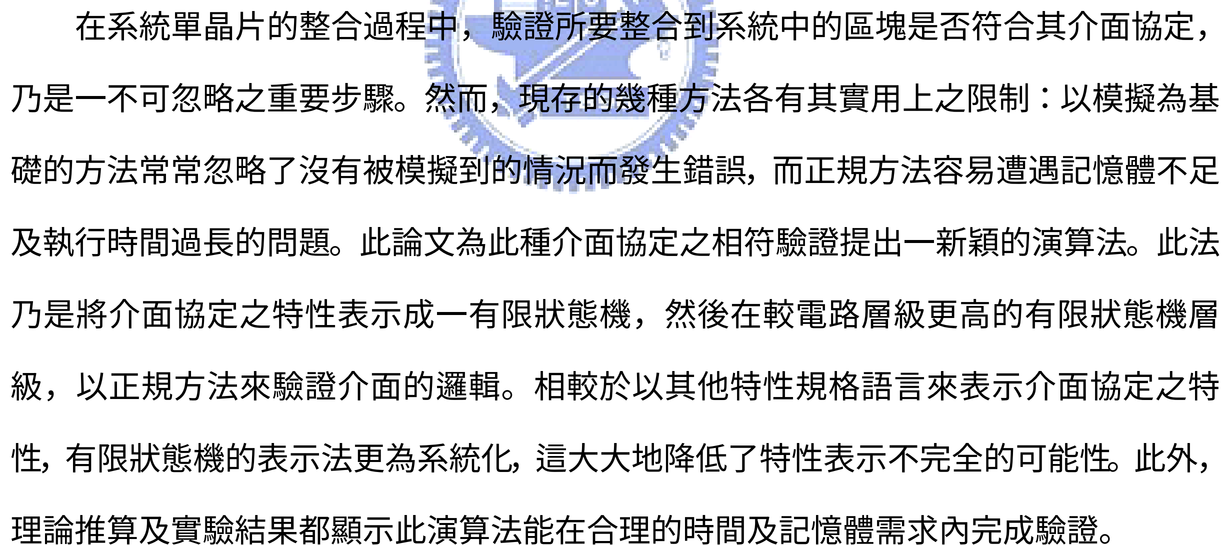
應用於系統單晶片中介面相符驗證之正規方法

研究生 ： 楊 雅 菁　　　　　　指導教授 ： 周 景 揚 博士

國 立 交 通 大 學

電 子 工 程 學 系　電 子 研 究 所 碩 士 班



摘 要

在系統單晶片的整合過程中，驗證所要整合到系統中的區塊是否符合其介面協定，乃是一不可忽略之重要步驟。然而，現存的幾種方法各有其實用上之限制：以模擬為基礎的方法常常忽略了沒有被模擬到的情況而發生錯誤，而正規方法容易遭遇記憶體不足及執行時間過長的問題。此論文為此種介面協定之相符驗證提出一新穎的演算法。此法乃是將介面協定之特性表示成一有限狀態機，然後在較電路層級更高的有限狀態機層級，以正規方法來驗證介面的邏輯。相較於以其他特性規格語言來表示介面協定之特性，有限狀態機的表示法更為系統化，這大大地降低了特性表示不完全的可能性。此外，理論推算及實驗結果都顯示此演算法能在合理的時間及記憶體需求內完成驗證。

A Formal Approach for Interface Compliance Verification in SoC

Student : Ya-Ching Yang Advisor : Dr. Jing-Yang Jou

Department of Electronics Engineering

Institute of Electronics

National Chiao Tung University

ABSTRACT

Verifying whether a building block conforms to certain interface protocol is one of the important steps while constructing a system-on-a-chip (SoC). However, most existing methods have their own limitations. Simulation-based methods have the false positive problem while formal property checking methods may suffer from memory explosion and excessive runtime. In this thesis, we propose a novel branch-and-bound algorithm for interface protocol compliance verification. The properties of the interface protocol are specified as a specification FSM, and the interface logic is formally verified at the higher FSM level. Using the FSM for property specification is relatively systematic than using other proprietary property languages, which greatly reduces the possibility of incomplete property identification. And it is shown theoretically and experimentally that the proposed algorithm can finish in reasonable time and space complexity.

Acknowledgements

I would like to express my sincere gratitude to my advisors, Professor Jing-Yang Jou and Professor Juinn-Dar Huang, for their insightful suggestion and patient guidance throughout the course of this work. I am also indebted to Chia-Chih Yen and Che-Hua Shih, for their great help and constructive suggestions on my research. Special thanks to all members in the EDA lab and 412 lab for their friendship and company. Finally, I have to show my greatest appreciation to my family and my friends for their love and encouragement.

Contents

摘要 i
Abstract ii
Acknowledgements iii
Contents iv
Lists of Tables vi
Lists of Figures vii
Chapter 1 Introduction 1
1.1 Related works 3
1.1.1 Simulation-based verification 3
1.1.2 Formal verification 5
1.1.3 Assertion-based verification 5
1.2 Motivation 6
1.3 Our approach 7
Chapter 2 Preliminaries 8
2.1 Bus signals 8
2.2 FSM 9
2.3 EFSM 10
2.4 Monitor 12
2.5 Specification FSM 12
2.5.1 A spec FSM example 13
2.5.2 Correctness of the spec FSM 14
2.5.3 Advantages 15
2.5.4 Limitations of expression power 15
2.5.5 Compliance verification with the spec FSM 16
Chapter 3 Our Approach 17
3.1 Problem formulation 17
3.2 Observations 18
3.3 Solution 19
3.4 Algorithm 21
3.5 Complexity analysis 22
3.5.1 Time complexity 22
3.5.2 Space complexity 23
Chapter 4 Extension 25
4.1 Spec EFSM 25
4.2 Extended algorithm 26

4.3 Complexity analysis 28
4.3.1 Time complexity 28
4.3.2 Space complexity..29
4.4 Other extensions 30
Chapter 5 Experimental Results..31
5.1 Program implementation 31
5.1.1 Input file format..31
5.1.2 Input file check 32
5.1.3 Counterexample..33
5.2 Result analysis 34
Chapter 6 Conclusions and Future Work 38
6.1 Contributions 38
6.2 Comparisons 39
6.3 Future work 39
Reference..41
Vita..45

List of Tables

Table 1. The DUVs and the verification results. 36
Table 2. Complexity comparison. 37

List of Figures

Figure 1-1. An IP core reused in different system platforms. 2
Figure 1-2. Typical compliance verification flows. 3
Figure 2-1. Notations of bus signals. 9
Figure 2-2. An example of FSM. 10
Figure 2-3. An EFSM example. 12
Figure 2-4. A part of the spec FSM of the AMBA AHB slave protocol. 14
Figure 3-1. The FSM of an AHB slave interface design. 18
Figure 3-2. The tree-growing process. 21
Figure 4-1. Specify the same protocol in (a) FSM and (b) EFSM 26
Figure 4-2. (a) A wrong implementation of the protocol in Figure 4-1 and (b) the tree generated by the extended algorithm. 27
Figure 5-1. The spec EFSM of Figure 4-1 in the enhanced BLIF. 32
Figure 5-2. The report of catching a bug in the spec FSM of AHB slave protocol. 33
Figure 5-3. The verification result of the design in Figure 4-2(a). 34

Chapter 1

Introduction

In modern system-on-a-chip (SoC) designs, certain number of building blocks should be reusable intellectual property (IP) cores to accelerate the design process. To achieve an even higher level of reusability, the platform-based design methodology, in which all IP cores are pre-verified, is usually adopted. Figure 1-1 illustrates how an IP core is reused in typical platform-based designs. An IP core is wrapped with the appropriate interface (I/F) logic that complies with certain I/F protocol (usually an on-chip bus protocol) so that it can concordantly communicate with other IP cores within the system. When the IP core is desired in another platform utilizing a different I/F protocol, all one has to do is simply changing the I/F logic wrapper without altering the core function logic. Thus, by separating the core function logic from the I/F logic, the IP core can be easily and quickly integrated into different system platforms utilizing different I/F protocols [1]. In addition, even under a given I/F protocol, the I/F logic can still vary in a big dynamic range due to numerous valid

configurations and options. Therefore, the interface compliance must be verified thoroughly during SoC integration.

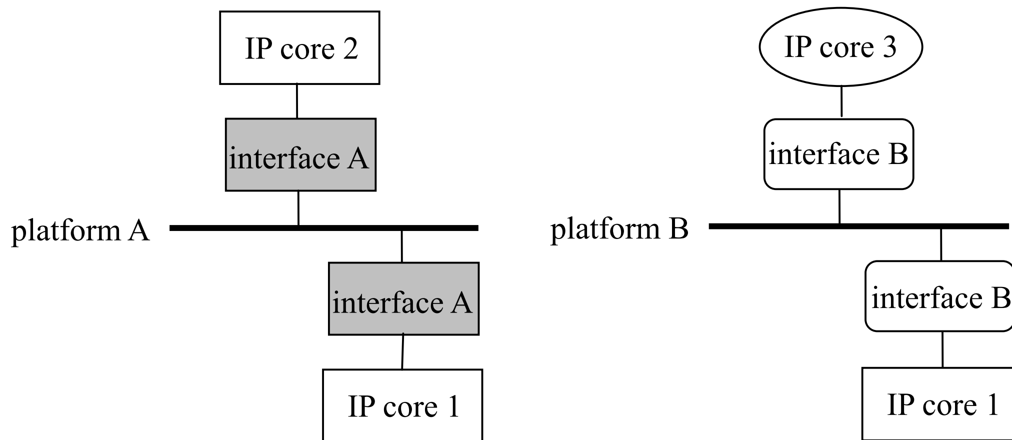


Figure 1-1. An IP core reused in different system platforms.

1896

Generally, the compliance verification process can be divided into two phases. The first phase, the property specification phase, is to translate the interface protocols into properties. Many property languages or specification styles can be adopted to specify the protocols. How to select among these styles depends on whether the verification tool in use supports this style and whether this style meets your requirements. The second phase, the verification phase, is to verify the design against those properties. There are two major categories for this verification phase: simulation-based (dynamic) methods and formal (static) ones. The methods in both categories have their own advantages and limitations.

Figure 1-2 illustrates this typical verification process. The two phases are clearly shown in the figure. The details in each block are explained in later sections.

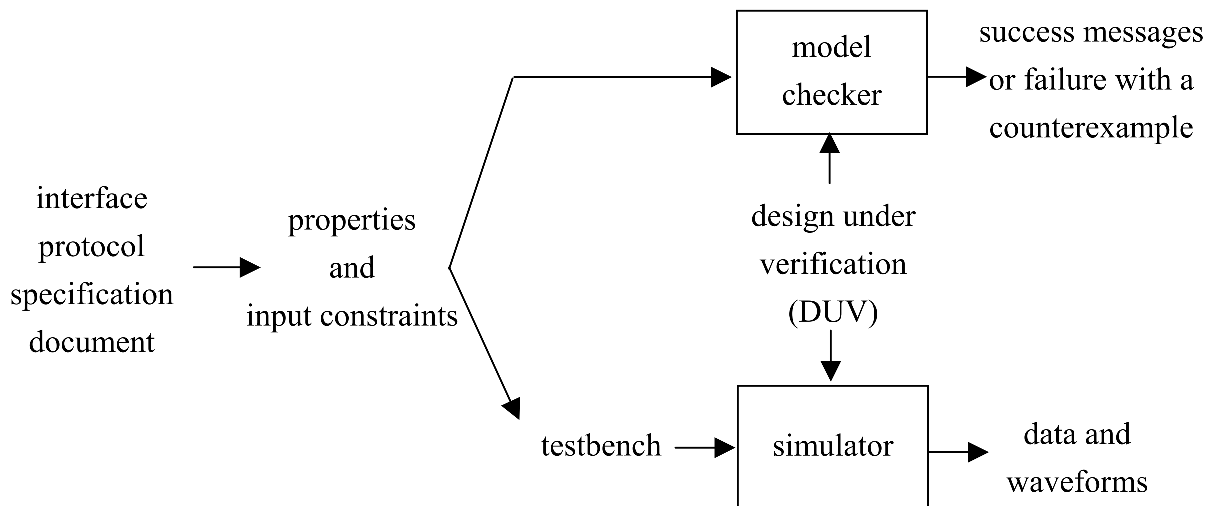


Figure 1-2. Typical compliance verification flows.

Most works on compliance verification involve both phases. They may choose different ways to specify properties in combination with different ways to accomplish the verification. In the following section, some studies of these works are given. The characteristics of their property specifying styles and verification schemes are also discussed.

1.1 Related works

1.1.1 Simulation-based verification

The simulation-based verification approach is age-old but popular. Several researches are based on this approach [2-8]. In [2], properties of a protocol specification are represented in HDL monitors. This specification style can be simulated by HDL simulators without extra interpretation. In addition, the outputs of the monitors can tell the correctness

of the design under verification (DUV). Based on this, a systematic method of representing properties of an I/F protocol as a monitor FSM is proposed in [3]. This work also defines its own coverage metrics to measure the quality of the simulation trace. Some other works focus on the automation of the verification process from high-level specification styles. In [4], the high-level specification style is based on the extension of regular expressions, and then a linear-time algorithm of translating the specification style into monitor circuits is proposed. Similar work of generating monitor circuits is done in [5], but the specification style of this work is the GSTE assertion graphs. In [6-7], input constraints and biasing are considered. They propose methods for automatically generating inputs that are legal and even biased to enhance the effectiveness of the simulation patterns.

Besides academic researches, there are some commercial products developed for compliance verification. For example, the commercial tool ACT [9] is available for verifying AMBA [10] compliance. Furthermore, Synopsys DesignWare provides verification IPs [11] for some popular interface protocols like PCI, USB, and AMBA. The verification IPs includes monitors, master IPs, and slave IPs. They provide a system environment so that the DUVs can be pre-verified before being integrated into real systems.

Although all works mentioned above greatly facilitate the simulation process, all simulation-based approaches encounter an unavoidable problem: the compliance can never be assured no matter how high the coverage is or how many cycles the simulation lasts for. The bugs of the design may lie in the corner where the simulation trace never reaches. In other words, the design might still be wrong while all simulation results tell that the design is correct. This *false positive* situation is the most severe problem of simulation-based verification.

1.1.2 Formal verification

Formal verification can avoid this kind of false positive problem [12-15]. Model checking techniques [16] are used for I/F protocol compliance verification in [12-14]. In these works, properties of the I/F protocol are specified in the CTL language, and then the model checker verifies the DUV against these properties and gives a counterexample if an error in the DUV is found. Once the model checker reports a success, the design is guaranteed to be 100% compliant with these properties.

However, properties in CTL are not easily thorough and the process of extracting properties from a specification document written by natural languages is generally complicated and painful. It is very likely that some properties are actually implied by the specification but accidentally not extracted and thus ignored during the formal verification. Moreover, memory explosion and excessively long runtime may be further serious problems while the size of the design increases. All these issues potentially prevent model checking techniques from being effectively and efficiently applied on interface compliance verification.

1.1.3 Assertion-based verification

Recently, the assertion-based verification (ABV) methodology is getting popular and several property specification languages (such as PSL [17], OVL [18], OVA [19], and SVA [20]) are developed to provide alternative ways to specify properties in additional to CTL. These emerging languages are relatively more easily understood than CTL at the semantic and syntactic level. Furthermore, these languages are supported in both simulation-based and formal verification tools. However, no matter what emerging property specification language is used, either the (simulation-based) dynamic ABV or (formal) static ABV inherently suffers from the same problems in traditional dynamic or static verification

mentioned previously.

1.2 Motivation

The property specification style in our work is directly inspired by [2-3]. The work in [2] gives clear and useful concepts of using HDL monitors as the interface protocol specifications, and then [3] gives the concepts of specifying monitors in the higher FSM level. Both works have the advantage that all engineers are familiar with these styles, HDL and FSM. Hence, engineers are very likely to accept the styles because no particular specification languages need to be learned. Our specification style, the specification FSM, is actually a variation of [3]. It inherits the advantage of comprehensibility and simplicity from [2-3].

However, the works in [2-3] use the simulation-based approach in the verification phase. They inevitably suffer the false positive problem which we try to avoid. Therefore, a novel formal verification approach using this friendly specification style has to be developed.

A similar work that specifies properties as state machines and verifies using formal techniques is proposed in [21]. However, this work specifies one single property in one state machine. If we specify an interface protocol in this method, the specification becomes many unorganized state machines. This leads to the same incomplete property identification problem as in traditional property languages. Moreover, verification efficiency may decline because it is disorderly and unsystematic.

1.3 Our approach

Our approach, unlike any of above, intends to formally verify whether the I/F logic is compliant with the I/F protocol at the FSM level. The properties of the I/F protocol are specified as a specification FSM. We believe that this style is more readable and systematic than other specification styles proposed in [2,4,5,17-20,22-24] and thus enables complete property extraction. The golden specification FSM is only created once for a specific I/F protocol and then can be used to verify all designs claimed to be compliant with. Developing the FSM of the I/F logic is generally essential since it is one of the design steps (before the HDL coding) under a typical modern design flow. Since the verification is applied at the higher FSM level and only the separated interface logic is under verification, our approach can effectively and efficiently complete the verification even if it is a formal method indeed.

1896

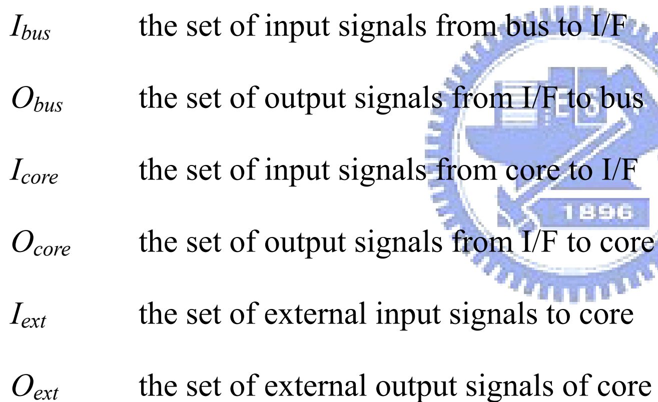
The rest of this thesis is organized as follows. Chapter 2 introduces necessary terminology, concepts, and the specification style of our compliance verification. In Chapter 3, our novel branch-and-bound algorithm that formally verifies the compliance between the DUV FSM and the golden specification FSM of the interface protocol is given in detail. Chapter 4 extends the algorithm to handle the compliance verification in which the specification is modeled as the EFSM. Chapter 5 shows the experimental results, and the concluding remarks are given in Chapter 6.

Chapter 2

Preliminaries

2.1 Bus signals

Typical I/O signals of a bus interface logic are shown in Figure 2-1.



I_{bus} the set of input signals from bus to I/F

O_{bus} the set of output signals from I/F to bus

I_{core} the set of input signals from core to I/F

O_{core} the set of output signals from I/F to core

I_{ext} the set of external input signals to core

O_{ext} the set of external output signals of core

In addition,

I_{ctrl} $I_{ctrl} \subseteq I_{bus}$, the subset of bus inputs that directly controls the bus behavior from the protocol perspective

O_{ctrl} $O_{ctrl} \subseteq O_{bus}$, the subset of bus outputs that directly controls the bus behavior from the protocol perspective

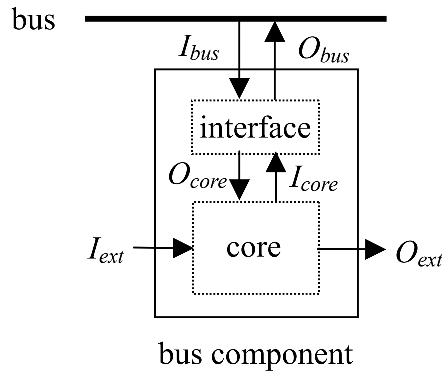
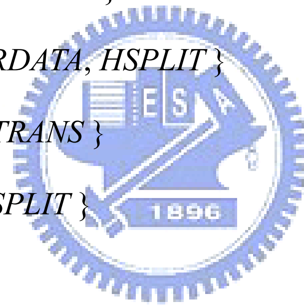


Figure 2-1. Notations of bus signals.

Use the AMBA AHB slave interface [10] as an example,

I_{bus} = { *HSEL, HREADYin, HADDR, HWRITE, HTRANS, HSIZE, HBURST, HWDATA, HMASTER, HMASTERLOCK* }

O_{bus} = { *HREADY, HRESP, HRDATA, HSPLIT* }



I_{ctrl} = { *HSEL, HREADYin, HTRANS* }

O_{ctrl} = { *HREADY, HRESP, HSPLIT* }

It is not uncommon that just a small portion of bus I/F signals are classified into I_{ctrl} and O_{ctrl}. For example, what value that the address/data bus exactly carries does not affect the bus behavior at the protocol level. Note also that I_{core}, O_{core}, I_{ext}, and O_{ext} may differ from design to design.

2.2 FSM

The FSM is a quintuple $M=(Q, \Sigma, \Delta, \sigma, q0)$ where

Q the set of symbols denoting states

Σ the set of symbols denoting inputs

Δ	the set of symbols denoting outputs
$\sigma: Q\times B^{\Sigma} \rightarrow Q\times B^{\Delta}$	the transition function
$q0$	$q0 \in Q$, the initial state

Additionally,

$\lvert e_{qi} \rvert$	the number of outgoing transition edges of the state qi
$f_{qi,qj}: B^{\Sigma}\times B^{\Delta} \rightarrow B$	the Boolean function s.t. $f_{qi,qj}(x, y)=1$ iff $\sigma(qi, x)=(qj, y)$

Figure 2-2 gives an example to illustrate above terminology. In all state transition graphs in this thesis, the initial states are specified in bold circles. In the example FSM,

$Q = \{ s1, s2, s3 \}$, $\Sigma = \{ i1, i2 \}$, $\Delta = \{ o1 \}$, $q0 = s1$,

$\lvert e_{s1} \rvert = 2$, $f_{s1,s2} = i1' \bullet o1$.

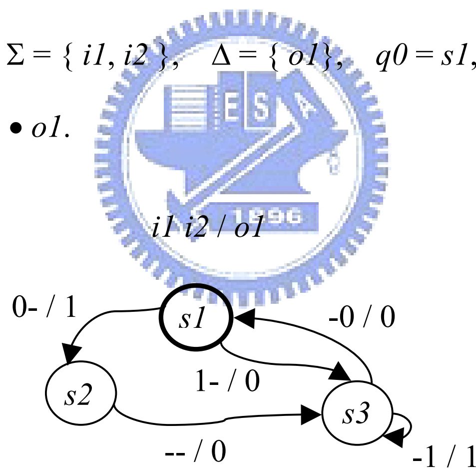


Figure 2-2. An example of FSM.

2.3 EFSM

The extended finite state machine (EFSM) [25] is a 7-tuple $M=(Q, \Sigma, \Delta, x, T, q0, x0)$ where

Q	the set of symbols denoting states
Σ	the set of symbols denoting inputs

Δ	the set of symbols denoting outputs
x	the set of symbols denoting variables
$x0$	the initial values of variables in x
$q0$	$q0 \in Q$, the initial state
T	the set of transitions t's, each t is a 6-tuple $t=(s_t, q_t, i_t, o_t, P_t, A_t)$ where
s_t, q_t, i_t, o_t	current state, next state, set of input values, set of output values
$P_t(x), A_t(x)$	the predicate and the action on current variables

Additionally, we further define:

$P_{qi,qj}(x)$	the predicate of the transition from the state qi to qj
$A_{qi,qj}(x)$	the action of the transition from the state qi to qj

A transition $t=(s_t, q_t, i_t, o_t, P_t, A_t)$ means if the input value is in i_t and the predicate $P_t(x)$ is evaluated true, then the EFSM outputs o_t, performs the action $x = A_t(x)$ and moves from the current state s_t to the next state q_t at the next cycle. Figure 2-3 illustrates an EFSM example. This EFSM contains only one variable v. In the state $s1$, if the input value is 0 and v>1, the EFSM outputs 0, and increases v by 1 while jumping to the next state $s2$ at the next cycle. The notations $P_{qi,qj}(x)$ and $A_{qi,qj}(x)$ are defined for the algorithm development in Section 4.2. In Figure 2-3,

$$P_{s1,s2}(v) = \begin{cases} \text{TRUE} & \text{if } v>1 \\ \text{FALSE} & \text{otherwise} \end{cases}$$

$$A_{s1,s2}(v) = v + 1$$

EFSMs are particularly suitable for specifying counters. Therefore, they can be utilized to specify interface protocols much more clearly. The details are discussed in

Section 4.1.

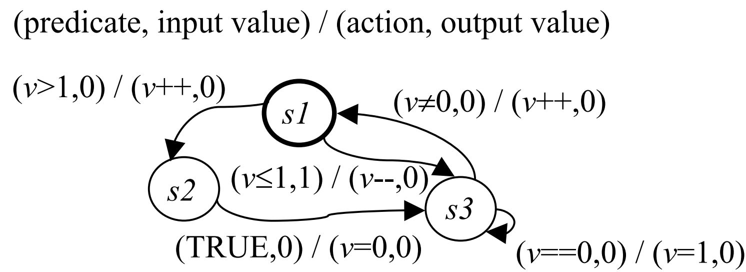
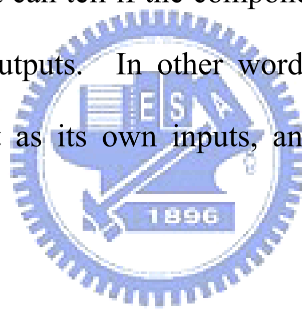


Figure 2-3. An EFSM example.

2.4 Monitor

A monitor of a component can tell if the component behaves according to the protocol by observing its inputs and outputs. In other words, the monitor takes the inputs and outputs of the bus component as its own inputs, and determines the correctness of the component.



2.5 Specification FSM

In our approach, the properties of the protocol specification are represented in the *specification FSM*, or *spec FSM*. It specifies whether the output response of a specific implementation (DUV) is legal or not under a given input sequence. In other words, the spec FSM actually acts as a *functional monitor* of the DUV. It observes both the input and output sequence and determine the correctness of its behavior. The possible DUV behaviors are concluded to be:

1. don't-care: The behavior is not defined since the input sequence is not supposed to appear.

2. legal: The output sequence is allowed by the protocol under a valid input sequence.
3. illegal: The output sequence is prohibited by the protocol under a valid input sequence.

For differentiating these behaviors, in every spec FSM, two special states are defined: *vio* and *dc*. The spec FSM moves to the state *dc* if a don't-care input sequence is applied to the DUV. If the DUV behaves illegally under a valid input sequence, the spec FSM moves to the state *vio*. If the DUV behaves legally under a valid input sequence, the spec FSM moves among other normal states excluding the state *dc* and *vio*. Accordingly, the spec FSM is a finite state machine $M=(Q, \Sigma, \Delta, \sigma, q0)$ whose behavior is a monitor of certain interface logic where Q contains all normal states along with two extra special states *vio* and *dc*, $\Sigma = I_{ctrl} \cup O_{ctrl}$, and $\Delta = \phi$. Note that, unlike typical functional monitor designs, the output set Δ of a spec FSM is empty. Because there is no need for extra outputs to indicate whether the DUV behavior is legal, illegal, or don't-care. Instead, the special states can indicate these situations as well. More details about how to construct a spec FSM are similar to those in [3].

2.5.1 A spec FSM example

A part of the spec FSM for the AMBA AHB slave protocol is given in Figure 2-4 as an example. Other parts are not shown for clean appearance. In the state *idle/busy*, if *HREADY* is not asserted or *HRESP* is not set to *OKAY*, the spec FSM moves to the state *vio* (*e9*). This implies that a slave cannot respond anything but *OKAY* to an *IDLE* or *BUSY* transfer, which is explicitly defined in the AMBA specification. In addition, in the state *orig*, if a transfer is initiated by asserting *HSEL* and *HREADYin* as well as setting *HTRANS* to *SEQ*, the spec FSM moves to the state *dc* (*e3*). This infers that the master should never set *HTRANS* to *SEQ* for the first transfer, which is an input constraint to the slave. These inputs can be treated as don't-cares since they are not supposed to appear.

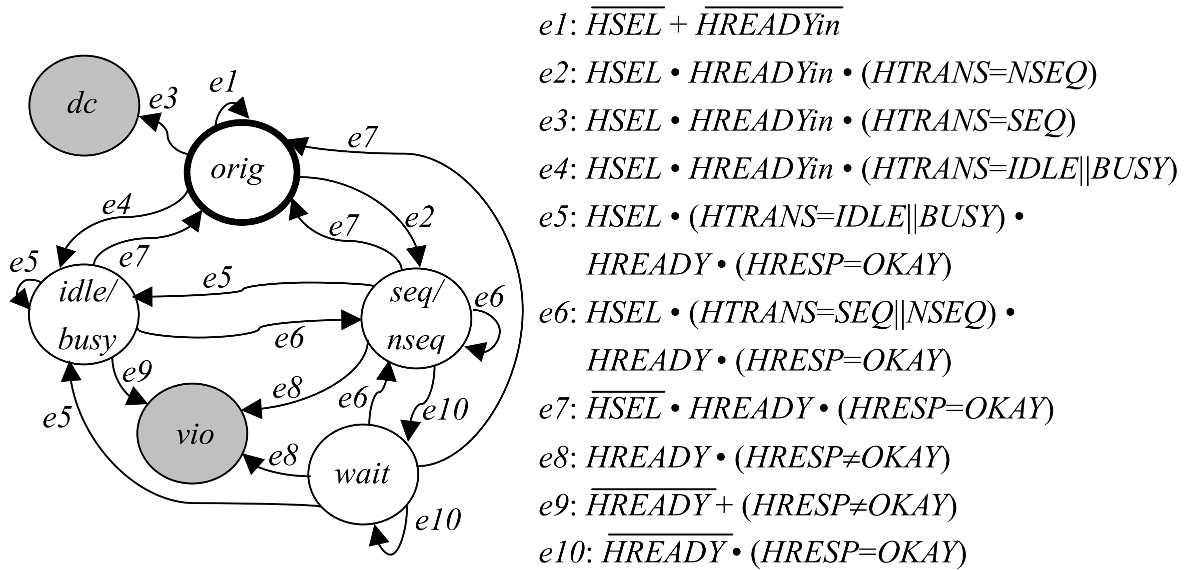


Figure 2-4. A part of the spec FSM of the AMBA AHB slave protocol.

2.5.2 Correctness of the spec FSM

We find a simple but effective method to check whether the given spec FSM is correct. The proposed method does not intend to guarantee that the spec FSM would completely and correctly represent the protocol. Instead, it is able to point out obvious errors in the spec FSM.

There are two rules to check the given spec FSM:

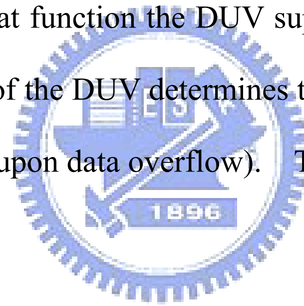
1. All input combinations of a normal state must be fully specified.
2. The outgoing edges of a normal state must be mutually disjoint.

These rules are not advanced but trivial. They are simply the rules for the fully specified deterministic FSM. But they do help a lot in practice. In our experiments, this check is done by our program automatically. It catches several bugs in the first version of the spec FSMs of AHB slave and WISHBONE slave protocols. These results are shown in Chapter 5.

2.5.3 Advantages

To translate an interface specification from a document into a spec FSM is relatively systematic than into rule-based properties (in CTL or emerging property languages). While building the spec FSM, all possible combinations of I_{ctrl} and O_{ctrl} are examined for each normal state since all normal states must be fully specified. This means that all possible situations of each state are taken into consideration. For property-based methods, however, it is hard to determine whether all properties are completely identified or not.

Nevereless, translation from specification documents into formal forms is really not an easy work no matter in which approach. But in our approach, this translation can be done once and for all, no matter what function the DUV supports (for example, retry-capable or not) or what internal situation of the DUV determines the bus outputs (for example, respond ERROR upon address error or upon data overflow). This greatly facilitates the verification flow.



2.5.4 Limitations of expression power

Although the spec FSM provides a systematic way for property specification, it is not omnipotent. For example, a typical liveness property, which says "*ack* should always be asserted at some time after *req* is asserted", cannot be explicitly represented in the spec FSM. Because it takes infinite states to represent the infinite future cycles. But if we are able to set a deadline for such liveness property, it may be clearly specified in some way. For example, the property "*ack* should always be asserted within 16 cycles after *req* is asserted" can be easily represented. This is the case for most interface hardware designs since the hardware cannot be designed to respond in infinite future.

Despite this limitation, our approach is still valuable. All other specification

languages have their own advantages and limitations. But our method is suitable for interface protocols since the interface logics are mostly control FSMs.

2.5.5 Compliance verification with the spec FSM

By introducing the spec FSM, a DUV is said to be *compliant* with the specification if and only if there exists no valid input sequence (of any arbitrary length) along with the corresponding DUV output sequence that can drive the spec FSM into its state *vio*.

Chapter 3

Our Approach

3.1 Problem formulation

Now the problem of interface compliance verification can be interpreted as the compliance verification between the DUV FSM and the spec FSM. It can be further formulated as:

Given the spec FSM $M_s = (Q_s, \Sigma_s, \phi, \sigma_s, qs0)$

where $\Sigma_s = I_{ctrl} \cup O_{ctrl}$

and the DUV FSM $M_d = (Q_d, \Sigma_d, \Delta_d, \sigma_d, qd0)$

where $\Sigma_d = I_{bus} \cup I_{core}$

and $\Delta_d = O_{bus} \cup O_{core}$,

verify if the DUV FSM complies with the spec FSM.

For the spec FSM, $\Sigma_s = I_{ctrl} \cup O_{ctrl}$ infers that it considers only the signals that affect the bus behavior. Other signals like the address or data bus can be neglected to reduce the complexity. For the DUV FSM, $\Sigma_d = I_{bus} \cup I_{core}$ and $\Delta_d = O_{bus} \cup O_{core}$ infers that it can contain self-defined core signals along with bus signals. Therefore, the DUV FSMs of the same protocol may contain different functional logic because of different core functions. But our algorithm can verify only the protocol part and cleverly neglect the functional logic that differs from design to design.

3.2 Observations

Figure 3-1 shows the FSM of an example AHB slave interface design. Its outputs in Figure 3-1 from left to right are *HREADY*, *HRESP[1]*, *HRESP[0]*. When it detects a write request from a master, it moves to the state *write* and responds *OKAY* to indicate that the write operation is done. When it detects a read request from a master, it first moves to the state *prep* to insert a wait cycle, and then moves to the state *read* and responds *OKAY* to indicate that the read operation is done at the next cycle. Otherwise, it stays in the state *sleep* when the slave is not requested or is requested for an *IDLE* or *BUSY* transfer.

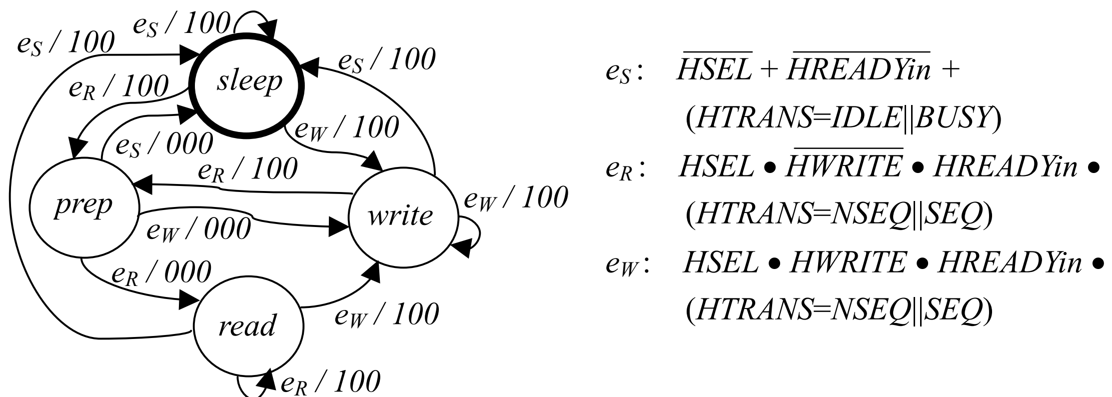


Figure 3-1. The FSM of an AHB slave interface design.

How do we verify if the FSM in Figure 3-1 complies with the protocol in Figure 2-4? Note that this is not simply the FSM equivalence checking problem since these two FSMs are intrinsically different (with different I/O sets). Besides, there is neither a subset nor a superset relation between these two FSMs. However, the states in these two FSMs do have some sort of *corresponding* relations. For example, when the DUV FSM is in the state *sleep*, the spec FSM may be in the state *orig*, because both of them mean the slave is not requested. These two states are named as a *corresponding state pair*.

However, the corresponding relation among states is not always 1-to-1. It can also be n-to-1 or 1-to-n. This is the reason why the DUV FSM is not simply a subset or superset of the spec FSM. For example, the state *orig* and the state *idle/busy* in the spec FSM are both able to correspond to the state *sleep* in the DUV FSM because the DUV responds identically when it is not requested or is requested for an *IDLE* or *BUSY* transfer. In addition, the state *seq/nseq* in the spec FSM is able to correspond to the state *read* and *write* in the DUV FSM because they all represent the data transfer requests.

Since this compliance verification is neither equivalence checking nor subset/superset checking, the problem must be solved by other means. Actually, the discussions in this section suggest that the DUV is proved to be compliant if and only if all possible corresponding state pairs are examined and none of them includes the state *vio*. Hence the issue becomes how to exhaustively explore all possible corresponding state pairs for the given FSMs.

3.3 Solution

An intuitive idea about finding all possible corresponding state pairs is to grow a tree whose nodes are the corresponding state pairs (or *state pairs* in short). The root node

consists of the two initial states. If there exist certain values of inputs and outputs that make the DUV FSM and the spec FSM move from the initial state to the state A and B, respectively, then node (A, B) is produced as a child of the root node.

Figure 3-2 illustrates the tree-growing process. *node1* is the state pair of the initial states in Figure 2-4 and Figure 3-1. Consider the outgoing edge e_2 of *orig* in Figure 2-4 and the outgoing edge $e_W/100$ of *sleep* in Figure 3-1, the intersection of the Boolean functions of these two edges ($f_{orig,seq/nseq} \bullet f_{sleep,write}$) is not equal to 0.

($f_{orig,seq/nseq} = HSEL \bullet HREADYin \bullet (HTRANS=NSEQ)$

$f_{sleep,write} = HSEL \bullet HWRITE \bullet HREADYin \bullet (HTRANS=NSEQ||SEQ) \bullet HREADY \bullet (HRESP=OKAY)$

=> $f_{orig,seq/nseq} \bullet f_{sleep,write}$

$= HSEL \bullet HWRITE \bullet HREADYin \bullet (HTRANS=NSEQ) \bullet HREADY \bullet (HRESP=OKAY)$

$\neq 0$)

This infers that there exists certain input along with the corresponding output that can simultaneously drive both transitions. For example, the input "*HSEL*=*HWRITE*=*HREADYin*=1, *HTRANS*=*NSEQ*", which drives e_w, along with the output *100*, can drive e_2 as well. Hence, *node5* (*seq/nseq*, *write*) is created as a child of *node1*. Similarly, all outgoing edges of *orig* versus all outgoing edges of *sleep* have to be considered in the above manner. Then we can get all children of *node1* as shown in Figure 3-2.

In this way, the process exhaustively grows all children and all grandchildren of the root node and so on. Finally all possible state pairs are present as nodes in this tree. However, this tree can have an infinite depth and thus the tree-growing process seems infeasible. Therefore, certain bounding condition is required to prune the tree to be finite without losing any possible corresponding state pair.

In fact, we can stop growing children of a node if this node has been present in the tree.

For example, in Figure 3-2, we do not have to grow children of *node2* since they should be the same as those of *node1*. This bounding condition does not fail to find any possible state pair. Because the sub-trees growing from *node1* is identical to those growing from *node2*, all possible state pairs growing from *node2* can also be found in those growing from *node1*. Hence, growing the tree rooted from *node2* is completely unnecessary.

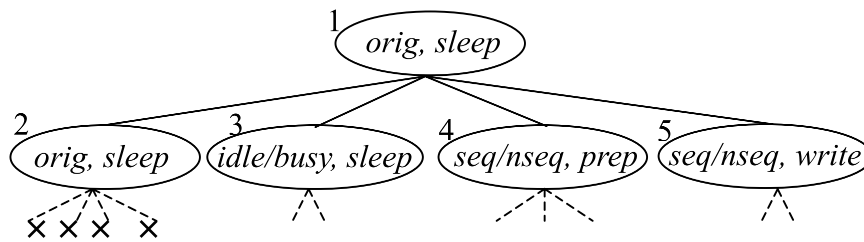
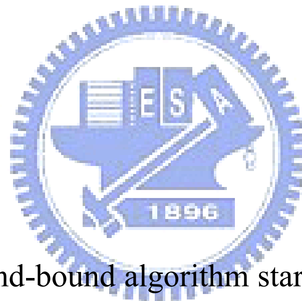


Figure 3-2. The tree-growing process.

3.4 Algorithm



In summary, our branch-and-bound algorithm starts to grow a tree from the initial state pair. It keeps growing child nodes for each node unless the node has been present in the tree or a violation to the protocol is found. The formal description of the algorithm is shown in the next page. In the algorithm, *S* denotes the stack containing all nodes that have been present in the tree. *qs* and *qd* denote the states of the spec FSM and the DUV FSM, respectively. *qs0* and *qd0* are the initial states. The function *grow_tree* performs the branch-and-bound operations on the node. The equation $f_{qs,qi} \bullet f_{qd,qj} \neq 0$ means the intersection of the Boolean functions of the selected edges is nonzero. That is, there exists certain I/O value that can make the spec FSM and the DUV FSM move from the current states, *qs* and *qd*, to the next states, *qi* and *qj*, respectively. If this condition holds and the next state pair has not been present in the tree (i.e., $(qi,qj) \notin S$), the next state pair *(qi,qj)* is created as a child node of the current state pair *(qs,qd)* by performing the function

grow_tree(qi,qj). Note that if the next state of the spec FSM is *vio* (i.e., *qi*=*vio*), the tree-growing process stops and the error trace is given as a counterexample. Furthermore, if the next state of the spec FSM is *dc* (i.e., *qi*=*dc*), the function *grow_tree(qi,qj)* is not performed since the input value that drives the transitions from *qs* and *qd* to *qi* and *qj* is illegal, which means this condition is not supposed to take place.

```
S=φ ;
grow_tree(qs0 ,qd0) ;

grow_tree (qs ,qd) {
        S = S ∪ (qs ,qd) ;
        for i=1 to |e_qs|                  // for all outgoing edges of the state qs
                for j=1 to |e_qd|          // for all outgoing edges of the state qd
                        if (  f_qs,qi • f_qd,qj ≠0 && (qi,qj) ∉S  )
                                if (qi≠dc && qi≠vio )
                                        grow_tree(qi ,qj) ;
                                else if (qi == vio)
                                        give a counterexample and exit;
}
```

3.5 Complexity analysis

3.5.1 Time complexity

The time complexity is estimated by the iteration count in the tree:

$$\text{iteration count} = \sum_{n=1}^{N} (|e_{qs,n}| \times |e_{qd,n}|) , \tag{3-1}$$

$$\text{time complexity} = O\left(\sum_{n=1}^{N} (|e_{qs,n}| \times |e_{qd,n}|) \right) . \quad (3\text{-}2)$$

In the above equations, $|e_{qs,n}|$ and $|e_{qd,n}|$ denote $|e_{qs}|$ and $|e_{qd}|$ at the n-th recursion, and N denotes the number of recursion times. For the worst case,

$$N = |Q_s| \times |Q_d|, \quad (3\text{-}3)$$

$$|e_{qs,n}| = |Q_s|, \quad (3\text{-}4)$$

$$|e_{qd,n}| = |Q_d|, \quad (3\text{-}5)$$

$$\Rightarrow \text{worst-case iteration count} = (|Q_s| \times |Q_d|)^2 , \quad (3\text{-}6)$$

$$\Rightarrow \text{worst-case time complexity} = O((|Q_s| \times |Q_d|)^2) . \quad (3\text{-}7)$$

Equation (3-3) holds only when all combinations of states in two FSMs are valid state pairs. Equation (3-4) and (3-5) hold only when the graph representations of two FSMs are complete graphs. However, these worst-case conditions rarely occur. Actually, experimental results show that the iteration count is typically far lower than this upper bound.

3.5.2 Space complexity

In the above algorithm, the space complexity is dominated by the size of two stacks. One is the *visited-node stack*, i.e., S in the algorithm. The other is the *error-trace stack*, which is not explicitly specified in the algorithm. The visited-node stack contains the corresponding state pairs that are already visited. The error-trace stack contains the I/O values and state conditions from the initial node to the node that the violation occurs. Note that the visited-node stack is always needed, while the error-trace stack is used only when a

violation occurs.

The size of the visited-node stack is affected by the number of corresponding state pairs, where as the size of the error-trace stack is affected by the depth of the violation. But anyhow, the worst-case space complexity of both stacks can be expressed as:

$$\text{worst-case space complexity} = O(\ |Q_s| \times |Q_d|\ ). \tag{3-8}$$

Because $|Q_s| \times |Q_d|$ is equal to the maximum number of state pairs, and also equal to the maximum depth of the tree.

Chapter 4

Extension

4.1 Spec EFSM

E S A
1896

Timing constraints are not uncommonly encountered in interface protocols. However, they cannot be easily specified in spec FSM sometimes. For example, a simple interface protocol, which defines "*ack* must be asserted within 16 cycles after *req* is asserted", is specified as the spec FSM in Figure 4-1(a). It requires so many states to represent such a simple protocol. Instead, if we specify this protocol in EFSM by introducing a variable *count* as in Figure 4-1(b), the representation becomes much clearer and easier.

Similar to the spec FSM, the spec EFSM is an EFSM M=(Q, Σ, Δ, x, T, $q0$, $x0$) whose behavior is a monitor of certain interface logic where Q contains all normal states along with two extra special states *vio* and *dc*, $\Sigma = I_{ctrl} \cup O_{ctrl}$, and $\Delta = \phi$. The predicates and actions provide a convenient way to specify timing constraints that are frequently encountered in interface protocols.

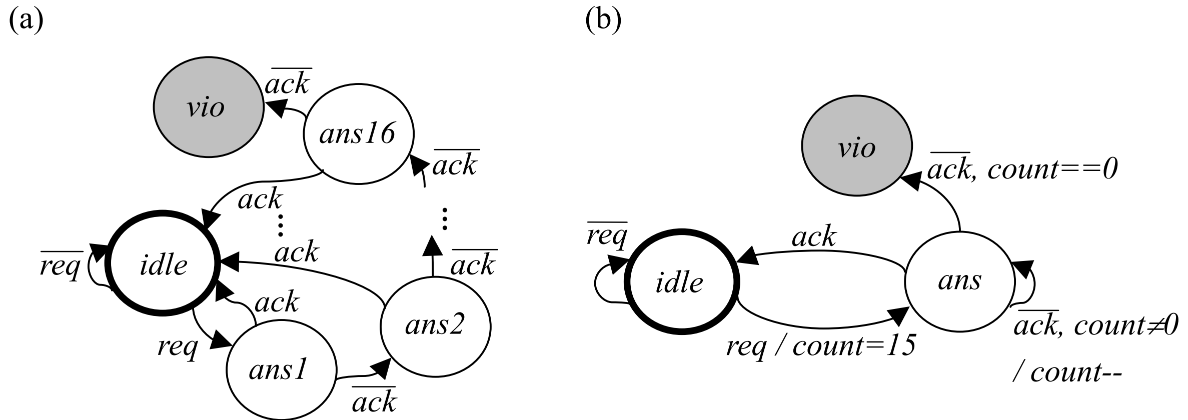


Figure 4-1. Specify the same protocol in (a) FSM and (b) EFSM

4.2 Extended algorithm

Meanwhile, the original algorithm should also be extended to handle the spec EFSM. Similarly, the extended algorithm grows a tree from each node unless the node has been present in the tree. The only difference now is that the node contains not only a state pair but also the values of the corresponding variables. A simple example is given in Figure 4-2. Figure 4-2(a) gives an erroneous implementation of the interface protocol in Figure 4-1. This design violates the protocol because it may not assert *ack* within 16 cycles if it sticks in the state *wait*. One possible scenario for the tree-growing process to find the violation is shown in Figure 4-2(b). The words in each node from left to right denote the spec EFSM state, the DUV state, and the value of the variable *count*. The nodes in the right branch illustrate the difference between the original algorithm and the extended one. As shown, the extended algorithm does not stop at the node (*ans*, *wait*, *14*) although this node has the same state pair as its parent node (*ans*, *wait*, *15*). The reason is that the extra variable *count* needs to be considered as well.

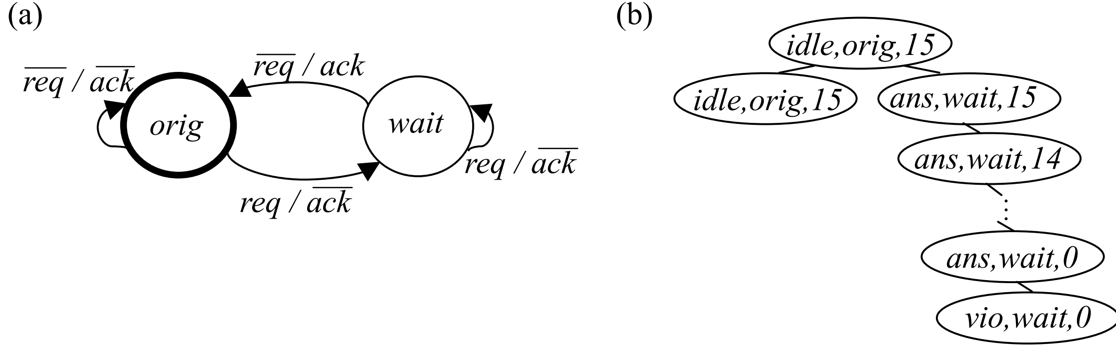


Figure 4-2. (a) A wrong implementation of the protocol in Figure 4-1 and (b) the tree generated by the extended algorithm.

The extended algorithm is given below (the modified parts are shaded). Similar to the algorithm in Section 3.4, *S* contains the nodes that have been present in the tree, and the function *extended_grow_tree* performs the branch-and-bound operations. But some extension is made here. The node now contains not only *qs* and *qd* but also *x*, the variable value because the variable value has to be considered as well. Moreover, in the function *extended_grow_tree*, the condition that a child node is created includes not only $f_{qs,qi} \bullet f_{qd,qj} \neq 0$ and $(qi, qj, x') \notin S$, but also $P_{qs,qi}(x)==true$, which means the predicate of current transition edge must be evaluated true. It is because the transitions can never take place if the predicate is not evaluated true.

Given the spec EFSM $M_s = (Q_s, \Sigma_s, \phi, x, T, qs0, x0)$

where $\Sigma_s = I_{ctrl} \cup O_{ctrl}$

and the DUV FSM $M_d = (Q_d, \Sigma_d, \Delta_d, \sigma_d, qd0)$

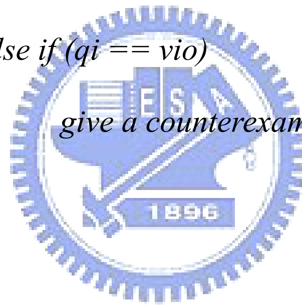
where $\Sigma_d = I_{bus} \cup I_{core}$

and $\Delta_d = O_{bus} \cup O_{core}$,

verify the compliance with the following algorithm:

```
S=φ ;
extended_grow_tree(qs0 ,qd0, x0) ;

extended_grow_tree (qs, qd, x) {
    S = S ∪ (qs, qd, x) ;
    for i=1 to |e_qs|                     // for all outgoing edges of the state qs
        for j=1 to |e_qd| {              // for all outgoing edges of the state qd
            x'= A_qs,qi(x);              // perform the action to the variable
            if (f_qs,qi•f_qd,qj ≠0 && P_qs,qi(x)==true && (qi, qj, x') ∉S)
                if (qi≠dc && qi≠vio )
                    extended_grow_tree(qi, qj, x') ;
                else if (qi == vio)
                    give a counterexample and exit ;
        }
}
```



4.3 Complexity analysis

4.3.1 Time complexity

The time complexity analysis of the extended algorithm is similar to the discussions in Section 3.5.1. The iteration count is equal to (3-1). That is,

$$\text{iteration count} = \sum_{n=1}^{N} (|e_{qs,n}| \times |e_{qd,n}|) \qquad (3\text{-}1)$$

where $|e_{qs,n}|$ and $|e_{qd,n}|$ denote $|e_{qs}|$ and $|e_{qd}|$ at the n-th recursion,

and N denotes the number of recursion times.

For the worst case,

$N = |Q_s| \times |Q_d| \times |range(A(x))|,$ (4-1)

$|e_{qs,n}| = |Q_s|,$ (4-2)

$|e_{qd,n}| = |Q_d|,$ (4-3)

=> worst-case iteration count = $|range(A(x))| \times (|Q_s| \times |Q_d|)^2$, (4-4)

=> worst-case time complexity = $O(\ |range(A(x))| \times (|Q_s| \times |Q_d|)^2\ )$. (4-5)

The term *range(A(x))* denotes the value range of the function *A(x)*, and thus |*range(A(x))*| denotes the number of possible values of *x*. Equation (4-1) holds when all combinations of two states and variable values are possible, and equation (4-2) and (4-3) holds when the graphic representations of two FSMs are complete graphs. Worst case like this rarely happens.

How exactly does the complexity change after this extension? Let us compare (4-5) to (3-7), although (4-5) is multiplied by a factor of |*range(A(x))*|, $|Q_s|$ of (4-5) is greatly reduced in general. Hence, the complexity does not increase significantly as expected.

4.3.2 Space complexity

Since the original state pair (*qs*, *qd*) is modified as (*qs*, *qd*, *x*), the original maximum number of state pairs $|Q_s| \times |Q_d|$ should be modified as $|range(A(x))| \times |Q_s| \times |Q_d|$ as well. This is the same case for the maximum depth of the tree. These modifications directly affect the maximum size of the visited-node stack and the error-trace stack, which further changes the worst-case space complexity. Therefore, the worst-case space complexity becomes:

worst-case space complexity = O($|range(A(x))| \times |Q_s| \times |Q_d|$). (4-6)

4.4 Other extensions

Except for the EFSM, the FSM has other variations, for example, the interacting FSM. Theoretically, these variations can be transformed into the equivalent FSMs. Therefore, they can be candidates of other extensions to our approach. For example, some concurrent behaviors of interface protocols can be modeled by the interacting FSM easily. However, the transformation process imposes a heavy burden on the algorithm but improves only little expressing ability. Therefore, only the EFSM extension is implemented since the trade is worthwhile.

Chapter 5

Experimental Results

5.1 Program implementation

1896

The proposed approach has been implemented in C. Our implementation can either formally prove that the given design is fully compliant with certain interface protocol or report a precise input sequence as a counterexample to show how the given design fails in the compliance verification. Additionally, the program can automatically check the correctness of the spec (E)FSM with the method in Section 2.5.2.

5.1.1 Input file format

Both the spec (E)FSM and the DUV FSM are accepted in the enhanced BLIF(Berkeley Logic Interchange Format) [26]. This format extends the BLIF with some syntax capable to describe the reasons of each transition as well as predicates and actions required by the EFSM model. Figure 5-1 shows an example in this format. The words after the character

are comments.   These comments clearly explain the syntax.

```
.model     the_req_ack_protocol        #model name
.inputs    req ack                     #input names
.variables count 0                     #variable and its initial value

.start_kiss
.i 2          #input number
.o 0          #output number
.s 3          #state number
.r idle       #initial state

#input value  c_st   n_st   reason                 predicate    action

0-            idle   idle   Not_Requested
1-            idle   ans    Receive_Request        NULL         count = 15
-1            ans    idle   Acknowledge
-0            ans    ans    Not_Acknowledge_Yet    count != 0   count - 1
-0            ans    vio    Ack_Exceed_16cycles    count == 0

.end_kiss
.end
```

Figure 5-1. The spec EFSM of Figure 4-1 in the enhanced BLIF.

The `reason` column in the enhanced BLIF provides a way to specify the meaning of each transition.   With these reasons, we can provide more meaningful counterexamples to improve the compliance verification.

5.1.2 Input file check

Our implementation can check the input files with the schemes described in Section 2.5.2.   In the experiments, it really catches several bugs in the original spec FSMs of the

AHB slave and the WISHBONE slave protocols. One of the bugs is shown in Figure 5-2. This bug is arisen from omitting the edge *e4* in Figure 2-4. That is, this transition is accidentally ignored in the original spec FSM. If we do not check with this method and neglect this transition in the spec FSM, the verification result may not be correct.

```
********************************************
 Result of pre-processing
********************************************
 Pre-process file golden_AHB_slave.txt ...
       ERROR!In state orig :
         Input 1100000 not specified!
 Please modify input file golden_AHB_slave.txt!

 Verification aborted! No result reported.
```

Figure 5-2. The report of catching a bug in the spec FSM of AHB slave protocol.

5.1.3 Counterexample

Our implementation gives very useful error-trace messages. It tells not only the I/O values and the state transitions, but also the reason of each transition. The error-trace message when verifying the design in Figure 4-2(a) is given in Figure 5-3. As we can see,

```
********************************************
 Result of this compliance verification
********************************************
  DESIGN VIOLATES SPEC. : Ack_Exceed_16cycles

  DUV input names:   req
  DUV output names:  ack

  Error trace shown below:
    c_stSPEC(DUV)      input   output n_stSPEC(DUV)  reason
    -------------------------------------------------------------------------
    idle(orig)          1       0      ans(wait)      Receive_Request
    ans(wait)           1       0      ans(wait)      Not_Acknowledge_Yet
    ans(wait)           1       0      ans(wait)      Not_Acknowledge_Yet
    ans(wait)           1       0      ans(wait)      Not_Acknowledge_Yet
    ans(wait)           1       0      ans(wait)      Not_Acknowledge_Yet
    ans(wait)           1       0      ans(wait)      Not_Acknowledge_Yet
    ans(wait)           1       0      ans(wait)      Not_Acknowledge_Yet
    ans(wait)           1       0      ans(wait)      Not_Acknowledge_Yet
    ans(wait)           1       0      ans(wait)      Not_Acknowledge_Yet
    ans(wait)           1       0      ans(wait)      Not_Acknowledge_Yet
    ans(wait)           1       0      ans(wait)      Not_Acknowledge_Yet
    ans(wait)           1       0      ans(wait)      Not_Acknowledge_Yet
    ans(wait)           1       0      ans(wait)      Not_Acknowledge_Yet
    ans(wait)           1       0      ans(wait)      Not_Acknowledge_Yet
    ans(wait)           1       0      ans(wait)      Not_Acknowledge_Yet
    ans(wait)           1       0      ans(wait)      Not_Acknowledge_Yet
    ans(wait)           1       0      vio(wait)      Ack_Exceed_16cycles


Time elapsed: < 1 sec.
```

Figure 5-3. The verification result of the design in Figure 4-2(a).

the `reason` column tells the physical meaning of the current status. These reasons, along with the detail I/O sequence and state transitions, greatly facilitates the debugging of the interface design.

5.2 Result analysis

In order to investigate the effectiveness and efficiency of the proposed algorithm, compliance verification is conducted over a set of real AMBA AHB-compliant and

WISHBONE-compliant [27] designs. In addition, to check whether the proposed algorithm can find the design flaws as expected, some errors are intentionally injected into the design *con7* and *mac* as two additional benchmark designs *con7_err* and *mac_err*, respectively. The experimental results are shown in Table 1.

There are some facts worthy to be mentioned in Table 1. As it is shown, all the designs under verification support different functional modes of the protocols, but the same spec (E)FSM can be used to verify all designs of the same protocol without altering. Furthermore, the two injected errors are successfully found. The error in the design *con7_err* is caused by a self-loop of the state performing the WAIT response. Thus the design may respond WAIT more than 16 cycles, which is not recommended in the AHB protocol. This is an error that designers are very likely to commit if they don't deal with the WAIT response carefully. The other error in the design *mac_err* is a little more complex. When an IDLE transfer is initiated after an ERROR response, the design does not respond OKAY but respond ERROR instead, which causes a violation to the AHB protocol. This error is not uncommon because the two-cycle ERROR response is intrinsically more complicated and error-prone. With our verification approach, these errors and the reasons leading to them are clearly identified.

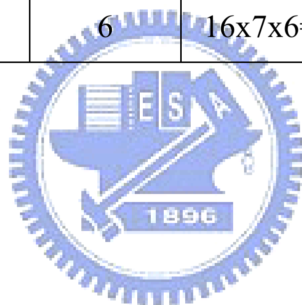
Table 1. The DUVs and the verification results.

I/F protocol type	DUV	result	supporting function	reason of violation
WISHBONE slave (spec FSM)	*spi* [28]	compliance	NORMAL and ERROR response	-
	ac3 ctrl[28]	compliance	NORMAL response	-
AMBA AHB slave (spec EFSM)	*con7*	compliance	OKAY and WAIT response	-
	mac	compliance	OKAY and ERROR response	-
	remap	compliance	OKAY, ERROR, and RETRY response	-
	con7_err	violation	OKAY and WAIT response	Wait>16cycles
	mac_err	violation	OKAY and ERROR response	Not_Respond_Okay

The complexity comparison is shown in Table 2. It clearly demonstrates that our algorithm can correctly complete the formal compliance verification for all given designs. The results also indicate that the actual time and space requirements are far less than the estimated ones from the worst-case analysis. As a matter of fact, each verification run listed in Table 1 finishes within just 1 second. It shows that our algorithm is capable of completing the formal compliance verification in reasonable time. Though we cannot find even larger real designs for investigation, we believe our algorithm can handle FSMs containing hundreds of states that are more complicated than FSMs of most practical designs.

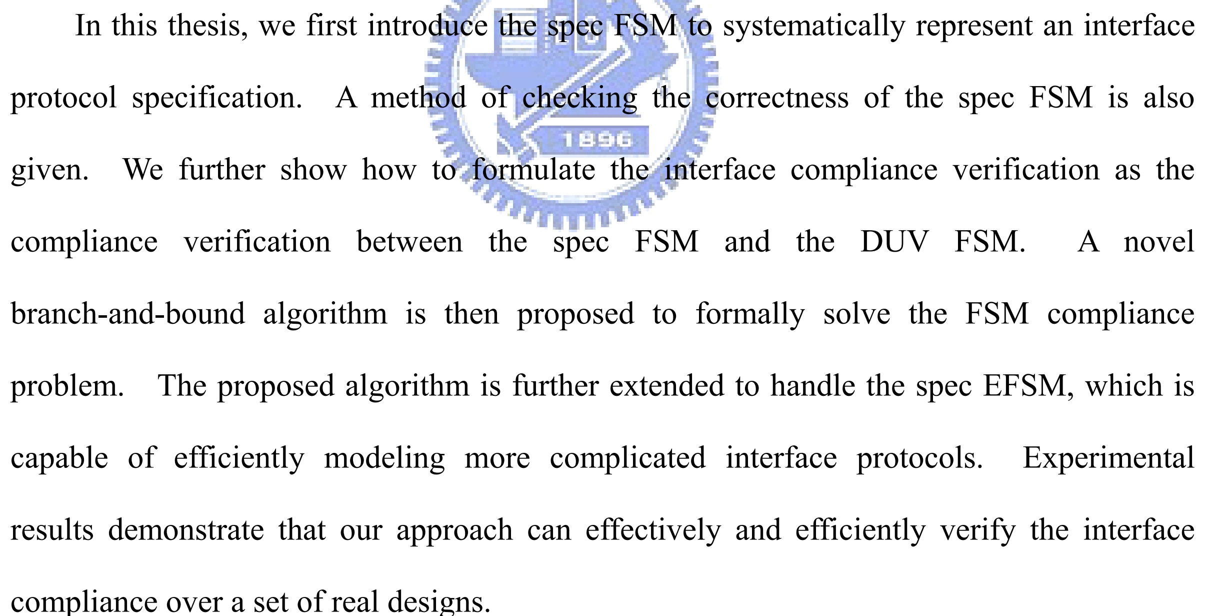
Table 2. Complexity comparison.

I/F protocol type	DUV	stack size ($\|S\|$)	$\|range(A(x))\| \times \|Q_s\|\times\|Q_d\|$	iteration count	$\|range(A(x))\|\times (\|Q_s\|\times\|Q_d\|)^2$
WISHBONE slave (spec FSM)	*spi* [28]	14	7x3=21	180	$(7\times3)^2$=442
	ac3 ctrl [28]	23	7x5=35	221	$(7\times5)^2$=1225
AMBA AHB slave (spec EFSM)	*con7*	11	16x7x4=448	204	$16\times(7\times4)^2$=12544
	mac	8	16x7x6=672	191	$16\times(7\times6)^2$=28224
	remap	8	16x7x6=672	136	$16\times(7\times6)^2$=28224
	con7_err	20	16x7x4=448	42	$16\times(7\times4)^2$=12544
	mac_err	6	16x7x6=672	57	$16\times(7\times6)^2$=28224



Chapter 6

Conclusions and Future Work



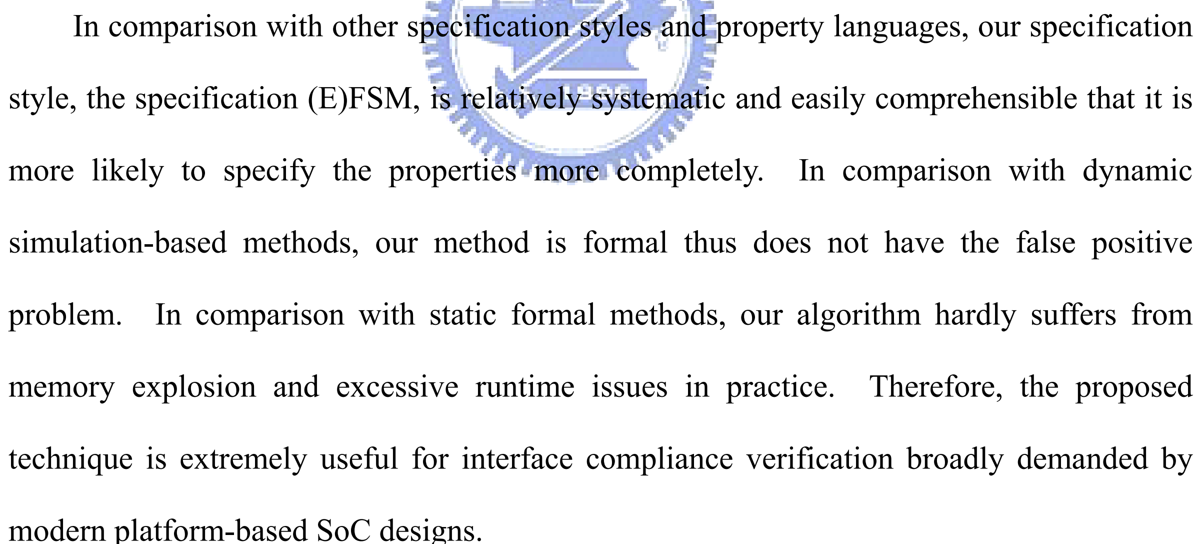
In this thesis, we first introduce the spec FSM to systematically represent an interface protocol specification. A method of checking the correctness of the spec FSM is also given. We further show how to formulate the interface compliance verification as the compliance verification between the spec FSM and the DUV FSM. A novel branch-and-bound algorithm is then proposed to formally solve the FSM compliance problem. The proposed algorithm is further extended to handle the spec EFSM, which is capable of efficiently modeling more complicated interface protocols. Experimental results demonstrate that our approach can effectively and efficiently verify the interface compliance over a set of real designs.

6.1 Contributions

The main contributions of this work are summarized as follows:

- property specification
  1. A method of specifying the interface protocol specification as the specification (E)FSM is proposed.
  2. A simple but effective way to check the correctness of the spec (E)FSM is given.
- verification technique
  1. A reasonable problem formulation that focuses on interface logic at FSM level is proposed.
  2. A formal algorithm is developed for interface compliance verification.

6.2 Comparisons



In comparison with other specification styles and property languages, our specification style, the specification (E)FSM, is relatively systematic and easily comprehensible that it is more likely to specify the properties more completely. In comparison with dynamic simulation-based methods, our method is formal thus does not have the false positive problem. In comparison with static formal methods, our algorithm hardly suffers from memory explosion and excessive runtime issues in practice. Therefore, the proposed technique is extremely useful for interface compliance verification broadly demanded by modern platform-based SoC designs.

6.3 Future work

Our approach may probably be enhanced to verify designs in different abstract level. For example, it can be used to verify RTL designs by extracting FSMs from RTL codes.

But the overhead introduced by transforming between different abstract levels needs to be considered carefully. Moreover, different verification schemes using the same spec (E)FSM can be developed. For example, in addition to this formal approach, the spec (E)FSM may turn into a monitor in simulation-based verification. Combining both simulation-based and formal approaches with the same spec (E)FSM will greatly facilitate the verification.

References

[1] VSI Alliance, *Virtual Component Interface (VCI) Standard - OCB 2 1.0*, http://www.vsia.org, Mar. 2000.

[2] Kanna Shimuzu, David L. Dill, and Alan J. Hu, "Monitor-Based Formal Specification of PCI," in *Proceedings of the 3th International Conference on Formal Methods in Computer-Aided Design*, Nov. 2000, pp. 335-353.

[3] Hue-Min Lin, Chia-Chih Yen, Che-Hua Shih, and Jing-Yang Jou, "On Compliance Test of On-Chip Bus for SOC," in *Proceedings of the Asia and South Pacific Design Automation Conference*, Jan. 2004, pp. 328-333.

[4] Marcio T. Oliviera and Alan J. Hu, "High-Level Specification and Automatic Generation of IP Interface Monitors," in *Proceedings of the 39th Design Automation Conference*, June 2002, pp. 129-134.

[5] Alan J. Hu, Jeremy Casus, and Jin Yang, "Efficient Generation of Monitor Circuits for GSTE Assertion Graphs," in *Proceedings of the 2003 IEEE/ACM International Conference on Computer-Aided Design*, Nov. 2003, pp. 154-159.

[6] Jun Yuan, Kurt Shultz, Carl Pixley, Hillel Miller, and Adnan Aziz, "Modeling Design Constraints and Biasing in Simulation Using BDDs," in *Proceedings of the 1999 IEEE/ACM International Conference on Computer-Aided Design*, Nov. 1999, pp. 584-589.

[7] Kanna Shimizu and David L. Dill, "Deriving a Simulation Input Generator and a Coverage Metric From a Formal Specification," in *Proceedings of the 39th Design Automation Conference*, June 2002, pp. 801-806.

[8] Serdar Tasiran, Yuan Yu, and Brannon Baston, “Using a Formal Specification and a Model Checker to Monitor and Direct Simulation,” in *Proceedings of the 40th Design Automation Conference*, June 2003, pp. 356-361.

[9] Andy Nightingale and John Goodenough, “Testing for AMBA™ Compliance,” in *Proceedings of the 14th Annual IEEE International ASIC/SOC Conference*, Sept. 2001, pp. 301-305.

[10] ARM Limited, *AMBA Specification (Rev 2.0)*, 13 May 1999.

[11] http://www.synopsys.com/products/designware/docs/vip/

[12] Pankaj Chauhan, Edmund M. Clarke, Yuan Lu and Dong Wang, “Verifying IP-Core Based System-On-Chip Designs,” in *Proceedings of the 12th Annual IEEE International ASIC/SOC Conference*, Sept. 1999, pp. 27-31.

[13] Ilan Beer, Shoham Ben-David, Cindy Eisner, Yechiel Engel, Raanan Gewitzman and Avner Landver, “Establishing PCI Compliance Using Formal Verification: A Case Study,” in *Proceedings of the 14th International Phoenix Conference on Computation and Communications*, Mar. 1995, pp. 373-377.

[14] Abhik Roychoudhury, Tulika Mitra, and S.R. Karri, “Using Formal Techniques to Debug the AMBA System-on-Chip Bus Protocol,” in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, 2003, pp. 828-833.

[15] Lubomir Ivanov and Ramakrishna Nunna, “Specification and Formal Verification of Interconnect Bus Protocols,” in *Proceedings of the 43rd IEEE Midwest Symposium on Circuits and Systems*, Aug. 2000, pp. 378-382.

[16] K. L. McMillan, *Symbolic Model Checking*, Kluwer Academic Publishers, 1993.

[17] http://www.eda.org/vfv/docs/psl_lrm-1.01.pdf/.

[18] http://www.verificationlib.org/.

[19] http://www.opervera.org/.

[20] http://www.systemverilog.org/.

[21] Yatin V. Hoskote, Jacob A. Abraham, and Donald S. Fussell, "Automated Verification of Teporal Properties Specified as State Machines in VHDL," in *Proceedings of the 5th Great Lakes Symposium on VLSI*, Mar. 1995, pp. 100-105.

[22] Annette Bunker and Ganesh Gopalakrishnan, "Using Live Sequence Charts for Hardware Protocol Specification and Compliance Verification," in *Proceedings of the IEEE International High Level Design Validation and Test Workshop*, Nov. 2001, pp. 95-100.

[23] Annette Bunker, Ganesh Gopalakrishnan, and Sally A. McKee, "Formal Hardware Specification Languages for Protocol Compliance Verification," *ACM Transactions on Design Automation of Electronic Systems*, vol. 9, no. 1, Jan. 2004.

[24] Kanna Suimizu, David L. Dill, and Ching-Tsun Chou, "A Specification Methodology by a Collection of Compact Properties as Applied to the Intel® Itanium™ Processor Bus Protocol," in *Proceedings of the 11th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, Sept. 2001, pp. 340-354.

[25] Cedric Besse and Ana Cavalli, "An Automatic and Optimized Test Generation Technique Applying to TCP/IP Protocol," in *Proceedings of the 14th IEEE International Conference on Automated Software Engineering*, Oct. 1999, pp. 73-80.

[26] University of California Berkeley, *Berkeley Logic Interchange Format (BLIF)*, Sept. 1996.

[27] OpenCores Organization, *Specification for the: WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores*, Rev. B.3, 2002.

[28] http://www.opencores.org/.

Vita

Ya-Ching Yang was born in Taipei, Taiwan on February 9, 1980. She received the B.S. degree in Electronics Engineering from National Chiao Tung University in June 2002. From September 2002, she is a graduate student of Professor Jing-Yang Jou in the Institute of Electronics, National Chiao Tung University. Her research interests include verification and Electronic Design Automation (EDA). She received the M.S. degree from National Chiao Tung University in June 2004.