execution, i.e., how a symbolic value can be passed to concrete system environment; and how a concrete system call can be translated into symbolic values.

Although related work can be found in DDT [46] with a technique called *selective symbolic execution* [18], the challenges are still open on how to simulate a comprehensive FDM (instead of a shallow symbolic device model used in DDT); how to ensure that the interface states of the FDM are always consistent in the view of the driver, which is not guaranteed by selective symbolic execution; and how to optimize the simulation since symbolic execution also suffers from the state explosion problem.

8.2.3 Co-monitoring

An approach to protocol conformance validation is monitoring, where the behaviors of a system is observed and compared to the golden model that describes the protocol. With respect to HW/SW interfaces, the behaviors of a device and its driver should be monitored together, i.e., co-monitoring. The formal model developed in co-specification can be used as the golden model for co-monitoring.

Figure 8.3 illustrates the framework of co-monitoring, which is different from co-simulation in four aspects:

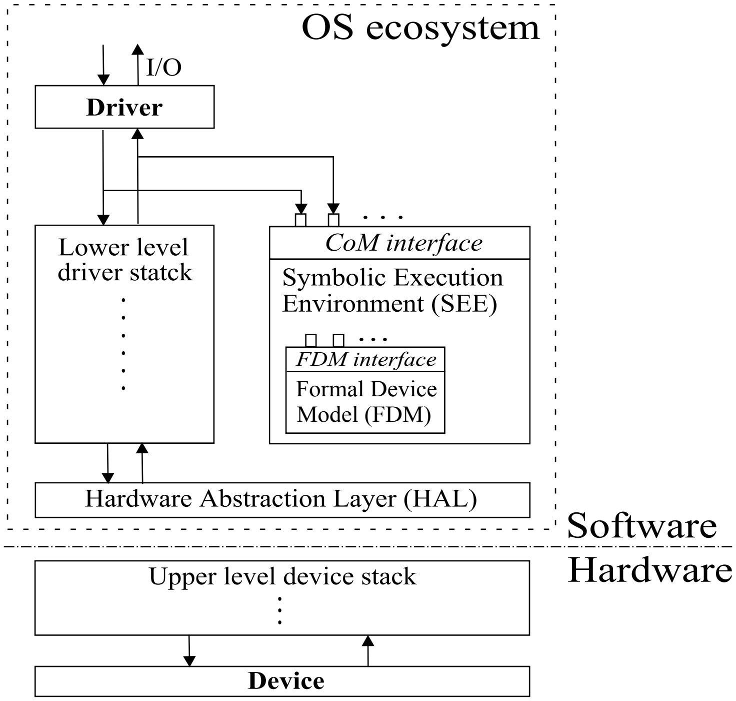

OS ecosystem
I/O
Driver
Lower level driver statck
CoM interface
Symbolic Execution Environment (SEE)
FDM interface
Formal Device Model (FDM)
Hardware Abstraction Layer (HAL)
Software
Hardware
Upper level device stack
Device

Figure 8.3: Co-monitoring using formal device model.

- First, there is a real hardware device interacting with the driver.
- Second, it only monitors the communications between the driver and the lower level driver stack; therefore, the communications should be affected as little as possible.
- Third, it symbolically executes the FDM according to the monitored communications, which makes the FDM a mirror of the device with respect to their states.

- Fourth, it raises an alarm when a protocol violation is detected.

One key part of co-monitoring is the *CoM (Co-Monitoring) interface*, which serves three functions:

- First, it monitors the communications between the driver and its underline stack in order to constrain the symbolic execution of the FDM.
- Second, it ensures the relative atomicity inside the FDM, i.e., hardware transaction functions should be atomic to each other.
- Third, it monitors the execution of the FDM and raises an alarm if the FDM's state indicates a protocol violation by either the driver or the device.

Essentially, co-monitoring does two things: deduces the device's states based on the monitored communications; and raises an alarm if a violation is detected by analyzing the FDM's states with the communications.

8.2.4 Formal-model-guided Automatic Test Case Generation

It is a common practice that a higher level design, a.k.a., a golden model, is developed before a system is actually implemented. Such a golden model is very useful to evaluate the correctness and efficiency of the design. After the system is implemented, it is also desired that the golden model can be used to guide the test case generation.

As illustrated in Figure 8.4, we can utilize a FDM as the golden model to generated test cases for its hardware device. There are three steps for automatic test case generation:

- First, it utilizes symbolic path exploring tools such as KLEE [17] to generate path constraints for the FDM. Path constraints describe the condition that must hold on execution of a path.

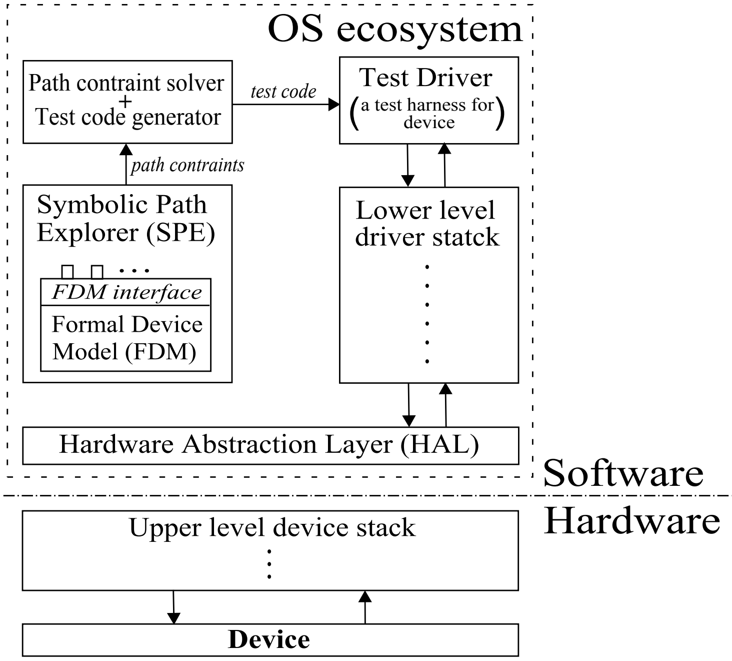


Figure 8.4: Automatic test case generation based on formal device model.

- Second, it implements a *path constraint solver*, which, given a path constraint, generates concrete input to the FDM in order to execute the path. It also implements a *test code generator*, which generates a test harness using the concrete input.

- Third, it loads the test harness, as a driver of the device, into the driver stack; therefore, automatic testing is conducted as if a driver operates its device.

Following this approach, test cases can be generated automatically and the device functionality can be covered by these test cases in a low cost, because the device is logically similar to its FDM and the symbolic path explorer often can generate path constraints in such a way that guarantees a high path coverage on the FDM. This approach can also be combined with co-monitoring; therefore, any protocol violation by the device can be automatically detected during testing.

REFERENCES

[1] Accellera. *Property Specification Language – Reference Manual*, 1.1 edition, June 9 2004.

[2] Rajeev Alur, Thomas A. Henzinger, and Pei-Hsin Ho. Automatic symbolic verification of embedded systems. *IEEE Transactions on Software Engineering (TSE)*, 22(3):181–201, March 1996.

[3] Felice Balarin, Harry Hsieh, Attila Jurecska, Luciano Lavagno, and Alberto L. Sangiovanni-Vincentelli. Formal verification of embedded systems based on CFSM networks. In *Proceedings of the 33st Design Automation Conference (DAC)*, pages 568–571, New York, NY, USA, 1996. ACM.

[4] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys)*, pages 73–85, New York, NY, USA, April 18-21 2006. ACM.

[5] Thomas Ball, Rupak Majumdar, Todd D. Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 203–213, New York, NY, USA, June 20-22 2001. ACM.

[6] Thomas Ball and Sriram K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *Proceedings of the 7th international SPIN conference on*

Model checking software, volume 1885 of *Lecture Notes in Computer Science*, pages 113–130. Springer, August 30 - September 1 2000.

[7] Thomas Ball and Sriram K. Rajamani. Boolean programs: A model and process for software analysis. Technical Report MSR-TR-2000-14, Microsoft Research, Microsoft Corporation, One Microsoft Way Redmond, WA 98052, February 2000.

[8] Thomas Ball and Sriram K. Rajamani. Generating abstract explanations of spurious counterexamples in C programs. Technical Report MSR-TR-2002-09, Microsoft Research, January 2002.

[9] Thomas Ball and Sriram K. Rajamani. The SLAM project: debugging system software via static analysis. In *Proceedings of the 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 1–3, New York, NY, USA, January 16-18 2002. ACM.

[10] Thomas Ball and Sriram K. Rajamani. SLIC: a Specification Language for Interface Checking (of C). Technical Report MSR-TR-2001-21, Microsoft Research, Microsoft Corporation, One Microsoft Way Redmond, WA 98052, January 2002.

[11] David Becker, Raj K. Singh, and Stephen G. Tell. An engineering environment for hardware/software co-simulation. In *Proceedings of the 29th Design Automation Conference (DAC)*, pages 129–134, Los Alamitos, CA, USA, June 8-12 1992. IEEE Computer Society.

[12] Berkeley. Ptolemy project. http://ptolemy.eecs.berkeley.edu/index.htm, October 2010.

[13] Dirk Beyer, Adam J. Chlipala, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The BLAST query language for software verification. In

Proceedings of the 11th International Static Analysis Symposium (SAS), volume 3148 of *Lecture Notes in Computer Science*, pages 2–18. Springer, August 26-28 2004.

[14] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker BLAST. *International Journal on Software Tools for Technology Transfer (STTT)*, 9(5-6):505–525, September 2007.

[15] Ahmed Bouajjani, Javier Esparza, and Oded Maler. Reachability analysis of pushdown automata: Application to model-checking. In *Proceedings of the 8th International Conference on Concurrency Theory (CONCUR)*, volume 1243 of *Lecture Notes in Computer Science*, pages 135–150. Springer, July 1-4 1997.

[16] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, August 1986.

[17] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 209–224, Berkeley, CA, USA, December 8-10 2008. USENIX Association.

[18] Vitaly Chipounov, Vlad Georgescu, Cristian Zamfir, and George Candea. Selective symbolic execution. In *Proceedings of the 5th Workshop on Hot Topics in System Dependability (HotDep)*, June 29 2009.

[19] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *Proceedings of the eighteenth ACM symposium on Operating systems principles (SOSP)*, pages 73–88, New York, NY, USA, October 21-24 2001. ACM.

[20] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM (JACM)*, 50(5):752–794, September 2003.

[21] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Proceedings of Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71, London, UK, May 1981. Springer.

[22] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.

[23] Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC, and Philips. *Universal Serial Bus Specification*, 2.0 edition, April 27 2000.

[24] Byron Cook, Alexey Gotsman, Andreas Podelski, Andrey Rybalchenko, and Moshe Y. Vardi. Proving that programs eventually do something good. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 265–276, New York, NY, USA, January 17-19 2007. ACM.

[25] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 415–426, New York, NY, USA, June 11-14 2006. ACM.

[26] Luis Alejandro Cortes, Petru Eles, and Zebo Peng. Formal coverification of embedded systems using model checking. In *Proceedings of the 26th EUROMICRO Conference*, pages 1106–1113, Washington, DC, USA, September 5-7 2000. IEEE Computer Society.

[27] Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *Proceedings of the Joint 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, pages 109–120, New York, NY, USA, September 10-14 2001. ACM.

[28] David A. Duffy. *Principles of automated theorem proving*. John Wiley & Sons, Inc., New York, NY, USA, 1991.

[29] Eclipse Foundation. Eclipse. http://www.eclipse.org, October 2010.

[30] Alessandro Forin, Behnam Neekzad, and Nathaniel L. Lynch. Giano: The two-headed system simulator. Technical Report MSR-TR-2006-130, Microsoft Research, September 2006.

[31] Paul Gastin and Denis Oddoux. Fast LTL to Büchi automata translation. In *Proceedings of the 13th International Conference on Computer Aided Verification (CAV)*, volume 2102 of *Lecture Notes in Computer Science*, pages 53–65, London, UK, July 18-22 2001. Springer.

[32] A. Ghosh, M. Bershteyn, R. Casley, C. Chien, A. Jain, M. Lipsie, D. Tarrodaychik, and O. Yamamo. A hardware-software co-simulator for embedded system design and debugging. In *Proceedings of Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 155–164, New York, NY, USA, August 29 - September 1 1995. ACM.

[33] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*. PhD thesis, University of Liege, November 1994.

[34] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In *Proceedings of the 9th International Conference on Computer Aided*

Verification (CAV), volume 1254 of *Lecture Notes in Computer Science*, pages 72–83, London, UK, June 22-25 1997. Springer.

[35] Daniel Gro*β*e, Ulrich Kühne, and Rolf Drechsler. HW/SW co-verification of embedded systems using bounded model checking. In *Proceedings of ACM Great Lakes Symposium on VLSI (GLSVLSI)*, pages 43–48, New York, NY, USA, April 30 - May 1 2006. ACM.

[36] Rajesh Gupta, Claudionor Coelho, and Giovanni De Micheli. Synthesis and simulation of digital systems containing interacting hardware and software components. In *Proceedings of the 29th Design Automation Conference (DAC)*, pages 225–230, Los Alamitos, CA, USA, June 8-12 1992. IEEE Computer Society.

[37] Andreas Hoffmann, Tim Kogel, and Heinrich Meyr. A framework for fast hardware-software co-simulation. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, pages 760–765, Piscataway, NJ, USA, March 12 - 16 2001. IEEE Press.

[38] IEEE. *IEEE Standard for Verilog (IEEE Std 1364-2005)*. IEEE, 2005.

[39] Intel. *Intel 8255x 10/100 Mbps Ethernet Controller Family – Open Source Software Developer Manual*, 1.3 edition, January 2006.

[40] Matt Kaufmann, J. Strother Moore, and Panagiotis Manolios. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[41] Christoph Kern and Mark R. Greenstreet. Formal verification in hardware design: a survey. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 4(2):123–193, April 1999.

[42] Daniel Kroening and Natasha Sharygina. Formal verification of systemc by automatic hardware/software partitioning. In *Proceedings of the 2nd ACM/IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*, pages 101–110, Washington, DC, USA, July 11-14 2005. IEEE Computer Society.

[43] Robert P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, Princeton, New Jersey, USA, 1994.

[44] Robert P. Kurshan, Vladdimir Levin, Marius Minea, Doron Peled, and Hüsnü Yenigün. Static partial order reduction. In *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 1384 of *Lecture Notes in Computer Science*, pages 345–357, London, UK, March 28 - April 4 1998. Springer.

[45] Robert P. Kurshan, Vladimir Levin, Marius Minea, Doron Peled, and Hüsnü Yenigün. Combining software and hardware verification techniques. *Formal Methods in System Design (FMSD)*, 21(3):251–280, November 2002.

[46] V. Kuznetsov, V. Chipounov, and G. Candea. Testing closed-source binary device drivers with DDT. In *Proceedings of the 2010 USENIX annual technical conference (USENIXATC)*, pages 12–12, Berkeley, CA, USA, June 22C25 2010. USENIX Association.

[47] Juncao Li, Nicholas T. Pilkington, Fei Xie, and Qiang Liu. Embedded architecture description language. *Journal of Systems and Software (JSS)*, 83(2):235–252, February 2010.

[48] Juncao Li, Xiuli Sun, Fei Xie, and Xiaoyu Song. Component-based abstraction and refinement. In *Proceedings of the 10th International Conference on*

Software Reuse (ICSR), volume 5030 of *Lecture Notes in Computer Science*, pages 39–51, Berlin, Heidelberg, May 25-29 2008. Springer.

[49] Juncao Li, Fei Xie, Thomas Ball, and Vladimir Levin. Efficient reachability analysis of Büchi pushdown systems for hardware/software co-verification. In *Proceedings of the 22nd International Conference on Computer Aided Verification (CAV)*, volume 6174 of *Lecture Notes in Computer Science*, pages 339–353. Springer, July 15-19 2010.

[50] Juncao Li, Fei Xie, Thomas Ball, Vladimir Levin, and Con McGarvey. An automata-theoretic approach to hardware/software co-verification. In *Proceedings of the 13th International Conference on Fundamental Approaches to Software Engineering (FASE)*, volume 6013 of *Lecture Notes in Computer Science*, pages 248–262. Springer, March 20-28 2010.

[51] Juncao Li, Fei Xie, and Huaiyu Liu. Guiding component-based hardware/software co-verification with patterns. In *Proceedings of the 33rd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO-SEAA)*, pages 67–74, Washington, DC, USA, August 28-31 2007. IEEE Computer Society.

[52] Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 137–151, New York, NY, USA, August 10-12 1987. ACM.

[53] Kenneth L. McMillan. *Symbolic Model-Checking: an approach to the state explosion problem*. PhD thesis, Carnegie Mellon University, May 1992.

[54] Kenneth L. McMillan. *The SMV System*. Carnegie Mellon University, November 6 2000.

[55] Mentor Graphics. Seamless. http://www.mentor.com, October 2010.

[56] Microsoft. Device simulation framework design guide. MSDN: http://msdn.microsoft.com/en-us/library/ff538293.aspx, October 2010.

[57] Microsoft. Framework USB reference. MSDN: http://msdn.microsoft.com/en-us/library/ff543092(VS.85).aspx, October 2010.

[58] Microsoft. Microsoft visual studio. http://www.microsoft.com/visualstudio/en-us/default.mspx, October 2010.

[59] Microsoft. OSRUSBFX2: sample WDF driver for USB 2.0 devices. MSDN: http://msdn.microsoft.com/en-us/library/ff544368(VS.85).aspx, October 2010.

[60] Microsoft. Programming techniques for framework-based drivers. MSDN: http://msdn.microsoft.com/en-us/library/ff544546.aspx, October 2010.

[61] Microsoft. Sample WDF driver for Intel 8255x 10/100 Mbps Ethernet controller. MSDN: http://msdn.microsoft.com/en-us/library/ff544373.aspx, October 2010.

[62] Microsoft. Synchronizing interrupt code. MSDN: http://msdn.microsoft.com/en-us/library/ff544728.aspx, October 2010.

[63] Microsoft. USBSAMP: sample WDF driver for USB 2.0 devices. MSDN: http://msdn.microsoft.com/en-us/library/ff544747(VS.85).aspx, October 2010.

[64] David Monniaux. Verification of device drivers and intelligent controllers: a case study. In *Proceedings of the 7th ACM & IEEE International conference on Embedded Software (EMSOFT)*, pages 30–36, New York, NY, USA, September 30 - October 3 2007. ACM.

[65] Brendan Murphy and Mario R. Garzia. Software reliability engineering for mass market products. *Software Reliabilty Engineering*, 8(1), December 2004.

[66] Open SystemC Initiative (OSCI). *http://www.systemc.org/*, October 6 2010.

[67] OSR. Sample WDF driver for Sealevel digital I/O kit. OSR: http://www.osronline.com/article.cfm?article=403, April 17 2007.

[68] Claudio Passerone, Luciano Lavagno, Massimiliano Chiodo, and Alberto L. Sangiovanni-Vincentelli. Fast hardware/software co-simulation for virtual prototyping and trade-off analysis. In *Proceedings of the 34st Design Automation Conference (DAC)*, pages 389–394, New York, NY, USA, June 9-13 1997. ACM.

[69] Doron Peled. Combining partial order reductions with on-the-fly model-checking. *Formal Methods in System Design (FMSD)*, 8(1):39–64, January 1996.

[70] Carl Pixley. Introduction to a computational theory and implementation of sequential hardware equivalence. In *Proceedings of the 2nd International Workshop, on Computer Aided Verification (CAV)*, volume 531 of *Lecture Notes in Computer Science*, pages 54–64, London, UK, June 18-21 1990. Springer.

[71] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, pages 46–57, Washington, DC, USA, October 31 - November 2 1977. IEEE Computer Society.

[72] Amir Pnueli. Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends. In *Current Trends in Concurrency*, volume 224 of *Lecture Notes in Computer Science*, pages 510–584. Springer, New York, NY, USA, 1986.

[73] Shaz Qadeer and Jakob Rehof. Context-bounded model checking of concurrent software. In *Proceedings of the 11th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 3440 of *Lecture Notes in Computer Science*, pages 93–107. Springer, April 4-8 2005.

[74] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351, London, UK, April 6-8 1982. Springer.

[75] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(2):416–430, March 2000.

[76] James A. Rowson. Hardware/software co-simulation. In *Proceedings of the 31st Design Automation Conference (DAC)*, pages 439–440, New York, NY, USA, June 6-10 1994. ACM.

[77] Stefan Schwoon. *Model-Checking Pushdown Systems*. PhD thesis, Technische Universität München, Institut für Informatik, June 2002.

[78] Sealevel Systems, Inc. *PIO-24.LPCI User Manual*, July 2006.

[79] Luc Semeria and Abhijit Ghosh. Methodology for hardware/software co-verification in C/C++. In *Proceedings of Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 405–408, New York, NY, USA, January 26 - 28 2000. ACM.

[80] Peter Shier. Using the device simulation framework for software simulation of USB devices. http://download.microsoft.com/download/5/b/9/5b97017b-e28a-4bae-ba48-174cf47d23cd/DEV098_WH06.ppt, 2006.

[81] Alok Sinha. Windows driver quality signature. http://download.microsoft.com/download/9/8/f/98f3fe47-dfc3-4e74-92a3-088782200fe7/TWDE05008_WinHEC05.ppt, February 2005.

[82] David A. Solomon. *Inside Windows NT*. Microsoft Press, 2 edition, 1998.

[83] Michael M. Swift. *Improving the Reliability of Commodity Operating Systems*. PhD thesis, University of Washington, October 2005.

[84] Jan Tretmans. *A formal approach to conformance testing*. PhD thesis, University of Twente, December 1992.

[85] Peter H. J. van Eijk, Chris A. Vissers, and Michel Diaz, editors. *The formal description technique LOTOS*. Elsevier Science Inc., New York, NY, USA, 1989.

[86] Moshe Y. Vardi and Pierre Wolper. Automata-theoretic techniques for modal logics of programs. *Journal of Computer and System Sciences (JCSS)*, 32(2):183–221, April 1986.

[87] Sven Verdoolaege, Gerda Janssens, and Maurice Bruynooghe. Equivalence checking of static affine programs using widening to handle recurrences. In *Proceedings of the 21st International Conference on Computer Aided Verification (CAV)*, volume 5643 of *Lecture Notes in Computer Science*, pages 599–613, Berlin, Heidelberg, June 26 - July 2 2009. Springer.

[88] Fei Xie and Huaiyu Liu. Unified property specification for hardware/software co-verification. In *Proceedings of the 31st Annual International Computer Software and Applications Conference (COMPSAC)*, pages 483–490, Washington, DC, USA, July 24-27 2007. IEEE Computer Society.

[89] Fei Xie, Guowu Yang, and Xiaoyu Song. Component-based hardware/software

co-verification for building trustworthy embedded systems. *Journal of Systems and Software (JSS)*, 80(5):643–654, May 2007.