JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 26, 1601-1617 (2010)

FSM-Based Formal Compliance Verification of Interface Protocols*

CHE-HUA SHIH+, YA-CHING YANG, CHIA-CHIH YEN, JUINN-DAR HUANG AND JING-YANG JOU
Department of Electronics Engineering
National Chiao Tung University
Hsinchu, 300 Taiwan
+*E-mail: matar@eda.ee.nctu.edu.tw*

Verifying whether a building block conforms to a specific interface protocol is one of the important steps in a platform-based system-on-a-chip design methodology. There are limitations for most of the existing methods for interface protocol compliance verification. Simulation-based methods have the false positive problem while formal property checking methods may suffer from memory explosion and excessive runtime. In this paper, we propose a novel approach for interface protocol compliance verification. The properties of the interface protocol are first specified as a specification FSM. Then the compliance of interface logic is formally verified at the higher FSM level so that the required memory and runtime can be greatly reduced. Finally, it is shown theoretically and experimentally that the proposed algorithm possesses acceptably low time complexity for practical applications.

Keywords: interface compliance verification, functional verification, formal verification, platform-based design methodology, protocol modeling

1. INTRODUCTION

In modern system-on-a-chip (SoC) designs, some building blocks are reusable intellectual property (IP) cores to accelerate the design process. The platform-based design methodology [1], in which all IP cores are pre-verified, is commonly adopted to achieve an even higher level of reusability. Fig. 1 illustrates how to reuse an IP core in typical platform-based designs. An IP core is wrapped with the appropriate interface (I/F) logic complying with certain I/F protocol so that it can concordantly communicate with other IP cores within a system. When the IP core is desired in another platform with a different I/F protocol, all a designer 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 [2]. In addition, even under a given I/F protocol, the I/F logic can still vary significantly due to numerous legal configurations and options. Therefore, the interface compliance must be verified thoroughly during SoC integration.

There are two major categories in the field of interface compliance verification: simulation-based (dynamic) methods and formal (static) ones. The simulation-based verification approach is age-old but popular. In [3], the authors use HDL monitors to represent the

Received October 17, 2008; revised March 9 & June 22, 2009; accepted July 17, 2009.
Communicated by Yao-Wen Chang.
* The previous version of this paper has been presented in the IEEE International Symposium on VLSI Design, Automation, and Test, April 2005, pp. 12-15.

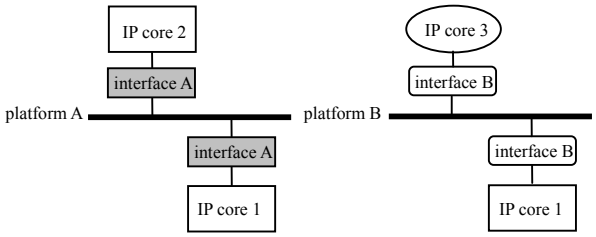


Fig. 1. An IP core reused in different system platforms.

properties of protocol specifications. These HDL monitors are simulated along with the design under verification (DUV) to determine its correctness. In [4-7], methods of generating monitor circuits, coverage metrics, or verification patterns from high-level specification styles are proposed. In [8], a commercial tool ACT is developed to facilitate AMBA [9] compliance verification. In [10], Lin *et al*. specify properties of an I/F protocol as a monitor FSM and then verify the compliance by simulation. All these works define certain coverage metrics to measure the quality of a simulation trace. However, the compliance can never be assured even if 100% coverage is achieved. In general, all simulation-based approaches suffer from this *false positive* problem in common.

Formal verification can avoid such false positive problem. Model checking [11] techniques are used for I/F protocol compliance verification in [12-14]. In these works, the authors use CTL language to describe the properties of I/F protocols. Then the model checker verifies the DUV against these properties. Once the model checker reports a success, the design is fully compliant to these properties. However, properties in CTL are not easily thorough and the process of extracting properties from a specification written in 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 formal verification. Moreover, memory explosion and excessively long runtime may be even serious problems as the design size increases.

Recently, the assertion-based verification (ABV) methodology is getting popular and several property specification languages (such as PSL [15], OVL [16], OVA [17], and SVA [18]) are developed to provide alternative ways to specify properties in addition to CTL. These emerging languages are relatively more understandable than CTL at the semantic and syntactic level. However, no matter which emerging property specification language is selected, either the dynamic ABV or static ABV inherently suffers from the same problems described earlier.

Our approach, unlike any of above, intends to formally verify whether the I/F logic is compliant to the I/F protocol at the FSM level. The properties of the I/F protocol are specified as a specification FSM. There are two reasons why we adopt a specification FSM to represent the interface protocol. First, the FSM-based specification style is adequate for interface protocols since the interface logic is mostly a control FSM. Second, most engineers are familiar to the FSM model. Compared with many property specification languages, engineers can use the spec FSM with a smaller or almost no learning effort. We believe that the FSM style is relatively more readable and systematic than rule-based specification styles and thus enables thorough 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. 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 design flow. Since the verification is done at the higher FSM level and only the separated I/F logic is under verification, our approach can efficiently accomplish the verification even if it is a formal method indeed.

The rest of this paper is organized as follows. Section 2 introduces necessary terminology and concepts of compliance verification. In section 3, the problem formulation and the proposed compliance checking algorithm are described in detail. Section 4 extends the algorithm to handle the compliance verification in which the specification is modeled as an extended FSM (EFSM). Section 5 shows the experimental results, and section 6 concludes this paper.

2. PRELIMINARIES

The FSM model is a common representation for logic design. In this section, we introduce the notations and the spec FSM model used in this paper.

2.1 Notations of Interface Signals

Typical I/O signals of bus interface logic are shown in Fig. 2.

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 from core

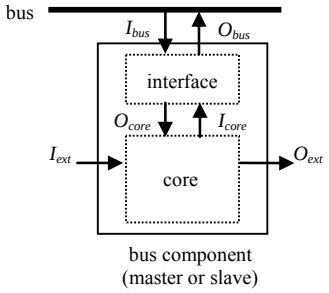


Fig. 2. Notations of bus signals.

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

We use the AMBA AHB slave interface [9] 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 common that just a small portion of bus I/F signals are classified into I_{ctrl} and O_{ctrl}. For example, what value the address/data bus exactly carries does not affect the bus behavior at the protocol level. Also note that I_{core}, O_{core}, I_{ext}, and O_{ext} may differ from design to design.

2.2 FSM

Definition 1 (FSM) An FSM is a quintuple $M = (Q, \Sigma, \Delta, \sigma, q_0)$ where
Q: the set of symbols denoting states
Σ: the set of symbols denoting inputs
Δ: the set of symbols denoting outputs
σ: $Q \times B^{\Sigma} \rightarrow Q \times B^{\Delta}$, the state transition function
q_0: $q_0 \in Q$, the initial state

Additionally,

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

2.3 Specification FSM (Spec FSM)

A design specification defines two important attributes:

1. **Input assumptions:** the valid input space.
2. **Output properties:** the legal output behaviors in the valid input space.

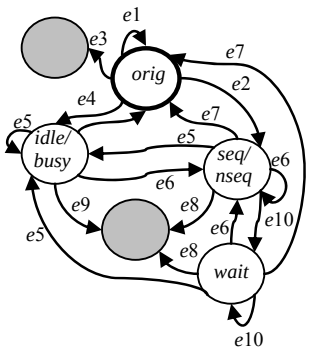
In our approach, the protocol specification is represented with a specification FSM, or spec FSM. Besides input assumptions, it specifies whether the output response of a specific implementation (DUV) is legal under a valid input sequence. In other words, the spec FSM actually acts as a functional monitor of the DUV. An I/O sequence of DUV is classified into one of the following three categories:

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.

Hence, in every spec FSM, two special states are defined: q_{vio} and q_{dc}. The spec FSM moves to the state q_{dc} if an invalid input sequence is applied to the DUV. If the DUV be-

haves illegally under a valid input sequence, the spec FSM moves to the state q_{vio}. If the DUV behaves legally under a valid input sequence, the spec FSM moves among other normal states excluding q_{dc} and q_{vio}. Accordingly, the spec FSM is a transformation based on a typical FSM $M = (Q, \Sigma, \Delta, \sigma, q_0)$ whose behavior is a monitor of certain I/F logic where Q contains all normal states along with two extra special states q_{vio} and q_{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 since there is no need for extra outputs to indicate whether the DUV behavior is legal, illegal, or don't-care. Definition 2 shows the formal description of the spec FSM. More details about how to systematically construct a spec FSM from an interface specification can be found in [10].

Definition 2 (spec FSM) A spec FSM is a 6-tuple $M = (Q, \Sigma, \sigma, q_0, q_{vio}, q_{dc})$ where
Q: the set of symbols denoting states
Σ: the set of symbols denoting inputs
σ: $Q \times B^{\Sigma} \rightarrow Q$, the state transition function
q_0: $q_0 \in Q$, the initial state
q_{vio}: $q_{vio} \in Q$, the illegal state
q_{dc}: $q_{dc} \in Q$, the don't-care state



$e1$: $\overline{HSEL} + \overline{HREADYin}$
$e2$: $HSEL \bullet HREADYin \bullet (HTRANS = NSEQ)$
$e3$: $HSEL \bullet HREADYin \bullet (HTRANS = SEQ)$
$e4$: $HSEL \bullet HREADYin \bullet (HTRANS = IDLE||BUSY)$
$e5$: $HSEL \bullet (HTRANS = IDLE||BUSY) \bullet HREADY \bullet (HRESP = OKAY)$
$e6$: $HSEL \bullet (HTRANS = SEQ||NSEQ) \bullet HREADY \bullet (HRESP = OKAY)$
$e7$: $\overline{HSEL} \bullet HREADY \bullet (HRESP = OKAY)$
$e8$: $HREADY \bullet (HRESP \neq OKAY)$
$e9$: $\overline{HREADY} + (HRESP \neq OKAY)$
$e10$: $HREADY \bullet (HRESP = OKAY)$

Fig. 3. The spec FSM of a simplified AMBA AHB slave protocol.

The spec FSM of a simplified AMBA AHB slave protocol is given in Fig. 3 as an example. 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* through *e*9. 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* through *e*3. This infers that a master should never set *HTRANS* to *SEQ* for the first transfer, which is an input constraint to a slave.

To translate an I/F protocol from a document into a spec FSM is relatively systematic than into rule-based properties. While building the spec FSM, all possible combinations of I_{ctrl} and O_{ctrl} are considered for each normal state, which means all possible transitions of each normal state are fully specified. For rule-based methods, however, it is really hard to determine whether all properties have been completely identified or not.

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

2.4 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 since it takes infinite states to represent the infinite future cycles. But if we are able to set a bound for such liveness property, the spec FSM is capable of specifying the bounded liveness property. For example, the property "*ack* should 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 is hardly designed to respond in infinite future.

Despite this limitation, the FSM-based specification style is still promising. All other specification languages have their own advantages and limitations. But our method is adequate for interface protocols since the interface logic is mostly a control FSM.

3. OUR APPROACH

In this section, we describe the details of the proposed interface compliance verification algorithm with FSM models.

3.1 Problem Formulation

The problem of interface compliance verification can be interpreted as the compliance verification between the spec FSM and the DUV FSM. We formulate it as follows:

Given the spec FSM $M_s = (Q_s, \Sigma_s, \sigma_s, q_{s0}, q_{vio}, q_{dc})$ where $\Sigma_s = I_{ctrl} \cup O_{ctrl}$
and the DUV FSM $M_d = (Q_d, \Sigma_d, \Delta_d, \sigma_d, q_{d0})$ 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.

Fig. 4 shows the DUV FSM of an AHB slave interface design. Its outputs in Fig. 4 from left to right are *HREADY*, *HRESP*[1], and *HRESP*[0]. When it receives 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 receives a read request from a master, it first moves to the state *prep*, and then moves to the state *read* along with an *OKAY* response to indicate that the read operation can be done at the next cycle. Otherwise, it stays in the state *sleep* when there is no request or the request is for an *IDLE* or *BUSY* transfer.

How do we verify if the FSM in Fig. 4 complies with the protocol in Fig. 3? Note that this is not simply equivalence checking 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. We

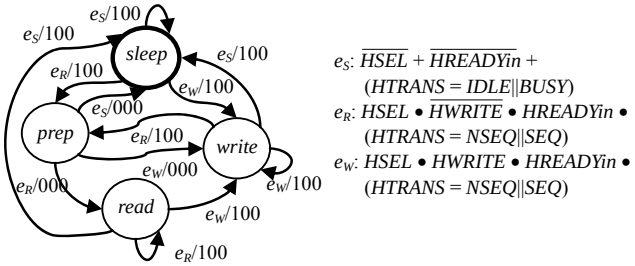


Fig. 4. The DUV FSM of an AHB slave interface design.

name these two states a corresponding state pair (or state pair in short). Definition 3 formally defines the corresponding state pair.

Definition 3 (Corresponding state pair) Given a spec FSM $M_s = (Q_s, \Sigma_s, \sigma_s, q_{s0}, q_{vio}, q_{dc})$ and a DUV FSM $M_d = (Q_d, \Sigma_d, \Delta_d, \sigma_d, q_{d0})$, (q_{s0}, q_{d0}) is a native corresponding state pair. Assume state $q_a \in Q_s$ and $q_b \in Q_d$ is a corresponding state pair. If there exist certain I/O values such that the spec FSM moves from q_a to $q_a' \in Q_s$ and the DUV FSM moves from q_b to $q_b' \in Q_d$, then (q_a', q_b') is also a corresponding state pair.

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 since 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 both indicate the status of data transfer.

Since this compliance verification is neither equivalence checking nor subset/superset checking, the problem must be solved by other methods. Property 1 shows that the DUV is compliant to the spec FSM if and only if all possible state pairs do not include the state q_{vio}. Hence the issue becomes how to explore all possible state pairs for the given FSM pair.

Property 1 Given a spec FSM $M_s = (Q_s, \Sigma_s, \sigma_s, q_{s0}, q_{vio}, q_{dc})$ and a DUV FSM $M_d = (Q_d, \Sigma_d, \Delta_d, \sigma_d, q_{d0})$, M_d is compliant to M_s if and only if all possible state pairs are examined and none of them includes the state q_{vio}.

Proof: ($\rightarrow$) Within a spec FSM, the state q_{vio} indicates a special status that the corresponding design behavior is illegal. If M_d is compliant to M_s, all states in M_d must not correspond to q_{vio} in M_s. Thus there is no state pair that contains q_{vio}.

($\leftarrow$) Now assume all possible state pairs of the two machines are exhaustively examined and none of them includes the state q_{vio}, it means that all states in M_d do not map to the state q_{vio}. That is, all state transitions in M_d are either legal or don't-care. Thus M_d is compliant to M_s.

3.2 Compliance Checking Algorithm

According to Property 1, the basic notion of our compliance checking algorithm for the given two FSMs is to find all possible state pairs, and verify if there is any one containing the state q_{vio}. In the beginning, there is a native state pair, consisting of the initial states from each of FSMs, respectively. This state pair is the root for further state pair exploration. Based on Definition 3, new state pairs can be found from the known state pairs repeatedly. Therefore, this process can completely collect all possible state pairs which are reachable from the initial state pair.

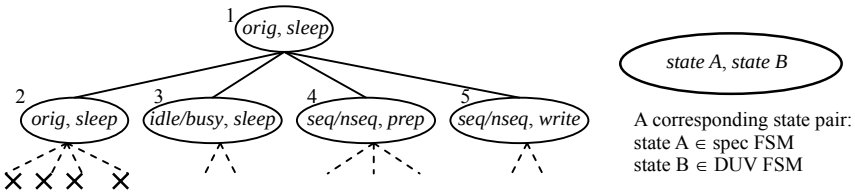


Fig. 5. The state pair exploration process.

Fig. 5 illustrates the state pair exploration process. *node*1 is the state pair of the initial states in Figs. 3 and 4. Consider the outgoing edge e_2 of *orig* in Fig. 3 and the outgoing edge e_W/100 of *sleep* in Fig. 4, the intersection of the Boolean functions of these two edges ($f_{orig,seq/nseq}$, $f_{sleep,write}$) is non-empty, which implies there exist certain input values along with the corresponding output values that can drive both transitions at the same time. For example, the set of input values "*HSEL* = *HWRITE* = *HREADYin* = 1, *HTRANS* = *NSEQ*", which drives e_w, along with the output 100, can drive e_2 as well. Hence, *node*5 (*seq/nseq*, *write*) is explored and represented as a child of *node*1. Similarly, all outgoing edges of *orig* versus all outgoing edges of *sleep* have to be examined in the same manner. Then we can get all children of *node*1 as shown in Fig. 5.

In this way, the exploration process recursively generates all descendants of the root node. Eventually all possible state pairs are present as nodes in this tree.

Property 2 Given two state pairs, P_A and P_B, if P_A and P_B are identical, then the two sets of state pairs, S_A and S_B, explored from P_A and P_B during the state pair exploration process are also identical.

Proof: Assume there is a I/O sequence which makes P_A move to another state pair P_C. Since P_A and P_B are identical, this sequence also makes P_B move to P_C. Thus the two sets of states explored from P_A and P_B are always the same.

In Fig. 5, since the subtree rooted at *node*1 is identical to that rooted at *node*2 according to Property 2, all possible state pairs explored from *node*2 can also be explored from *node*1. Hence, exploring the subtree rooted at *node*2 is completely unnecessary. Without losing any reachable state pairs, he exploration process can safely stop finding children for a node if this node has been already visited. That is, the exploration process becomes very effective by pruning the search tree without affecting the final result.

```
S = ϕ;
explore_tree(q_s0, q_d0);
explore_tree(q_s, q_d) {
   S = S ∪ (q_s, q_d);
   for i = 1 to |e_qs|
      for j = 1 to |e_qd|
      if (f_qs,qi • f_qd,qj ≠ 0 && (q_i, q_j) ∉ S)
         if (q_i ≠ q_dc && q_i ≠ q_vio)
            explore_tree(q_i, q_j);
         else if (q_i == q_vio)
            give a counterexample and exit;
}
```

Fig. 6. The compliance checking algorithm for FSMs.

In summary, our algorithm starts constructing a search tree from the initial state pair. It keeps exploring child nodes for each existing node unless the node has already appeared in the search tree. The pseudo code of the algorithm, *explore_tree*, is shown in Fig. 6.

3.3 Complexity Analysis

The time complexity of the proposed algorithm is estimated by the iteration count:

$$\text{iteration count} = \sum_{n=1}^{N} (|e_{q_s,n}| \times |e_{q_d,n}|). \tag{1}$$

In Eq. (1), $|e_{q_s,n}|$ and $|e_{q_d,n}|$ denote $|e_{q_s}|$ and $|e_{q_d}|$ at the nth recursion, and N denotes the recursion depth. For the worst case,

$$N = |Q_s| \times |Q_d|, \tag{2}$$

$$|e_{q_s,n}| = |Q_s|, \tag{3}$$

$$|e_{q_d,n}| = |Q_d|, \tag{4}$$

$$\text{Maximum iteration count} = (|Q_s| \times |Q_d|)^2. \tag{5}$$

$$\text{Worst-case time complexity} = O((|Q_s| \times |Q_d|)^2). \tag{6}$$

Eq. (2) holds only when all combinations of states in two FSMs are state pairs. Eqs. (3) and (4) hold only when the two FSMs are both complete graphs. However, these worst-case conditions rarely occur. Actually, experimental results show that the iteration count is typically far lower than this theoretical upper bound. Eq. (6) shows that the complexity of the proposed method is polynomial to the product of the states sizes of spec FSM and DUV FSM. To our knowledge, the state numbers of spec FSM and DUV FSM are under an acceptable scale in interface compliance verification. That is, the proposed method can work well in this specific application.

4. EXTENSION

To make the compliance checking method more flexible, we extend the proposed algorithm with the EFSM-based modeling style. In this section, we introduce the spec EFSM model and describe how to extend the original algorithm with it.

4.1 Spec EFSM

The EFSM model is an FSM extended with internal variables. It gives a more efficient way to describe the behavior of a sequential circuit and relaxes the state explosion problem suffered by traditional finite state machine models. The EFSM model has been widely used in many research works [21-23].

Definition 4 (EFSM) An extended finite state machine is a 7-tuple $M = (Q, \Sigma, \Delta, x, T, q_0, x_0)$ 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
q_0: $q_0 \in Q$, the initial state
x_0: the initial values of variables in x
T: the set of transitions, each transition 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:

$P_{q_i,q_j}(x)$: the predicate of the transition from the state q_i to q_j
$A_{q_i,q_j}(x)$: the action of the transition from the state q_i to q_j

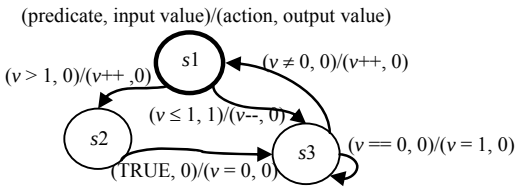


Fig. 7. An EFSM example.

A transition $t = (s_t, q_t, i_t, o_t, P_t, A_t)$ means if the input values are in i_t and the predicate $P_t(x)$ is evaluated true, then the EFSM outputs o_t, performs the action $A_t(x)$ and moves from the current state s_t to the next state q_t. Fig. 7 illustrates an EFSM example. The initial state $s1$ is the one in bold circle. 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 moving to the next state $s2$.

To best fit our approach, we further define the spec EFSM as follows.

Definition 5 (spec EFSM) A spec EFSM is an 8-tuple $M = (Q, \Sigma, x, T, q_0, x_0, q_{vio}, q_{dc})$ where
Q: the set of symbols denoting states
Σ: the set of symbols denoting inputs
x: the set of symbols denoting variables
q_0: $q_0 \in Q$, the initial state
x_0: the initial values of variables in x
q_{vio}: $q_{vio} \in Q$, the illegal state
q_{dc}: $q_{dc} \in Q$, the don't-care state
T: the set of transitions, each transition t is a quintuple $t = (s_t, q_t, i_t, P_t, A_t)$ where
s_t, q_t, i_t: current state, next state, set of input values
$P_t(x)$, $A_t(x)$: the predicate and the action on current variables

4.2 Extension with Spec EFSM

Timing constraints are common in interface protocols. However, it is sometimes tedious to specify them with a spec FSM. 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 Fig. 8 (a). It requires a large number of states to represent such a simple protocol. Instead, if we specify this protocol with an EFSM by introducing a variable *count* as in Fig. 8 (b), the representation becomes much clearer and easier. The building process of the spec EFSM is similar to that of the spec FSM described in section 2.3.

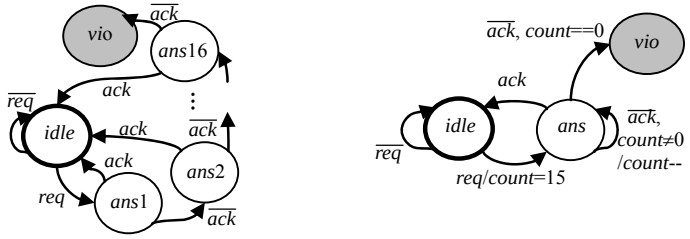


Fig. 8. Specify the same protocol in (a) FSM and (b) EFSM.

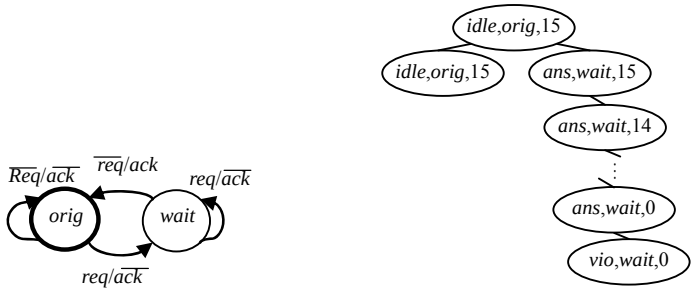


Fig. 9. (a) A wrong implementation of the protocol in Fig. 8; (b) The tree explored by the extended algorithm.

Meanwhile, we extend the compliance checking algorithm in section 3 for handling the spec EFSM. Similarly, the extended algorithm explores the subtree rooted at a node unless that the tree has already contained the node. The only difference is that the node is associated with not only a state pair but also the values of the corresponding variables. A simple example is given in Fig. 9. Fig. 9 (a) gives an erroneous implementation of the interface protocol in Fig. 8. This design violates the protocol because it may not assert *ack* within 16 cycles if it sticks in the state *wait*. Fig. 9 (b) shows one possible scenario for the exploration process to find the violation. The three columns in each node from left to right denote the state of spec FSM, the state of DUV FSM, and the value of the variable *count*. The nodes in the right branch demonstrate 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. That is, the extended algorithm prunes the subtree rooted at a node only when there is already an existing node associated with the identical state pair and variable values.

The extended algorithm *extended_explore_tree* is given in Fig. 10. The differences between *explore_tree* and *extended_explore_tree* are highlighted (shaded). The worst-case time complexity of *extended_explore_tree* is:

$$\text{Maximum iteration count} = |range(A(x))| \times (|Q_s| \times |Q_d|)^2. \quad (7)$$

The term $range(A(x))$ is the value range of the function $A(x)$ and thus $|range(A(x))|$ is the number of possible values of x. By comparing Eq. (7) against Eq. (6), although the two equations look different, the complexity represented by both equations are actually the same. That is, the EFSM simply provides a more elegant way to represent a specification. Fig. 8 exactly demonstrates the case.

Given a spec *EFSM* $M_s = (Q_s, \Sigma_s, x, T, q_{s0}, x_0, q_{vio}, q_{dc})$ where $\Sigma_s = I_{ctrl} \cup O_{ctrl}$
and a *DUV FSM* $M_d = (Q_d, \Sigma_d, \Delta_d, \sigma_d, q_{d0})$ 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_explore_tree(q_s0, q_d0, x_0);
extended_explore_tree(q_s, q_d, x) {
   S = S ∪ (q_s, q_d, x);
   for i = 1 to |e_qs|
      for j = 1 to |e_qd| {
         x′ = A_qs,qi(x);
         if (f_qs,qi • f_qd,qj ≠ 0 && P_qs,qi(x) == ture && (q_i, q_j, x′) ∉ S)
            if (q_i ≠ q_dc && q_i ≠ q_vio)
               extended_explore_tree(q_i, q_j, x′);
         else if (q_i == q_vio)
               give a counterexample and exit;
}
```

Fig. 10. The compliance checking algorithm for spec EFSM and DUV FSM.

5. EXPERIMENTAL RESULTS

We implement the proposed algorithm in C language. The implementation can either formally check that the given design is fully compliant to a certain interface protocol or report an input sequence as a counterexample to show how the given design fails in the compliance verification. The experiments are conducted over a set of real AMBA AHB-compliant and WISHBONE-compliant designs [24]. The DUV FSMs are represented with BLIF-KISS format [25] while the spec (E)FSMs are written in enhanced BLIF, which is capable of describing predicates and actions required for the EFSM model. We build a spec FSM for WISHBONE protocol and a spec EFSM for AMBA AHB protocol to completely represent the essential I/F specifications, respectively. To check whether the proposed algorithm can find the design flaws as expected, we intentionally inject errors into the design *con*7 and *mac* as two additional benchmark designs *con*7*_err* and *mac_err*.

Table 1 shows the experimental results. As shown, the DUVs utilize different functional modes of the protocols, but the same (E)FSM can be used to verify all designs of the same protocol without altering. Furthermore, the compliance verification algorithm can successfully detect the injected errors. The error in the design *con*7*_err* is induced by a self-loop of the state performing the *WAIT* operation. 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 do not deal with the *WAIT* response carefully. The other error in the design *mac_err* is about the two-cycle response behavior. An *IDLE*

Table 1. The DUVs and the verification results.

I/F protocol type	DUV	Result	Utilized functional modes	Violation cause
WISHBONE slave (spec FSM)	*spi* [26]	Pass	Normal & Error response	–
	*ac*97 *ctrl* [26]	Pass	Normal response	–
AMBA AHB slave (spec EFSM)	*con*7	Pass	OKAY & WAIT response	–
	mac	Pass	OKAY & ERROR response	–
	remap	Pass	OKAY, ERROR, & RETRY response	–
	*con*7*_err*	Fail	OKAY & WAIT response	Wait > 16 cycles
	mac_err	Fail	OKAY & ERROR response	Erroneous 2-cycle response

Table 2. Complexity comparison.

I/F protocol type	DUV	Result	Number of different nodes	Theoretical bound $\lvert range(A(x))\rvert \times (\lvert Q_s\rvert \times \lvert Q_d\rvert)^2$
WISHBONE slave (spec FSM)	*spi*	Pass	180	$(7 \times 3)^2 = 442$
	*ac*97 *ctrl*	Pass	221	$(7 \times 5)^2 = 1225$
AMBA AHB slave (spec EFSM)	*con*7	Pass	204	$16 \times (7 \times 4)^2 = 12544$
	mac	Pass	191	$16 \times (7 \times 6)^2 = 28224$
	remap	Pass	136	$16 \times (7 \times 6)^2 = 28224$
	*con*7*_err*	Fail	42	$16 \times (7 \times 4)^2 = 12544$
	mac_err	Fail	57	$16 \times (7 \times 6)^2 = 28224$

transfer must be initiated after an *ERROR* response, the design does not respond *OKAY* but respond *ERROR* instead, which is a violation of the AHB protocol. With our verification approach, these errors and the reasons leading to them are clearly identified.

Table 2 displays the complexity comparison. The results indicate that the actual number of nodes is far less than that in the worst case analysis. As a matter of fact, each verification run listed in Table 1 finishes within few seconds on a 300MHz UltraSPARC II workstation with 256MB RAM. It shows that our algorithm is capable of completing the formal compliance verification of interface protocol in reasonable time.

6. CONCLUSIONS

In this paper, we introduce the spec FSM to systematically represent an interface protocol specification. We further show how to formulate the interface compliance verification as the compliance checking between the spec FSM and DUV FSM. A state pair exploration 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 effectively modeling more sophisticated properties. Experimental results demonstrate that our approach can effectively and efficiently verify the interface compliance over a set of real designs.

In comparison with simulation-based methods, our method is formal thus does not have the false positive problem. In comparison with other 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 design environment.

REFERENCES

1. K. Keutzer, S. Malik, A. R. Newton, J. M. Rabaey, and A. Sangiovanni-Vincentelli, "System level design: Orthogonalization of concerns and platform-based design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 19, 2000, pp. 1523-1543.
2. VSI Alliance, Virtual Component Interface (VCI) Standard – OCB 2 1.0, http://www.vsia.org, 2000.
3. K. Shimuzu, D. L. Dill, and A. J. Hu, "Monitor-based formal specification of PCI," in *Proceedings of the 3rd International Conference on Formal Methods in Computer-Aided Design*, 2000, pp. 335-353.
4. M. T. Oliviera and A. J. Hu, "High-level specification and automatic generation of IP interface monitors," in *Proceedings of the 39th Design Automation Conference*, 2002, pp. 129-134.
5. A. J. Hu, J. Casus, and J. Yang, "Efficient generation of monitor circuits for GSTE assertion graphs," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, 2003, pp. 154-159.
6. J. Yuan, K. Shultz, C. Pixley, H. Miller, and A. Aziz, "Modeling design constraints and biasing in simulation using BDDs," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, 1999, pp. 584-589.

7. K. Shimizu and D. L. Dill, "Deriving a simulation input generator and a coverage metric from a formal specification," in *Proceedings of the 39th Design Automation Conference*, 2002, pp. 801-806.
8. A. Nightingale and J. Goodenough, "Testing for AMBA™ compliance," in *Proceedings of the 14th Annual IEEE International ASIC/SOC Conference*, 2001, pp. 301-305.
9. ARM Limited, *AMBA Specification* (Rev 2.0), 1999.
10. H. M. Lin, C. C. Yen, C. H. Shih, and J. Y. Jou, "On compliance test of on-chip bus for SOC," in *Proceedings of the Asia and South Pacific Design Automation Conference*, 2004, pp. 328-333.
11. K. L. McMillan, *Symbolic Model Checking*, Kluwer Academic Publishers, Norwell, MA, 1993.
12. P. Chauhan, E. M. Clarke, Y. Lu, and D. Wang, "Verifying IP-Core based system-on-chip designs," in *Proceedings of the 12th Annual IEEE International ASIC/SOC Conference*, 1999, pp. 27-31.
13. I. Beer, S. Ben-David, C. Eisner, Y. Engel, R. Gewitzman, and A. Landver, "Establishing PCI compliance using formal verification: a case study," in *Proceedings of the 14th International Phoenix Conference on Computation and Communications*, 1995, pp. 373-377.
14. A. Roychoudhury, T. 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. http://www.eda.org/vfv/docs/psl_lrm-1.01.pdf/.
16. http://www.verificationlib.org/.
17. http://www.opervera.org/.
18. http://www.systemverilog.org/.
19. A. Bunker and G. 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*, 2001, pp. 95-100.
20. A. Bunker, G. Gopalakrishnan, and S. A. McKee, "Formal hardware specification languages for protocol compliance verification," *ACM Transactions on Design Automation of Electronic Systems*, Vol. 9, 2004, pp. 1-32.
21. K. T. Cheng and A. S. Krishnakumar, "Automatic functional test generation using the extended finite state machine model," in *Proceedings of the 30th Design Automation Conference*, 1993, pp. 86-91.
22. D. Lee and M. Yannakakis, "Optimization problems from feature testing of communication protocols," in *Proceedings of the 4th International Conference on Network Protocols* , 1996, pp. 66-75.
23. C. Besse and A. 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*, 1999, pp. 73-80.
24. OpenCores Organization, *Specification for the: WISHBONE System-on-Chip* (*SoC*) *Interconnection Architecture for Portable IP Cores*, Rev. B.3, 2002.
25. University of California Berkeley, *Berkeley Logic Interchange Format* (*BLIF*), Sept. 1996.
26. http://www.opencores.org/.

Che-Hua Shih (石哲華) received the B.S. degree in Electrical Engineering from National Tsing Hua University and M.S. degree in Electrical Engineering from National Chiao Tung University. He is currently working toward the Ph.D. degree at National Chiao Tung University. His research interests include design verification and electronics design automation.

Ya-Ching Yang (楊雅菁) has a B.S. and an M.S. degree in Department of Electronics Engineering at National Chiao Tung University, Hsinchu, Taiwan. Her research interests include formal verification and IC design flow.

Chia-Chih Yen (顏嘉志) is currently a principal R&D engineer at SpringSoft Inc. His research interests include IC system design, verification, testing, power optimization, and design closure techniques. Yen has a B.S. in Electrical Engineering from National Taiwan University and an M.S. and Ph.D. in the Department of Electronics Engineering at the National Chiao Tung University, Hsinchu, Taiwan.

Juinn-Dar Huang (黃俊達) received the B.S. and Ph.D. degrees in Electronics Engineering from National Chiao Tung University, Hsinchu, Taiwan, in 1992 and 1998, respectively. He is currently an Assistant Professor in the Department of Electronics Engineering and the Institute of Electronics, National Chiao Tung University. His current research interests include high-level synthesis, design verification, 3D IC architecture/CAD, and microprocessor design. Dr. Huang is currently a Guest Editor of the International Journal of Electrical Engineering (IJEE). He is also serving on the Organizing Committee of IEEE/ACM ASP-DAC

2010 and the Technical Program Committee of IEEE VLSI-DAT 2010. He has been the Secretary General of Taiwan IC Design Society (TICD) from 2004 to 2008, the Technical Program Committee Vice-Chair of VLSI Design/CAD Symposium 2008, the Technical Program Committee member of IEEE/ACM DATE 2008, and the Organizing Committee member of IEEE ICFPT 2008. He is a member of the IEEE, ACM, IEICE, and Phi Tau Phi.

Jing-Yang Jou (周景揚) received the B.S. degree in Electrical Engineering from National Taiwan University, Taiwan, R.O.C., and the M.S. and Ph.D. degrees in Computer science from the University of Illinois at Urbana-Champaign, in 1979, 1983, and 1985, respectively. He is currently the Vice Chancellor, University System of Taiwan. He is the Executive Director of National SoC Program from April 2007. He was the Director General of National Chip Implementation Center, National Applied Research Laboratories in Taiwan from February 2004 to June 2007. He is a full Professor and was Chairman of Electronics Engineering Department from 2000 to 2003 at National Chiao Tung University, Hsinchu, Taiwan. Before joining Chiao Tung University, he was with GTE Laboratories from 1985 to 1986 and with AT&T Bell Laboratories at Murray Hill from 1986 to 1994.

He received the distinguished paper award of the IEEE International Conference on Computer-Aided Design in 1990, the Outstanding Academy-Industry Cooperation Achievement Award granted by Ministry of Education (MOE), Taiwan, in 2002, and the Outstanding Electrical Engineering Professor Award from CIEE in 2006. His research interests include logic and physical synthesis, design verification, CAD for low power and Network on Chips. He has published more than 160 technical papers. Dr. Jou is a Fellow of IEEE.

He was elected to the President of the Taiwan Integrated Circuit Design Society (TICD) 2007-2008. He serves as Associate Editor for IEEE Transactions on Very Large Scale Integration Systems. He was the Technical Program Chairs of 2007 VLSI-DAT, the 12th VLSI Design/CAD Symposium (2001), and the Asia-Pacific Conference on Hardware Description Languages (APCHDL'97). He was the Conference Chair of 2008 VLSI-DAT.