國 立 交 通 大 學

電子工程學系電子研究所

博 士 論 文

利用延伸式有限狀態機來實現介面規格相符驗證之研究

Interface Compliance Verification Using the EFSM Model

研 究 生 ：石哲華

指導教授 ：周景揚 博士

中華民國九十八年九月

利用延伸式有限狀態機來實現介面規格相符驗證之研究

學生：石哲華　　　　　指導教授：周景揚

國立交通大學

電機學院　電子工程學系　電子研究所

摘要

進入了系統單晶片(SOC)時代後，整合大量矽智產(intellectual property)於單一晶片上，被視為設計複雜系統及加速設計流程的有效方案。這些矽智產往往來自不同的設計團隊或公司，為了提高矽智產的再使用性與減少整合時所需的時間，矽智產通常會針對特定的介面協定(interface protocol)設計，而擁有相容介面協定的矽智產群便可以很容易地在彼此間傳遞資料。今日的介面協定為了提供更高速、更具有彈性的使用，其規格(specification)也愈益複雜，因此，驗證一個矽

智產設計是否吻合其傳輸介面，能夠在整合後正確地溝通資料，便成為現今系統單晶片驗證上的一大課題。

模擬驗證(simulation-based Verification)是目前最廣泛應用在介面相容性驗證(interface compliance verification)的方法，模擬驗證主要是利用模擬器(simulator)來模仿晶片的實際運作，具有較低的進入門檻及可處理較大電路為其優勢。在模擬驗證中，驗證人員透過適當的外部輸入向量(input stimuli)來驅動待驗證設計的內部行為，同時，觀察模擬時的訊號變化來檢查是否有違反設計規格的情形，在一段時間的模擬之後，涵蓋率量度(coverage metric)則經常被採用來量化當前的驗證品質。傳統上運用人力來撰寫輸入向量、比對模擬結果的作法，不僅費時耗力也容易出錯，如果能夠利用工具自動化地完成上述工作，則可有效地加速模擬驗證的流程。

一般的介面規格大多是使用自然語言(例如英文)或時序示意圖(timing diagram)這類非正規的方法來描述，在驗證自動化的過程中，首要任務便是如何將這些介面規格轉譯為定義明確的形式(利用正規的語言或表示法)。然而許多正規驗證語言的學習及使用上的難度較高，容易導致轉譯過程中的錯誤，進而影響驗證的正確性。在這篇論文中，我們利用了有限狀態機模型的優點，同時考量介面規格常見的特性，發展了兩種基於延伸式有限狀態機模型(Extended Finite State Machine)的介面規格描述方式，用來系統化地解譯介面規格。

對於介面相容性驗證的問題，我們提出了一套完整的自動化流程。透過所提出之演算法，我們可以從單一延伸式有限狀態機模型自動化地產生模擬驗證時所需之各種主要元件：包含了向量產生器(stimulus generator)、協定檢查器(protocol checker)及涵蓋率分析器(coverage analyzer)。這些元件由於來自同一個模型，可以避免元件間的不一致性，同時，我們的方法也提供了許多便於驗證與除錯的特性，可用來大幅提升驗證的效率，實驗的結果顯示了，我們的方法的確可以有效地執行介面規格相容性驗證並縮短所需的時間。

Interface Compliance Verification Using the EFSM Model

Student: Che-Hua Shih **Advisor: Dr. Jing-Yang Jou**

Department of Electronics Engineering

Institute of Electronics

National Chiao Tung University

Abstract

In designing a modern system-on-a-chip (SOC), the platform-based design methodology with reusable intellectual property (IP) cores is usually adopted to accelerate both the design and verification process. In this kind of methodology, an IP core is often wrapped with certain interface logic and integrated into a system platform based on a specific interface protocol. To ensure that an IP core can concordantly communicate with others within the system, it is very important to guarantee that its interface logic exactly conforms to the protocol for communication. Hence, interface compliance verification becomes an essential part in the SOC verification flow.

The simulation-based approaches are widely-used in interface compliance verification works. Simulation-based verification has a lower barrier to entry and can handle large designs. In this kind of approaches, appropriate input stimuli are applied to trigger the design's internal operations. Simultaneously, the signal changes are observed to check if there is any violations against the specification. Besides, certain coverage metrics are usually adopted to quantify the verification completeness. Typically, those tasks are made manually which are time-consuming and error-prone. To achieve high verification efficiency, automation is necessary to speed up the verification process.

In general, interface protocol specifications are written with natural languages or timing diagrams. To enable the automatic verification process, the first task is to translate the original specification into a well-defined representation. In this dissertation, we developed two kinds of extended finite state machine (EFSM) models which are suitable for representing common interface protocols. Besides, we propose a unified framework using the EFSM model for interface compliance verification. Via the proposed algorithms, the

EFSM model can be automatically translated into a simulation kit consisting of three verification components, a stimulus generator, a protocol checker, and a coverage analyzer. The simulation kit has many useful features to increase the verification efficiency. Our experimental results demonstrate that the proposed framework improves not only the performance but also the quality for interface compliance verification.

Acknowledgements

This dissertation would not have been possible to complete without the assistance and support of numerous individuals. First, I would like to express my sincere appreciation to my advisor, Professor Jing-Yang Jou (周景揚) for his support, suggestions and guidance throughout my graduate life. And I would also like to thank Professor Juinn-Dar Huang (黃俊達), who had given me a lots of valuable suggestions and discussions.

I would like to thank the members of my dissertation committee, Professor Sy-Yen Kuo (郭斯彥), Professor Sao-Jie Chen (陳少傑), Professor Kuen-Jong Lee (李昆忠), Professor Shi-Yu Huang (黃錫瑜), Professor Chau-Chin Su (蘇朝琴), and Professor Chun-Yao Wang (王俊堯), for their comments and suggestions .

Besides, special thanks to Geeng-Wei Lee (李耿維), Cheng-Yeh Wang (王成業), Tai-Ying Jiang (江泰盈), Hen-Ming Lin (林恆民), Chia-Chih Yen (顏嘉志), Chien-Hua Chen (陳建華), Bu-Ching Lin (林步青), and all other EDA members for the wonderful time we share together.

My deepest appreciation goes to my wife, Huei-Min Lin (林惠敏), who fills my life with laughter and love. Thank for her patient care on my life and her endless support. Finally, I would like to devote this dissertation and its honor to my parents and family. Thanks for their love and encouragement.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Interface Compliance Verification

In designing a modern system-on-a-chip (SOC), the platform-based design methodology with reusable intellectual property (IP) cores is usually adopted to accelerate both the design and verification process. Fig. 1.1 illustrates the concept of this methodology. Each pre-verified IP core is wrapped with certain interface logic and integrated into a system platform based on a specific interface protocol. In order to ensure that an IP core can concordantly communicate with others within the system, it is very important to guarantee that its interface logic exactly conforms to the protocol for communication. Hence, interface compliance verification becomes an essential part in the SOC verification flow.

There are two major categories in the field of interface compliance verification: formal methods and simulation-based ones. Fig. 1.2 illustrates the basic concept of these two

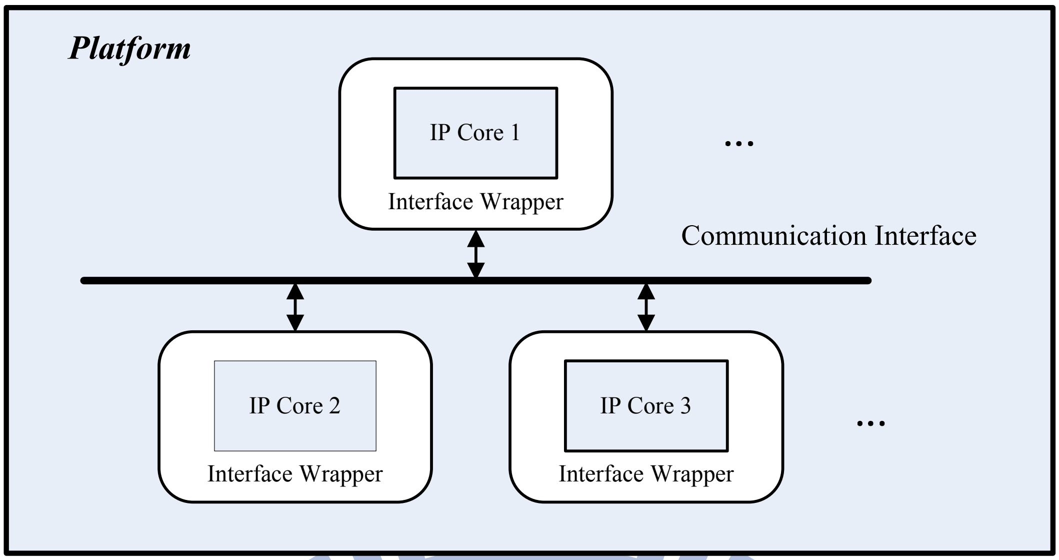


Figure 1.1: The platform-based design methodology.

methods. Formal methods mathematically determines whether the design is correct or not. In [1][2][3][4][5][6], the authors use CTL (Computation Tree Logic) [1] to describe the expected properties and the *DUV (design under verification)* is modeled as a finite state machine. Then the model checking [7] technique is applied to verify 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.

The simulation-based approaches are classical but still widely-used in today's verifi-

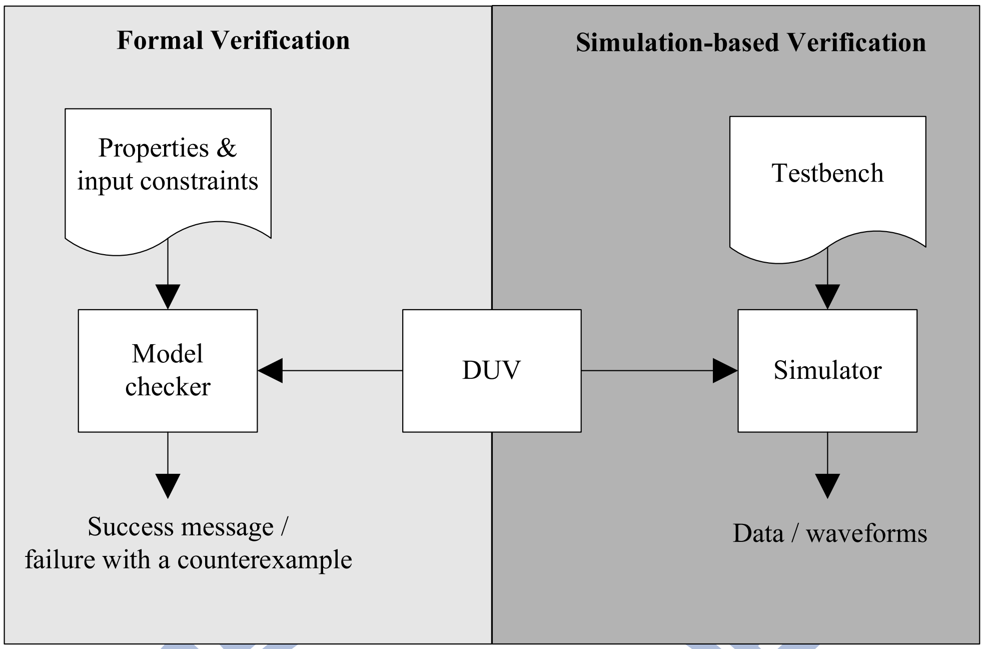

Formal Verification
Simulation-based Verification
Properties & input constraints
Testbench
Model checker
DUV
Simulator
Success message / failure with a counterexample
Data / waveforms

Figure 1.2: Typical interface compliance verification flow.

cation works. In simulation-based approaches, a simulator is adopted to create an artificial universe that imitates the behaviors of a real design. In [8], they write the protocol specification with HDL monitors which can be directly used in the HDL simulation environment to check if the simulation trace violates the target protocol. In [9], the authors propose a systematic flow to represent the interface behaviors with a monitor FSM. They also defines a path-based coverage metric to check the exercised functionalities. A regular-expression-based specification style with a monitor circuit generation algorithm is proposed in [10]. Another similar work in [11] generates monitor circuit from GSTE assertion graphs. Simulation-based approaches have better scalability against formal ones. However, the simulation-based approaches suffer from long simulation time and the false positive problem. Hence, how to perform simulation-based verification efficiently and effectively is very import.

A typical simulation environment for functional verification is shown in Fig. 1.3. The DUV is the verification target. The two directed edges labelled with I_{DUV} and O_{DUV} denote the *input and output (I/O)* signal sets of the DUV, respectively. Three tasks, the stimulus generation, the correctness checking, and the coverage analysis, are included in the environment. During simulation, the stimulus generation task provides necessary I_{DUV} values to trigger the DUV's operations, and the correctness checking task examines the simulation data for error detection. After simulation, the coverage analysis is performed to measure the verification completeness. Each task plays an important role in simulation-based verification, and many techniques have been proposed to fulfill these tasks in the last decades.

1.2 Stimulus Generation

To exercise different operations of a DUV, various input stimuli are required. On the one hand, since the inputs to the DUV must conform to the specification, randomly generated stimuli without any constraints are usually invalid which must be identified, discarded and thus slow down the verification process. Generally, input stimuli are often made manually by the designers who are familiar with the specification. In common interface protocol specifications, stimuli applied to a DUV need to dynamically interact with the DUV's responses, thus the stimulus generation must be made with extreme cares. On

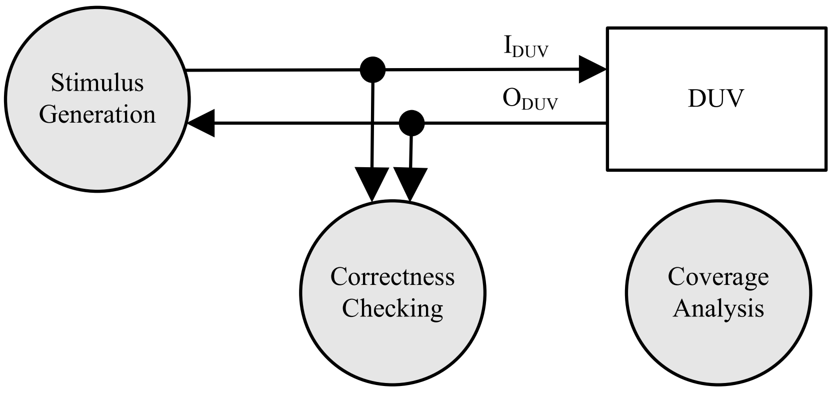


Figure 1.3: A typical environment for simulation-based verification.

the other hand, in order to expose possible design errors of the DUV, a large amount of stimuli are required. However, the manual stimulus development tends to be error-prone and time-consuming. As a result, many research works [12] [13] [14] [15] [16] [17] [18] [19] [20] [21] prefer to build a constraint-based stimulus generator to automate this process. Constraints can be regarded as formal specifications of design behaviors. A constraint-based stimulus generator produces only valid stimuli based on the given constraints. Meanwhile, it is generally not uncommon that multiple valid stimuli exist for a given set of constraints. Some generators pick the first valid one they find while others choose one of the valid stimuli randomly. In general, verification engineers prefer owning more controllability over simulation, thus biasing techniques are often along with the stimulus generation. Biasing allows user-defined settings to increase or decrease the appearance counts of certain stimuli. While constraints limit the random stimuli to valid space, the bias settings can further guide the stimuli to hit the interested design corners.

Previous researchers typically use satisfiability (SAT) solvers [22] [23] or binary decision diagram (BDD) [24] solvers as their constraint-solving engines. The SAT method is a formal technique that is generally capable of solving a large number of complex constraints. However, a typical SAT solver often generates only one feasible solution under a given constraint set with no biasing capability. Alternatively, solving constraints with BDDs is relatively easier to be combined with the bit-level biasing approach. However, the weakness of the BDD-based solver is the memory explosion problem while handling a large set of complex constraints.

Yuan et al. [13] propose a general-purpose stimulus generation system, called *Sim-Gen*. Constraints are represented as Boolean formulas with state variables. Then they conjoin related constraints into a single BDD before simulation. Next, BDDs are traversed in a top-down fashion to produce stimuli. Since the traversal needs only one pass without any backtracking, this engine has good performance in solving constraints. Furthermore, it has the advantage of its bit-level biasing ability, that is, users can define the desired stimulus biasing at bit-level to adjust the branch probabilities of BDD nodes during stimulus generation. Another work by Shimizu et al. [14] targets on interface verification. First, authors write a list of interface constraints in a proprietary specification style. Next, they create BDDs with appropriate constraints on-the-fly instead of before simulation. In this way, BDDs can be smaller and thus solved more quickly. However, this approach needs to rebuild a new set of BDDs at every simulation cycle.

In [18], the author presents a word-level constraint-solving engine, called *RACE*, to simplify the stimulus generation for multi-bit signals. *RACE* can handle constraints described with multi-bit operands and most operators in high-level verification languages. The constraint-solving mechanism of *RACE* is similar to *PODEM* [25], which is a popular gate-level ATPG (*Automatic Test Pattern Generation*) algorithm. *RACE* makes a series of value assignments to find a satisfied stimulus. While a conflict occurs during these assignments, the backtraking procedure is invoked to find another set of value assignments. Due to the nature of the word-level constraints and the less memory demand, *RACE* is capable of solving a large number of complex constraints. Nevertheless, while the solution space for a set of target constraints is small, the backtracking procedure would be taken frequently and thus slows down the stimulus generation.

1.3 Correctness Checking

Correctness checking verifies if the simulation results violate what the specification expects. A waveform viewer is commonly-used for correctness checking. It visualizes multiple signal transitions over time to help designers observe the simulation data. In fact, examining a large amount of complex simulation data by hand requires tremendous effort and time. In order to speed up the verification process, automatic correctness checkers/monitors are often adopted to replace manual waveform checking or data comparing. In general, there are three common types of checker implementations. The first type is to build a table to record the relationships between the input stimuli and output responses of the DUV. And then check the table to see if the correct I/O relationships are preserved during simulation. The second type is to use a golden reference model whose I/O behaviors are regarded as an accurate design. When the identical input stimuli are simultaneously applied into both the reference model and the DUV, violations can be detected if their responses are different. The third type is the assertion-based approach [26]. In this approach, the specification is presented as rules (assertions) that must be satisfied throughout the simulation.

1.4 Coverage Analysis

Coverage metrics are usually adopted to quantitatively analyze the simulation completeness. They can not only measure how well a design is verified objectively but also help improve the quality of verification stimuli. They are capable of guiding further stimuli to explore those unverified design corners. In general, there are two major categories of coverage metrics [27]: code coverage and functional coverage.

Code coverage metrics concentrate on identifying which part of the implementation code has been executed in the DUV. That is, they measure how much of the implementation has been exercised [28] [29] [30] [31] [32]. For example, statement coverage, branch coverage, and condition coverage are well-known code coverage metrics. Many simulators have already provided simple code coverage analysis. However, the fundamental issue of all code coverage metrics is that they can only measure how well the structural code has been exercised. They are not designed to check whether all the functions expected in the design specification have been examined. Namely, the verification quality is generally considered not good enough for modern complex SOC designs even if a high code coverage has been achieved.

Hence, the functional coverage is usually applied to further boost the verification quality. Functional coverage, as its name suggests, focuses on the design functionality. It measures how much of the original design specification has been verified. Therefore, functional coverage is an appropriate measurement to check if the design aim, "to properly implement all the functions in the specification", is achieved.

1.5 Automatic Simulation Kit Generation

To speed up the verification process, manual works on stimulus generation, correctness checking, and coverage analysis can be replaced with a simulation kit, including a stimulus generator, a correctness checker, and a coverage analyzer. Typically, it takes significant effort and time to implement these simulation components according to the original specification. As mentioned, many research works aim at developing certain simulation components efficiently. However, for a complete verification environment, the required simulation components may be created by different methodologies and/or different verification teams. Whenever the specification is modified, the stimulus generator, the correctness checker, and the coverage analyzer should be individually revised according to the specification changes. This process often introduces consistency and integration problems to the whole environment.

To overcome these problems, we develop a unified framework for automatic simulation kit generation. We use extended finite state machine (EFSM) model [33] as the specification style for properly describing general interface behaviors. The complete simulation kit can then be automatically built from a given specification. Thus verification engineers just need to focus on maintaining the correctness of specification and the validity of every derived component is then guaranteed instantly. Hence, this kind of methodology can provide a more robust and efficient verification framework.

1.6 Dissertation Organization

The rest of this dissertation is organized as follows. Chapter 2 introduces the overall flow of the proposed verification methodology. Unified frameworks using a generator-based EFSM and checker-based EFSM are described in Chapter 3 and Chapter 4, respectively. Finally, we give the concluding remarks and future works in Chapter 5.

Chapter 2

An Overview of Our Methodology

In this chapter, we will first discuss two different viewpoints for specification interpreting. Then, we will give an overview of our methodology and compared it with previous works.

2.1 Different Viewpoints for Interpreting a Specification

Most interface protocols are written in natural languages or waveform diagrams. Like other automation methodologies, we need to formally specify the interface protocol first. Interpreting an interface specification is like to specify what is valid I/O traces. As shown in Fig 2.1, according to the target simulation components, engineers are used to interpret a specification in either generator's or checker's point of view. From a checker's viewpoint, it observes the behaviors of I_{DUV} and O_{DUV} simultaneously to determine whether the current trace is valid or not. From a generator's viewpoint, it produces valid input stimuli

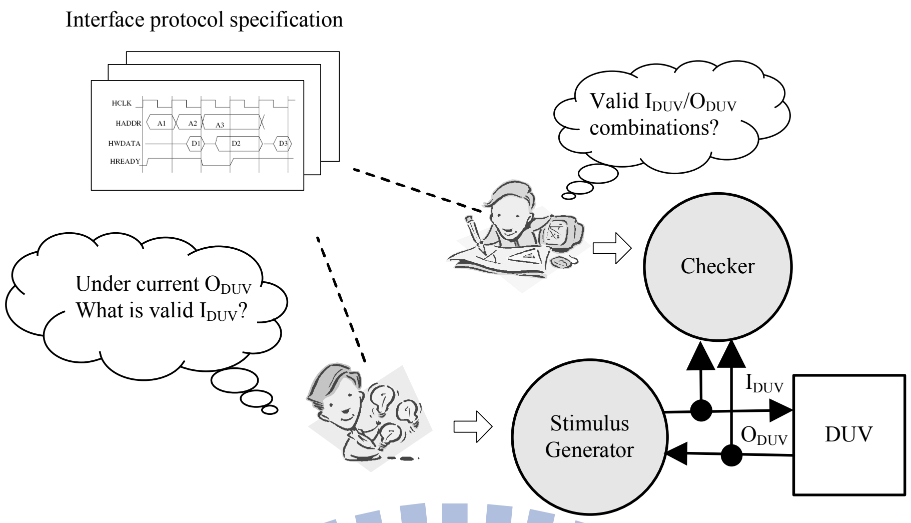

Interface protocol specification
HCLK
HADDR
HWDATA
HREADY
A1
A2
A3
D1
D2
D3
Valid I_{DUV}/O_{DUV} combinations?
Under current O_{DUV} What is valid I_{DUV}?
Checker
Stimulus Generator
I_{DUV}
O_{DUV}
DUV

Figure 2.1: Modeling in different points of view.

based on the output responses of the design. Actually, the generator and checker share an underlying semantics which defines a set of valid input / output traces. As a result, we develop two alternative EFSM models, the generator-based EFSM (GEFSM) and the checker-based EFSM (CEFSM), as the specification styles in our methodology.

One of the major differences between the two proposed models is that the CEFSM is a deterministic machine and the other is a non-deterministic machine. In a *deterministic* EFSM, for a given combination of input, output, variable, and state values, there is at most one transition can be enabled. On the contrary, a non-deterministic EFSM allows multiple possible transitions. On the one hand, the GEFSM mainly considers the output responses of the DUV and the internal state to define the valid stimuli to the DUV. Since the multiple valid stimuli are usually allowed under certain output responses, the state transitions of the GEFSM are non-deterministic which is more intuitive and flexible to write a generator. On the other hand, the structure of the CEFSM imitates a real checker which considers

Table 2.1: Comparisons among two EFSM models

Model	**Input**	**Output**	**State transition**	**Independent checker?**
GEFSM	O_{DUV}	I_{DUV}	$Non-deterministic$	No
CEFSM	$I_{DUV} \cup O_{DUV}$	$None$	$Deterministic$	Yes

all the input and output signals as the state transition conditions. Table 2.1 shows the basic comparison between them. Each of the two models can be translated to a complete simulation kit. However, there is a slight difference between the two translated kits. The CEFSM-based correctness checker can operate correctly with other stimulus generators / testbenches. But the GEFSM-based correctness checking ability can only accompany the GEFSM-based stimulus generator.

2.2 The Flow of Our Methodology

The flow of our methodology is illustrated in Fig. 2.2. The interface protocol specification is expressed with a modified EFSM model initially. The model can be translated into a stimulus generator, a correctness checker, and a structural coverage analyzer. Meanwhile, the interested transactions can be specified using SOL. These transactions are further translated into a functional coverage analyzer automatically. Then the whole system, including the DUV, stimulus generator, correctness checker, and coverage analyzer, is simulated. According to the outcome from the correctness checker, we can know if the DUV conforms to the interface protocol. The EFSM-based checker can not only detect protocol violations, but also provide useful information for debugging. In traditional de-

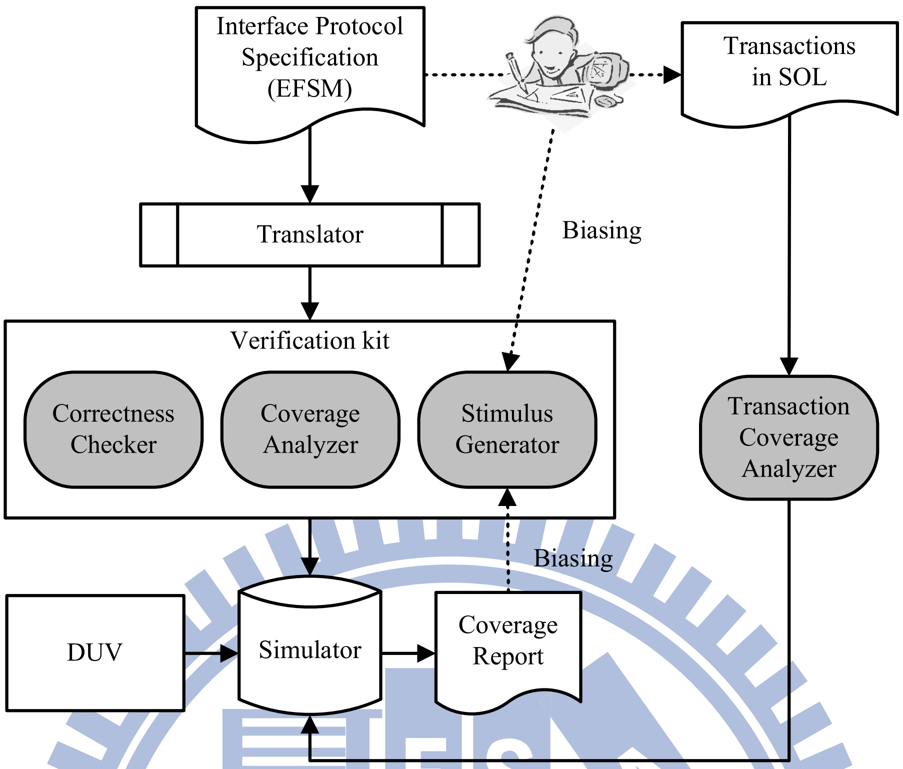


Figure 2.2: The overall flow of our verification methodology.

bugging process, designers inspect the simulation data to discover possible error sources. They usually need to manually associate the simulation waveforms with the interface behaviors. As mentioned before, in the EFSM model, each state represents a certain interface status, and each state transition represents certain interface behaviors. Tracing the simulation sequences via the EFSM model can easily link the simulation data and the desired transactions together. This makes the debugging work simpler and more intuitive. From the coverage analyzer, the report can show how many interested transactions have been verified. Moreover, the coverage information can be used to further guide the biasing options of stimulus generator to hit those unverified corner cases.

2.3 Related Works

Some research works have been proposed to build a complete simulation environment. In [14], the authors create a list of interface constraints in a proprietary specification style, which is called the "antecedent implies consequent" form. The antecedent and consequent are both Boolean formulae. When an antecedent's condition is met, the implied consequent's condition must be held simultaneously. These constraints can be directly used as assertions during simulation. This work creates BDDs from the asserted constraints on-the-fly and applies BDD traversals to find a valid stimulus. The coverage metric in this work is defined as the proportion of how many antecedent conditions are triggered during simulation. Fig. 2.3 illustrates this rule-based framework.

Another work proposed by Ara et al. [34] defines valid signal values hierarchically with a regular-expression-based (RE-based) language, CWL (Component Wrapper Language). These definitions can be converted to corresponding finite automata acting as correctness checkers during simulation. Verification engineers can manually assemble these definitions to develop required stimuli more quickly. How many states and transitions of all finite automata are visited is counted as the main coverage measurement in this approach. Fig. 2.4 illustrates the CWL-based framework.

We list the comparisons between the two existing approaches and ours in Table 2.2. Compared to previous approaches, the major contributions of our approach are:

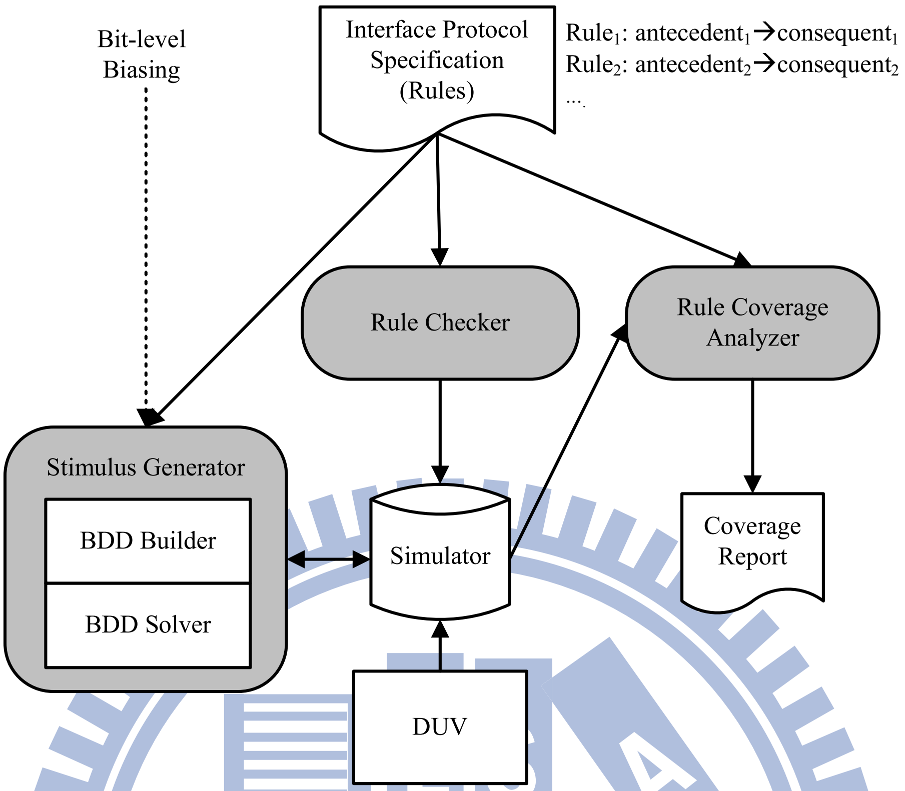


Figure 2.3: The flow of rule-based methodology.

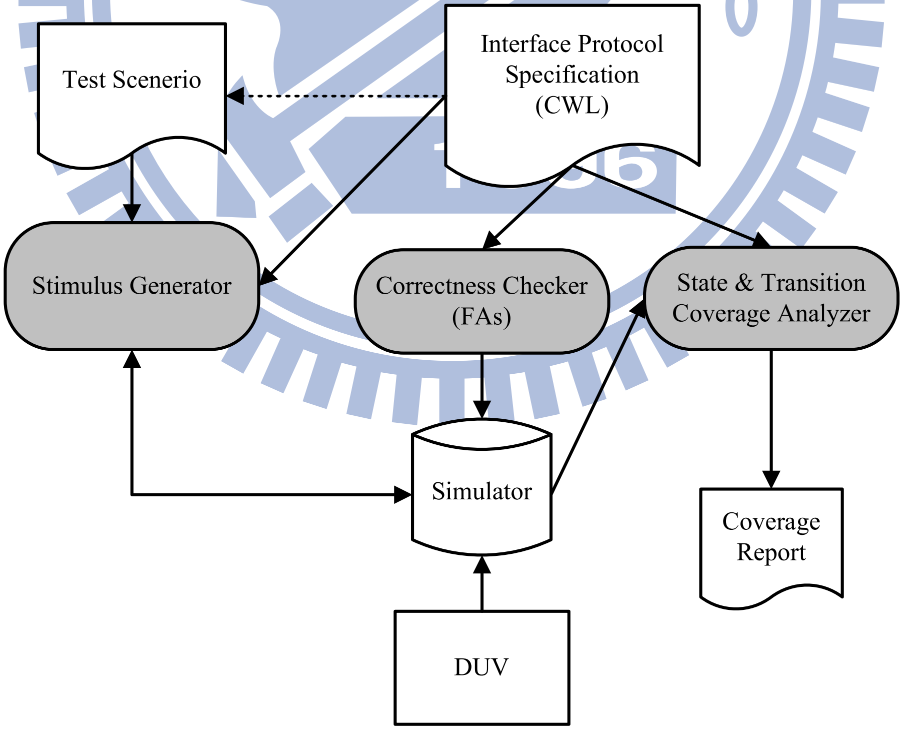

Test Scenerio
Interface Protocol Specification (CWL)
Stimulus Generator
Correctness Checker (FAs)
State & Transition Coverage Analyzer
Simulator
Coverage Report
DUV

Figure 2.4: The flow of CWL-based methodology.

Table 2.2: Comparisons among three approaches

Comparison item	**[14]**	**[34]**	**Our approach**
Modeling category	Rule-based	RE-based	EFSM-based
Allowed modeling operator	Logic operators	RE operators	Logic operators and arithmetic operators
Checker type	Rules	Finite Automata	EFSM
Stimulus generation	Fully-automatic	Semi-automatic	Fully-automatic
Stimulus biasing ability	Bit-level only	Not mentioned	Multiple biasing options
Coverage metric	Rule	State and transition	State, transition, and transaction
Hardware acceleration	Hard	Hard	Easy

- Modify the classical EFSM model to ease the description of interface : The refined EFSM model allows **a richer set of operators** providing better description power.

- Propose a methodology to obtain a set of simulation components from one specification automatically and these components are also **synthesizable** to enable **hardware acceleration** during verification.

- Support **various stimulus biasing options** to increase the stimulus effectiveness.

- Develop a **transaction description language**, State-Oriented Language (SOL) [35], to precisely evaluate how many interface functions are actually exercised.

Chapter 3

GEFSM-Based Framework

In the chapter, we introduce the GEFSM-based framework to develop a stimulus generator for interface compliance verification. As well, the diversified biasing strategies including the transition-level, transaction-level, and bit-level/word-level biasing provide users higher controllability over stimulus generation. The GEFSM-based stimulus generator is also capable of checking if the DUV conforms to the interface protocol during simulation. Furthermore, unlike SAT- or BDD-based constraint solvers, this generator can be easily implemented in synthesizable Hardware Description Language (HDL). Therefore, it is feasible to dramatically speed up the verification process via a hardware accelerator or emulator.

3.1 The Traditional EFSM Model

Definition 1 *For a variable v, its value set is D_v.*

Definition 2 *For a finite set of variables $V=\{v_1, v_2, ..., v_n\}$, its value set D_V is the n-dimensional Cartesian product $D_{v_1} \times D_{v_2} \times \ldots \times D_{v_n}$.*

The EFSM model is a finite state machine extended with internal variables. It provides 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.

Definition 3 *An EFSM is a 7-tuple (Q, Σ, Δ, X, q_0, X_0, F,U,T), where*

Q	*a set of finite states*
Σ	*a set of inputs*
Δ	*a set of outputs*
X	*a set of variables*
q_0	*the initial state, $q_0 \in Q$*
X_0	*a set of initial values for variables in X*
F	*a set of enabling functions f_i such that f_i: $D_X \rightarrow \{0,1\}$*
U	*a set of update transformation functions u_i such that u_i: $D_X \rightarrow D_X$*
T	*a set of state transition relation such that: T: $Q \times D_X \times D_\Sigma \rightarrow Q \times D_X \times D_\Delta$*

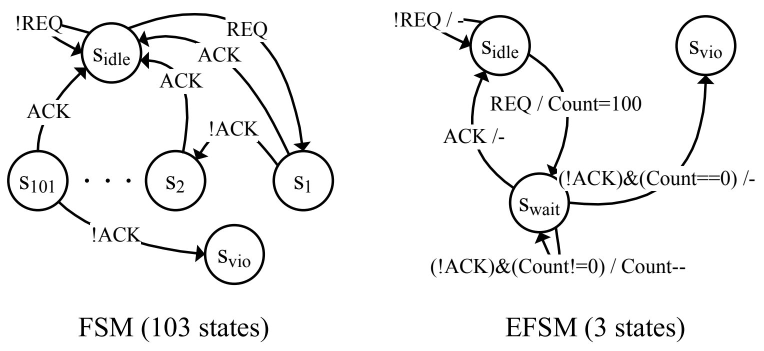


Figure 3.1: A same protocol in two different FSM representations.

Internal variables used in the EFSM model can significantly reduce the number of required states. For example, as shown in Fig. 3.1, if a protocol specifies that a slave can arbitrarily insert up to 100 wait cycles before responding to a master's request, the classical FSM requires 103 states to model this behavior while the EFSM only needs 3 states.

3.2 The GEFSM Model

Though the EFSM model has been widely used in many previous research works [36] [37] [38], we slightly modify the definition of the state transition to best fit our own need here. The modified model is called the Generator-based EFSM model which can be seem as a protocol specification from the generator's point of view.

Definition 4 *A GEFSM is a 7-tuple (Q, Σ, Δ, X, q_0, X_0, T), where*

Q *a set of finite states*

Σ *a set of inputs*

Δ	*a set of outputs*
X	*a set of variables*
q_0	*the initial state,* $q_0 \in Q$
X_0	*a set of initial values for variables in* X
T	*a set of state transitions, each transition t is a 4-tuple* (s_t, q_t, e_t, u_t)*, where*

s_t	*the current state,* $s_t \in Q$
q_t	*the next state,* $q_t \in Q$
e_t	*the transition enabling function, returning true(1)/false(0) to enable/disable the transition,* $e_t : D_\Sigma \times D_\Delta \times D_X \rightarrow \{0, 1\}$
u_t	*the update transformation function, updating the values of the subset* $S \subseteq \Delta \cup X$, $u_t : D_\Sigma \times D_\Delta \times D_X \rightarrow D_S$

Definition 5 *The GEFSM is a non-deterministic machine. That is, for any state* $s \in Q$ *and its outgoing transition set* T_s= $\{t \mid t = (s_t, q_t, e_t, u_t) \in T$ *and* $s_t = s\}$*. It is allowed that* $\exists\, t_i$*,* $t_j \in T_s$*,* $t_i \neq t_j$ *and* $d \in D_\Sigma \times D_\Delta \times D_X$*, s.t. both* $e_{t_i}(d) = 1$ *and* $e_{t_j}(d) = 1$.

3.3 An Example Protocol Specification

In order to clearly illustrate our methodology, we introduce a protocol simplified from the AMBA AHB [39] as an example. Fig. 3.2 shows the simplest transfer of this protocol.

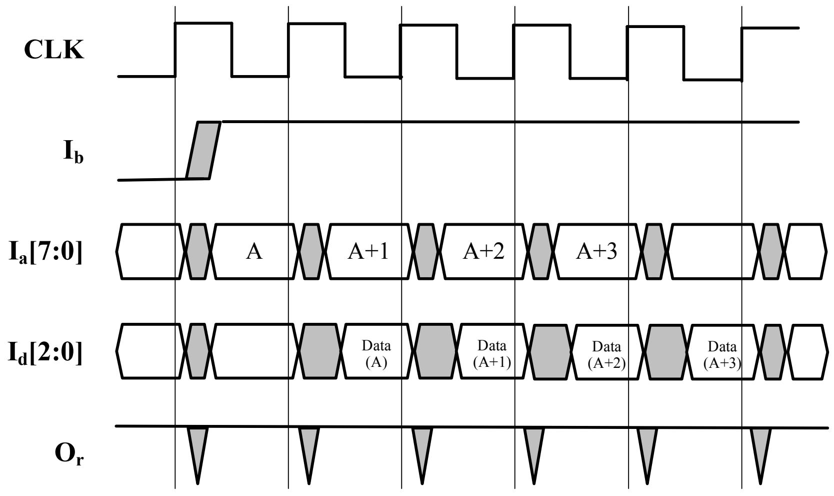


Figure 3.2: A simple transfer of a 4-beat burst.

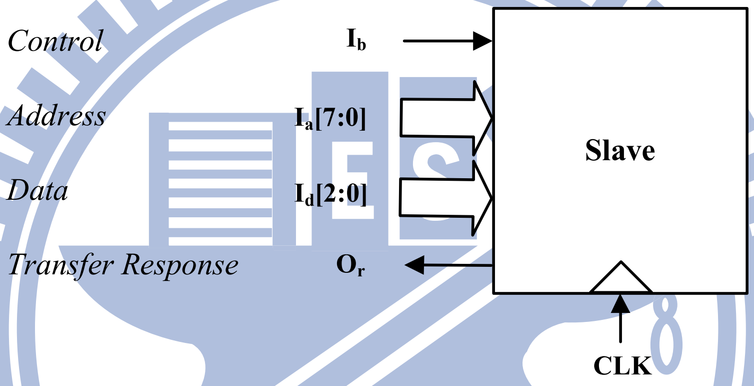


Figure 3.3: The slave interface of the simplified protocol.

This protocol defines how a slave receives a 4-beat burst from a master. As shown in Fig. 3.3, the slave only considers 4 interface signals: O_r, I_b, I_a, and I_d. Signal O_r belongs to O_{DUV} and others belogns to I_{DUV}. The following is a part of the specification about the bus master issuing a 4-beat burst to a slave:

(Item 1) A 4-beat burst includes 4 data transfers. A burst can tightly follow another burst.

(Item 2) The address (I_a) of the next transfer in a burst is equal to the address of the

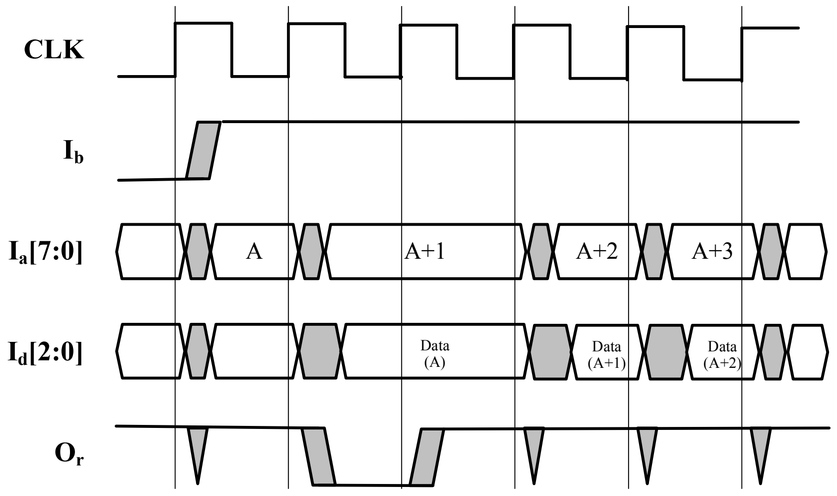


Figure 3.4: A burst with a wait state.

current transfer plus one.

(Item 3) The slave asserts the ready signal (O_r) when it is ready for the current transfer. The master should send the values of the current data (I_d) and the next address.

(Item 4) A slave can insert extra wait cycles by deasserting O_r. In this situation, all output signals of the master must hold their previous values. Fig. 3.4 shows a burst with a wait state added for the first transfer.

(Item 5) The master can choose to continue the burst ($I_b = 1$) or send a busy response ($I_b = 0$) to temporarily suspend the next transfer. Fig. 3.5 shows a burst with a busy state added for the second transfer.

According to the protocol specification, we build the corresponding GEFSM step by step as shown in Fig. 3.6 and Fig. 3.7. The final GEFSM Fig. 3.7 is shown which has four states (s_0, s_1, s_2, and s_3) and 17 transitions ($t_1 \sim t_{17}$) with an internal variable (x_1) for

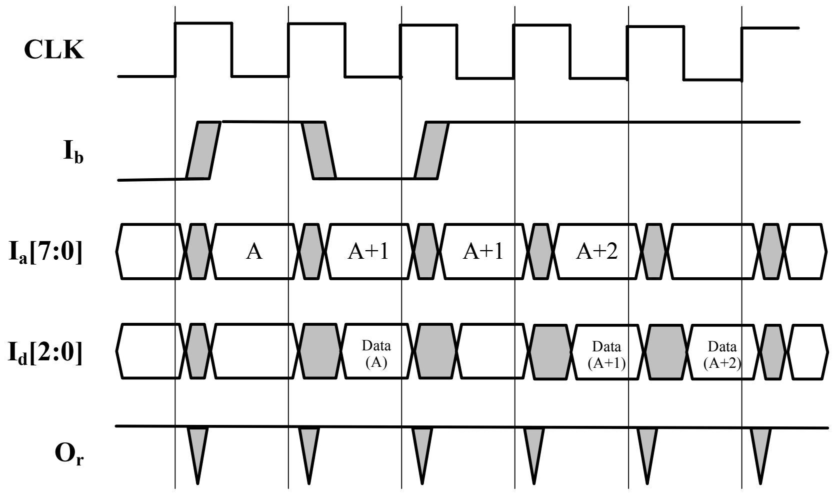


Figure 3.5: A burst with a busy state.

burst-length counting.

3.4 Stimulus Generation Flow

We develop a translator that can automatically translate a given GEFSM model into the corresponding stimulus generator. The generated stimulus generator is capable of producing massive random stimuli which are fully compliant with the given interface protocol via the corresponding GEFSM.

For a GEFSM-based stimulus generator, Fig. 3.8 illustrates the 3-phase flow about how to automatically generate the stimulus on-the-fly based on the current DUV's response during dynamic simulation:

1. **Evaluation**: At the current state, evaluate the enabling functions of all its outgoing transitions. The return values of the enabling functions are determined by the cur-

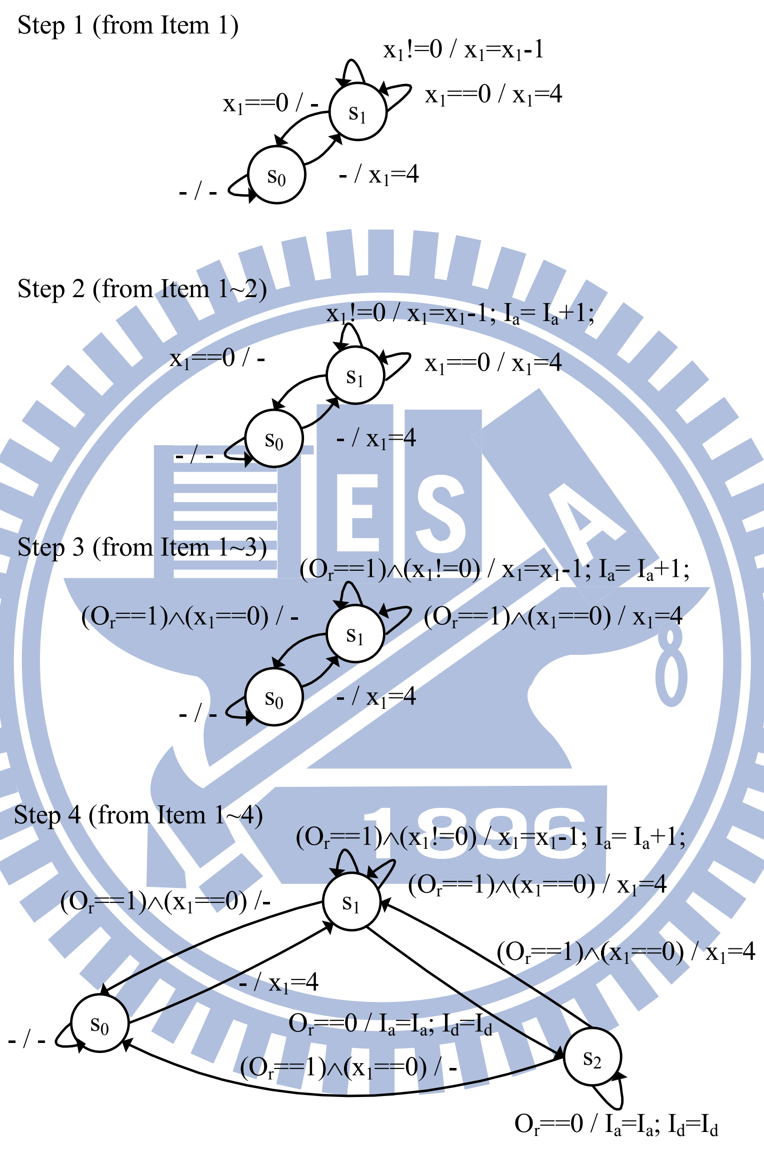
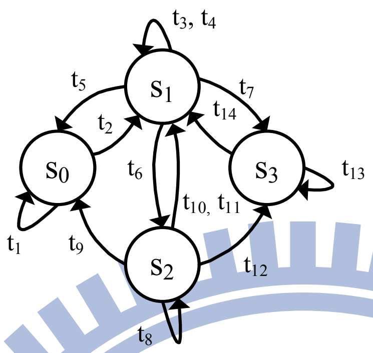

Step 1 (from Item 1)
$x_1!=0$ / $x_1=x_1-1$
$x_1==0$ / -
$x_1==0$ / $x_1=4$
s_1
s_0
- / -
- / $x_1=4$
Step 2 (from Item 1~2)
$x_1!=0$ / $x_1=x_1-1$; $I_a= I_a+1$;
$x_1==0$ / -
$x_1==0$ / $x_1=4$
s_1
s_0
- / -
- / $x_1=4$
Step 3 (from Item 1~3)
$(O_r==1)\wedge(x_1!=0)$ / $x_1=x_1-1$; $I_a= I_a+1$;
$(O_r==1)\wedge(x_1==0)$ / -
$(O_r==1)\wedge(x_1==0)$ / $x_1=4$
s_1
s_0
- / -
- / $x_1=4$
Step 4 (from Item 1~4)
$(O_r==1)\wedge(x_1!=0)$ / $x_1=x_1-1$; $I_a= I_a+1$;
$(O_r==1)\wedge(x_1==0)$ / $x_1=4$
$(O_r==1)\wedge(x_1==0)$ /-
s_1
$(O_r==1)\wedge(x_1==0)$ / $x_1=4$
- / $x_1=4$
s_0
- / -
$O_r==0$ / $I_a=I_a$; $I_d=I_d$
$(O_r==1)\wedge(x_1==0)$ / -
s_2
$O_r==0$ / $I_a=I_a$; $I_d=I_d$

Figure 3.6: The protocol modeling steps with GEFSM.

Step 5 (from Item 1~5)



GEFSM
$\sum$: O_r
Δ: I_a, I_b, I_d
State: S_0, S_1, S_2, S_3
Variable: x_1

Transition definitions:
$t=(s_t, q_t, e_t, u_t)$

$t_1=(S_0, S_0, -, \{I_b==0\})$
$t_2=(S_0, S_1, -, \{I_b==1; x_1=4\})$
$t_3=(S_1, S_1, (O_r==1)\wedge(x_1!=0), \{I_b=1; I_a=I_a+1; x_1=x_1-1\})$
$t_4=(S_1, S_1, (O_r==1)\wedge(x_1==0), \{I_b=1; x_1=4\})$
$t_5=(S_1, S_0, (O_r==1)\wedge(x_1==0), \{I_b=0\})$
$t_6=(S_1, S_2, O_r==0, \{I_a=I_a; I_d=I_d\})$
$t_7=(S_1, S_3, (O_r==1)\wedge(x_1!=0), \{I_b=0; I_a==I_a+1; x_1=x_1-1\})$
$t_8=(S_2, S_2, O_r==0, I_a=I_a; I_d=I_d)$
$t_9=(S_2, S_0, (O_r==1)\wedge(x_1==0), \{I_b=0\})$
$t_{10}=(S_2, S_1, (O_r==1)\wedge(x_1==0), \{I_b=1; x_1=4\})$
$t_{11}=(S_2, S_1, (O_r==1)\wedge(x_1!=0), \{I_b=1; I_a=I_a+1; x_1=x_1-1\})$
$t_{12}=(S_2, S_3, (O_r==1)\wedge(x_1!=0), \{I_b=0; I_a=I_a+1; x_1=x_1-1\})$
$t_{13}=(S_3, S_3, O_r==1, \{I_b=0; I_a=I_a; I_d=I_d\})$
$t_{14}=(S_3, S_1, O_r==1, \{I_b=1; I_a=I_a; I_d=I_d\})$

Figure 3.7: The GEFSM of the example protocol.

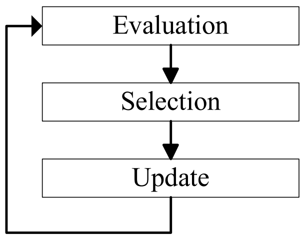


Figure 3.8: Stimulus generation flow.

rent values of the interface signals (Σ, Δ) and internal variables (X). A transition is put into the *next transition candidate set* (NTCS) only if its enabling function is evaluated true. Some transitions might be evaluated false and excluded in the NTCS. It actually means the current signal values on the interface prevent the stimulus generator from producing certain stimuli for the next cycle. In other words, the stimulus generator implicitly solves constraints presented by the given protocol during the stimulus generation process. Meanwhile, in case the NTCS is empty after the evaluation, it means the behavior of DUV must violate the protocol because it makes the stimulus generator find no valid move for the next cycle. That is, the stimulus generator can not only generate the valid stimuli but also serve as a protocol compliance checker at the same time.

2. **Selection**: From the non-empty NTCS, randomly pick one as the next transition based on the given transition *weights*. A transition with a higher weight has a higher probability to be selected as the next transition. Hence, some sort of biasing strategies can be utilized here to meet volatile requirements of users. Meanwhile, the next transition becomes *determinate* at this phase no matter what selection strategy

is in use.

3. **Update**: Assign values to the outputs and variables according to the update transformation function of the selected transition. This phase consists of the constrained and unconstrained parts:

   (a) **Constrained part**: Outputs and variables explicitly defined in the update transformation function must be assigned with the constrained values.

   (b) **Unconstrained part**: Outputs not defined in the update transformation function can be randomly assigned with any valid values within their own domains. Again, some kind of biasing strategies can be used here.

   Actually, the mission of the Update phase is to generate a complete and valid stimulus. After the Update phase, the GEFSM moves to the next state through the selected transition and then the stimulus generation process goes back to the Evaluation phase for the next cycle.

Now we use the stimulus generator built from the GEFSM in Fig. 3.7 to demonstrate this flow. Assume the current state is s_1:

Case 1 (current state = S_1**,** $I_a = 20, I_b = 1, I_d = 0, O_r = 1, V_b = 3$)

The enabling functions e_{t_3} and e_{t_7} are both evaluated true at the Evaluation phase. It means the two corresponding transitions, t_3 and t_7, are the next transition candidates.

Next, at the Selection phase, one of the candidates would be chosen as the next transition. Assume the transition t_7 is selected, the constrained outputs and variables of u_{t_7} need to be updated as the defined values – assigning I_a, I_b, and X_1 to 21, 0, and 2, respectively. The remaining unconstrained outputs can be assigned to any valid values. For instance, we can set I_d to 2. Finally, the current state moves from s_1 to s_3.

Case 2 (current state = S_3, $O_r = 0$)

In this case, all enabling functions return false at the Evaluation phase. In other words, no possible transition exists and a protocol violation is detected by the stimulus generator. Meanwhile, it terminates the current simulation and reports the error.

3.5 Stimulus Biasing

Biasing techniques can help verification engineers generate stimuli to hit desirable corner cases more easily. BDD-based approaches can generally support the stimulus biasing. However, due to the BDD's inherent topology, only bit-level signals can be biased in these approaches. Instead, our stimulus generator is capable of achieving an even higher flexibility, shown later, to bias the generated stimuli. This feature is extremely useful to exercise those uncovered scenarios to get a better simulation quality.

Table 3.1: Biasing information I

Biasing type	**Weight**
Transition-level	$W_{t_3} = 20$
	$W_{t_4} = 40$
	$W_{t_5} = 40$
	$W_{t_6} = 100$
	$W_{t_7} = 80$
Word-level	$W_{d=2'b00} = 5$
	$W_{d=2'b01} = 40$
	$W_{d=2'b10} = 40$
	$W_{d=2'b11} = 15$

3.5.1 Transition-level Biasing / Transaction-level Biasing

Since each transition indicates certain interface behavior, we use the *transition-level biasing* to guide the state transition. Due to the non-determinism, there may exist multiple valid transitions after the Evaluation phase. A strategy is needed to pick one from these candidates. One method is to give each transition t an individual weight w_t. Then, the probability that a candidate transition t_i is selected is defined as:

$$P_{t_i} = \frac{w_{t_i}}{\sum_{t_j \in NTCS} w_{t_j}}, \; t_i \in \text{NTCS}$$

For example, in Case 1 of the previous example, t_3 and t_7 are both transition candidates. If the biasing information is given as Table 3.1, t_7 has a higher probability (80%) to be chosen than t_3 (20%) does. Furthermore, if preliminary simulation results do not exercise certain states or transitions, the related transition weights can be increased accordingly. Another similar approach is the *transaction-level biasing*. We can define a meaningful transaction in terms of a sequence of transitions and then bias all these transitions at the

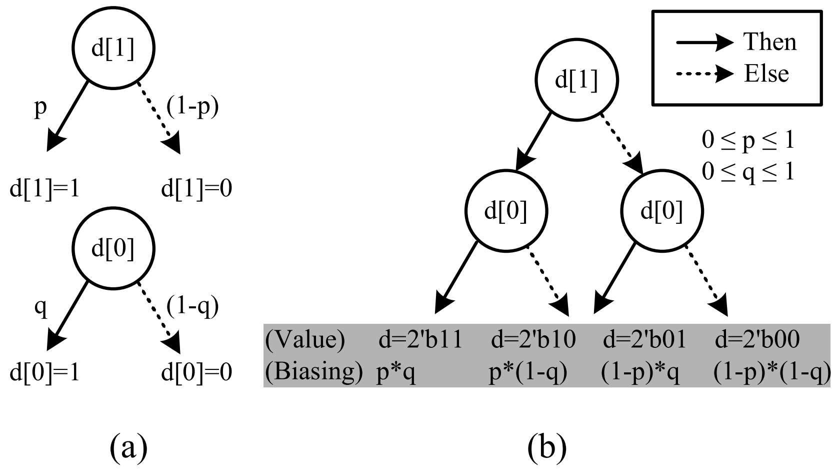


Figure 3.9: An example of bit-level biasing.

same time.

3.5.2 Bit-level biasing / Word-level biasing

As mentioned before, bit-level biasing is available while BDD-based approaches are adopted. The basic idea of these approaches is to traverse BDDs to obtain a valid solution. Fig. 3.9 is an example to demonstrate the bit-level biasing scheme. In Fig. 3.9(a), there are two BDD nodes representing distinct bit signals, d[0] and d[1]. Each node has two outgoing branches “Then” and “Else” to indicate that the signal is assigned to 1 or 0. General bit-level biasing schemes set a probability value along with each branch to affect the BDD traversal outcomes. For example, p and q are bit-level biasing parameters for “d[1]=1” and “d[0]=1” in Fig. 3.9, respectively.

In typical protocols, many signals are defined in a group of bits, i.e., a word, instead of a single bit only. Therefore, the capability of the word-level biasing becomes critical and essential. As shown in Fig. 3.9(b), bit-level biasing settings can result in different word-level biasing. However, bit-level biasing has limitations in many situations. For example, in our approach, we allow the weight settings for different word values of d as shown in Table 3.1 to apply direct word-level biasing. This biasing setting simultaneously increases the appearance probabilities of “d=01” and “d=10”. Note that, under bit-level biasing, the positive biasing of “d=01” implies the negative biasing of “d=10”. Obviously, there is no way for bit-level biasing to achieve the distribution of word-level biasing given in Table 3.1.

In our approach, the word-level biasing settings can affect the generated stimuli in two manners. On the one hand, while the biased signal is in the unconstrained part of the Update phase, the signal’s value can be directly produced via a weighted random number generator with the distribution specified by the word-level biasing. On the other hand, if the biased signal appears in the constrained part, i.e., the signal value relates to which transition is selected, the biasing effect could be reflected by changing the transition weights. Assume the word-level biasing list for an n-bit signal s is “$W_{s=0}$, $W_{s=1}$, ..., $W_{s=2^n-1}$” and $C^t_{s=i}$ is a binary variable indicating if “$s = i$” is feasible while selecting t as the next transition, for $0 \leq i \leq 2^n - 1$. While the original transition weight of the transition t is w_t, the modified transition weight according to the word-level biasing is:

$$w'_t = w_t * \frac{\sum_{i=0}^{2^n-1} C^t_{s=i} * W_{s=i}}{\sum_{i=0}^{2^n-1} W_{s=i}}$$

Table 3.2 shows an example of this mechanism. When the word-level weights of the signal b are given, the original transition weights, as shown in Table 3.1, can be modified according to the update transformation functions of those transitions. While t_3 and t_7 constrain the value of I_b to 1 and 0, respectively, the modification ratio, W'_{t_3}/W_{t_3}:W'_{t_7}/W_{t_7}, is 3:1, which is the same as the word-level biasing setting of I_b ($W_{I_b=1}$:$W_{I_b=0}$). After this adjustment, the transitions tending to generate signal values with higher word-level weights should appear more frequently.

3.5.3 Feedback for Biasing Refinements

By understanding the design intent and desired scenarios, engineers can manually change those biasing settings to improve simulation quality in later simulation runs. Besides, we could also build a weight tuner which can collect current simulation information and adjust the biasing settings automatically.

In general, a weight tuner should follow certain formulae or strategies to bias the simulation to unverified corners. In other words, the new weight settings should raise the appearance probability of those stimuli which are seldom or even not produced in the previous stimulus set. For example, here is a simple strategy for increasing the transition coverage of GEFSM and balancing every transition's appearance count as far as possible. Assume current outgoing transitions are $\{t_1, t_2, ..., t_n\}$ and the corresponding weight set is W=$\{w_{t_1}, w_{t_2}, ..., w_{t_n}\}$. If t_i is selected as the next transition and w_{t_i} is greater than n,

Table 3.2: Biasing information II

Biasing type	**Weight**
Word-level	$W_{I_b=0} = 1$
	$W_{I_b=1} = 3$
	Modified weight
Transition-level	$W'_{t_3} = 80 * (3/4) = 60$
	$W'_{t_4} = 20 * (3/4) = 15$
	$W'_{t_5} = 40 * (1/4) = 10$
	$W'_{t_6} = 100 * (4/4) = 100$
	$W'_{t_7} = 80 * (1/4) = 20$

we can apply the following weight tuning formula to every $w_{t_j} \in$ W:

$$w_{t_j} = \begin{cases} w_{t_j} - n, & \text{for j=i} \\ w_{t_j} + 1, & \text{for j}\neq\text{i} \end{cases}$$

By decreasing the weight of frequently appeared transition and increasing weights of others, this kind of feedback mechanism can dynamically modify biasing settings during simulation. Hence, more varied stimulus sequences can be obtained using an automatic weight tuner.

3.6 Automatic Translation

We implement an translator as shown in Fig. 3.10. The translator first reads in the given GEFSM of the specific interface protocol and the biasing information. It then automatically produces a corresponding stimulus generator in target HDL form. The upper part in Fig. 3.10 is a typical simulation-based verification environment. DUVs are usually written in HDL. However, high-level testbenches or stimulus generators are often devel-

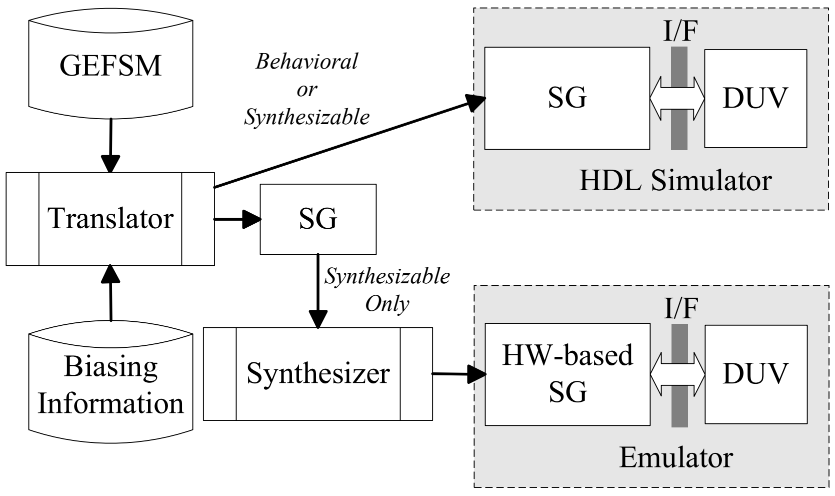


Figure 3.10: Stimulus generator translation flow.

oped in C/C++, so they need to interact with the HDL simulator via the Programming Language Interface (PLI). Extra simulation overhead is required due to the PLI communication need. Since our stimulus generator is implemented in native HDL, it can thus save simulation time compared to those approaches using PLI.

Fig. 3.11 shows the interface of the stimulus generator. The primary input ports (Σ) and output ports (Δ) of generated stimulus generator are defined in GEFSM model. Besides, it has an additional output port, "FAIL", to indicate whether any violation occurs. This translation process can be finished in the following three steps:

(Step 1) First, the compiler translates the enabling functions into assignment statements. For example, the enabling function e_{t_3} in Fig. 3.7 can be written with the following Verilog statement:

$$e_{t_3} = (O_r == 1)\&\&(x_1 != 0);$$

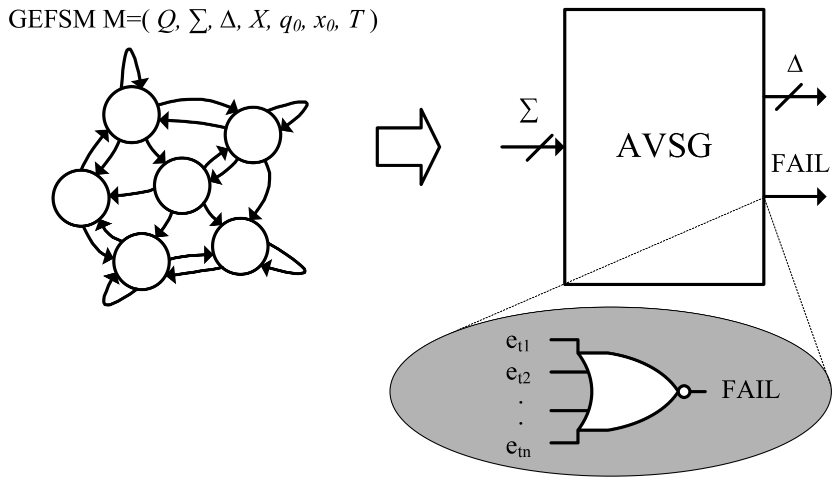


Figure 3.11: The interface of the stimulus generator.

In this statement, e_{t_3} is a binary variable representing the evaluation result. The evaluation results for all outgoing transitions of the current state pass through an NOR gate and then drive the output signal "FAIL" as shown in Fig. 3.11. As explained, it implies if no available transition for the next move, the DUV must have some design error.

(Step 2) Unlike the deterministic machine, the GEFSM has a set of transition candidates. In our approach, the stimulus generator chooses the next transition according to the given transition weights (transition-level biasing). We use a weighted selection procedure as shown in Fig. 3.12 to realize this mechanism. First, it sums the weights of transitions in NTCS via additions or a lookup-up table. Then, it generates a random number between 0 and the weight sum. Typically, random numbers can be obtained from a software call provided by most HDL simulators. According to this number and weight distribution, a decoder can determine the selection result.

(Step 3) The major task of the stimulus generator is to generate proper values for output signals which are constrained by the selected transition. On one hand, the constraints are defined in the update transformation functions, and the compiler directly translates them into assignment statements. For example, the update transformation function u_{t_3} can be implemented with three statements in Verilog:

$$\begin{aligned} x_1 &= x_1 - 1; \\ I_a &= I_a + 1; \\ I_b &= 1; \end{aligned}$$

On the other hand, the weighted selection procedure is used to assign the unconstrained output signals with weighted random values (bit-level/word-level biasing).

Synthesizable stimulus generator

Practically, simulation jobs are very time-consuming for large complex SOC designs. Hence hardware accelerators or emulators are usually utilized to speed up the verification process around 100 $\sim$ 1000 times. The lower part of Fig. 3.10 demonstrates this acceleration environment. In general, SAT- or BDD-based generators are inherently hard

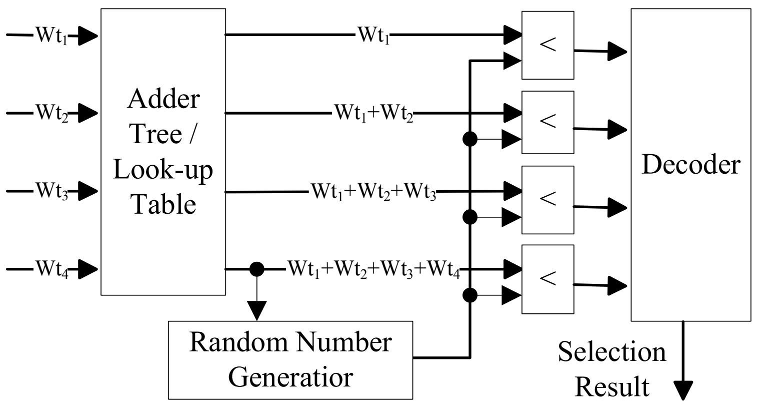


Figure 3.12: Weighted selection procedure.

to be synthesized to hardware, and thus prevent such kind of acceleration. Conversely, the proposed translator is capable of generating the stimulus generator in synthesizable HDL form on one condition – only synthesizable operators are allowed in the transition enabling and update transformation functions of the given GEFSM. In our experiences, the set of synthesizable operators are generally large enough for modeling most of interface protocols. Furthermore, random numbers are required to implement the weighted selection schemes requested by the Selection and Update phases. This hardware implementation is similar to the hardware Lottery manager circuit in [40]. To make our stimulus generator fully synthesizable, a hardware-based LFSR (Linear Feedback Shift Register) [41] is used. The combination of these 2 efforts enables the truly hardware-based stimulus generator as well as the use of hardware acceleration techniques. Fig. 3.13 shows the block diagram of the proposed GEFSM-based stimulus generator.

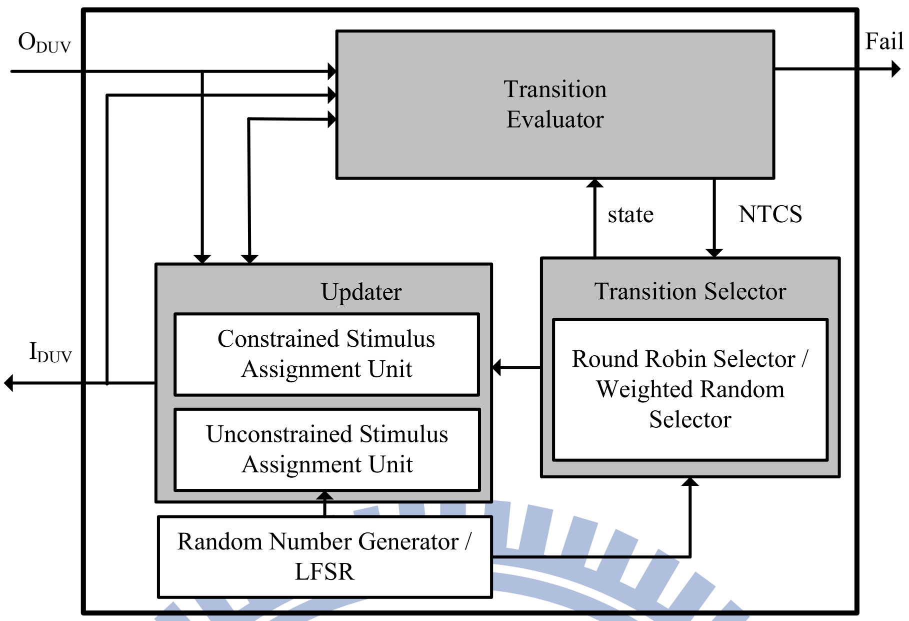

O_DUV
Fail
Transition Evaluator
state
NTCS
Updater
Transition Selector
Constrained Stimulus Assignment Unit
I_DUV
Round Robin Selector / Weighted Random Selector
Unconstrained Stimulus Assignment Unit
Random Number Generator / LFSR

Figure 3.13: The block diagram of the proposed GEFSM-based stimulus generator.

3.7 Experimental Results

To demonstrate the effectiveness and efficiency of our approach, we use the WISHBONE [42] and AMBA AHB protocols as the test cases. The experiments are conducted over a set of 3 WISHBONE-compliant and 3 AHB-compliant bus slave designs. The WISHBONE test cases are obtained from the OPENCORES organization [43], and the others are internal designs. Table 3.3 shows the basic information of each design. All experiments are performed on a Sun Blade-2000 workstation with 1GB RAM. Simulation is conducted using a Cadence Verilog-XL simulator.

We first model both the WISHBONE and AHB protocols in GEFSM. The resultant GEFSM of the WISHBONE requires 4 states and 17 transitions while AHB's requires 6 states and 46 transitions. Then the target stimulus generators are directly generated from

Table 3.3: Basic information of selected DUVs

Design	Description	Protocol
AC97	Simple AC97 controller	WISHBONE
SPI	Serial Peripheral Interface	WISHBONE
PTC	PWM/Timer/Counter	WISHBONE
RGB2YCrCb	RGB-to-YCrCb Translator	AMBA AHB
CON	Convolution Calculator	AMBA AHB
MAC	Multiply-accumulator	AMBA AHB

the corresponding GEFSMs through the proposed compiler.

Overall experimental results confirm that generated stimulus generators are fully capable of providing a large amount of valid verification stimuli for all 6 designs. Next, we report certain detailed experimental results to show the power of biasing and the run-time efficiency of our stimulus generation.

3.7.1 Stimulus Biasing

We apply the proposed biasing methods on the experiments over the design *CON*. We first show the effectiveness of the transaction-level biasing. Suppose each transition initially has equal weight and users want to exercise a five-cycle transaction, "Nonseq $\rightarrow$ Busy $\rightarrow$ Seq $\rightarrow$ Busy $\rightarrow$ Seq". However, this transaction never occurs in the initial unbiased simulation of 1000 cycles. Then we increase the weights of related transitions (e.g., Nonseq$\rightarrow$Busy) by 10 times. Under the new bias setting, the expected transaction appears 19 times in first 1000 cycles. It demonstrates that the transaction-level biasing technique can help achieve higher functional coverage in shorter simulation cycles.

Another experiment focuses on performing the word-level biasing on the AHB signal

Table 3.4: Results of word-level biasing

HBURST	**Burst type**	**Weight**	**Count**
000	SINGLE	10	13022 (9.95%)
001	INCR	20	25996 (19.86%)
010	WRAP4	40	52377 (40.02%)
011	INCR4	5	6521 (4.98%)
100	WRAP8	15	19645 (15.01%)
101	INCR8	0	0 (0%)
110	WRAP16	0	0 (0%)
111	INCR16	10	13317 (10.18%)

HBURST. *HBURST* is a three-bit signal indicating the current burst type as shown in the first and second columns of Table 3.4. For certain verification need, we set the expected probabilities on different *HBURST* values in terms of the word-level weights. Note that no single bit-level bias setting on *HBURST* can produce the identical word-level biasing shown in the third column in Table 3.4. After 1 million simulation cycles, the appearance count of each burst type is reported in the last column. The results perfectly match the given bias setting. This experiment also implies that we can disable certain types of stimuli by setting the corresponding weights as zeros. This skill is extremely useful when a DUV does not fully implement all features specified in the interface protocol. In short, through the proposed biasing techniques, it becomes much easier to get the desired stimuli.

3.7.2 Performance Analysis

Shorter stimulus generation time should be always preferred during verification. To illustrate the performance of the stimulus generation, the run-time is compared with those of

Table 3.5: Run-time analysis of different stimulus generators

Design	**Stimulus generator**			
	PRSG	*Ours*	SG_1	SG_2
AC97	117.56 s	123.04 s	136.92 s	139.81 s
SPI	19.45 s	24.03 s	36.60 s	37.92 s
PTC	14.88 s	17.42 s	31.52 s	33.90 s
RGB2YCrCb	11.10 s	12.20 s	23.97 s	24.46 s
CON	10.43 s	11.37 s	24.01 s	25.09 s
MAC	12.80 s	13.82 s	22.11 s	22.55 s
Total	186.22 s	201.88 s	275.13 s	283.73 s
Ratio	0.92	1.00	1.36	1.41

other stimulus generation methods. We build four simulation environments which contain different stimulus generators. The run-time (for 1 million simulation cycles) for 6 real designs in each simulation environment is reported in Table 3.5. In the first environment, we use a *pure random stimulus generator* (PRSG) to produce stimuli. This is the most trivial way to generate massive random stimuli at virtually no cost. The second environment is to use our stimulus generator instead of the PRSG. From Table 3.5, the difference of run-time required by the PRSG and our stimulus generator is quite small. It shows our stimulus generator can be as efficient as a PRSG. However, the PRSG generally produces invalid stimuli while ours only generates stimuli fully compliant with the protocol. Besides, we use CUDD (Colorado University Decision Diagram) [44] package to build two BDD-based implementations, referred to as SG_1 and SG_2, which are similar to the stimulus generators in [13] and [14], respectively. Since PLI mechanism is required between the HDL simulator and BDD-solving engine for these two environments, the experimental results show that these two environments take averagely 36% and 41% more run-time than our GEFSM-based approach.

3.7.3 Error Detection

While the GEFSM model describes the target interface protocol, the stimulus generator can also detect those design errors which violate the interface protocol. For demonstrating this feature, we do inject protocol-related errors into designs, and the experimental results show that the stimulus generator can indeed capture all these kinds of design errors. We also inject protocol-independent errors to correct designs. For example, one such injected error is to change the internal computing function of the design *RGB2YCrCb*. As expected, this kind of error can not be detected by our stimulus generator since the error effect does not affect the interface behavior. To detect those internal design bugs, users need to build other checkers to investigate the simulation results and the stimulus generator purely serves as a stimulus generator in this case.

3.7.4 Synthesis Results

As mentioned, our stimulus generator can be easily mapped to real hardware through a commercial logic synthesizer. We use Synoposys Design Vision as the synthesizer under UMC 0.18 technology. The synthesizer reports that only 1.8K and 5.7K NAND2-equivalent gates are required to implement the stimulus generators of the WISHBONE and AMBA AHB protocols, respectively. This result clearly shows that the proposed stimulus generator can be easily and cost-effectively integrated into a high-performance emulator-based verification flow.

3.8 Summary

We propose a GEFSM-based approach for interface compliance verification. In this approach, the GEFSM model is used to represent the interface specification from the generator's view and then automatically translated into a dedicated stimulus generator. This stimulus generator can produce a large amount of valid random stimuli and simultaneously check the correctness of the interface behavior. In addition, it can be easily implemented in synthesizable HDL to enable the hardware acceleration. By supporting transaction-, transition-, and word-level biasing, the stimulus generator provides users better controllability over where a simulation run heads for. The experimental results demonstrate that these biasing methods do successfully generate appropriate stimuli as expected. Moreover, the extremely low overhead in simulation time shows the remarkably high efficiency of our stimulus generation. Hence, this approach can indeed improve the simulation quality as well as speed up the verification process.

Although the proposed GEFSM can be translated to a stimulus generator which can also check the simulation traces simultaneously, this approach cannot build a pure checker. That is, the protocol checking mechanism is along with the stimulus generation. This may limit the usage model of the GEFSM-based approach while engineers may sometimes require a pure protocol checker to examine the interface behaviors triggered by another stimulus generator or manual stimuli. In the next chapter, we proposed a CEFSM-based methodology which is capable of producing a single stimulus generator, protocol checker, or coverage analyzer to handle different verification requirements.

Chapter 4

CEFSM-Based Framework

In this chapter, a CEFSM-based unified framework is introduced to generate a complete verification kit for interface compliance verification. Compared to the GEFSM-based approach, the correctness checker translated from a CEFSM can operate under other stimulus generation schemes. We also introduce a transaction-level coverage metric which is proper to define transactions within an EFSM.

4.1 The CEFSM Model

Definition 6 *Given a function $f : D_a \rightarrow D_b$ with respect to mapping variable a to b, the* ***relation function*** $R : D_a \times D_b \rightarrow \{0, 1\}$ *is defined as:*

$$R(a,b) = \begin{cases} 1 & , if\ b == f(a)\ where\ a \in D_a,\ b \in D_b \\ 0 & , otherwise \end{cases}$$

if we have to cope several functions $\vec{f} : D_A \rightarrow D_B$ with the variables A=$\{a_1, a_2, \dots, a_n\}$ and B=$\{b_1, b_2, \dots, b_m\}$, then the relation $R : D_A \times D_B \rightarrow \{0, 1\}$ is defined as:

$$R(A,B) = \bigwedge_{i=1}^{m} (b_i == f_i(a_1, a_2, \dots, a_n))$$

Definition 7 *The existential quantification of a function $f(x_1,\ x_2,\ \dots,\ x_i,\ \dots,\ x_n)$ with respect to a binary variable x_i, denoted as $\exists x_i f$, is $f_{x_i=1} \vee f_{x_i=0}$, where $f_{x_i=1}$ = $f(x_1,\ x_2,\ \dots,\ x_i = 1,\ \dots,\ x_n)$ and $f_{x_i=0}$ = $f(x_1,\ x_2,\ \dots,\ x_i = 0,\ \dots,\ x_n)$.*

Definition 8 *The existential quantification of a function $f(x_1,\ x_2,\ \dots,\ x_i,\ \dots,\ x_n)$ with respect to a variable x_i, whose value set D_{x_i}=$\{k_1,\ k_2,\ \dots,\ k_m\}$, is $\exists x_i f$ = $f_{x_i=k_1} \bigvee f_{x_i=k_2} \bigvee \cdots \bigvee f_{x_i=k_m}$.*

Definition 9 *The existential quantification of a function f with respect to a set of variables X=$\{x_1,\ x_2,\ \dots,\ x_k\}$, is defined as a sequence of single-variable operations: $\exists X f$=*

$\exists x_1(\exists x_2 \ldots (\exists x_k\, f))$.

A checker-based EFSM model is modified from the traditional EFSM model to better fit the need for describing a general interface protocol specification from a checker's point of view.

Definition 10 *A CEFSM M is a 7-tuple (Q, Σ, Σ', X, q_0, X_0, T), where*

Q	*a finite set of states*
Σ	*a set of signals*
Σ'	*a copy of Σ to hold the values of Σ in the last state transition*
X	*a set of variables*
q_0	*the initial state, $q_0 \in Q$*
X_0	*a set of initial values for variables in X*
T	*a set of state transitions, each transition t is a 5-tuple (s_t, q_t, r_t, p_t, a_t), where*

s_t	*the current state, $s_t \in Q$*
q_t	*the next state, $q_t \in Q$*
r_t	*the relation function with respect to $\vec{f_t} : D_{\Sigma'} \to D_{\Sigma_t}$, where Σ_t is a subset of Σ in which each signal is related to Σ'. $r_t : D_{\Sigma'} \times D_{\Sigma_t} \to \{0,1\}$ can be written as:*

$$r_t(\Sigma', \Sigma_t) = \bigwedge_{for\ each\ b_i\ \in\ \Sigma_t} (b_i == f_{t_i}(\Sigma'))$$

p_t *the predicate function,* $p_t : D_X \rightarrow \{0, 1\}$

a_t *the action function,* $a_t : D_X \rightarrow D_X$

Assume that the current signal values (σ), previous signal values (σ'), and current variable values (x) are given. Performing a state transition t_i=(s_{t_i}, q_{t_i}, r_{t_i}, p_{t_i}, a_{t_i}) means that M is initially in the state s_{t_i} with those given values such that $r_{t_i}(\sigma', \sigma)$=1, and $p_{t_i}(x)$=1, then M moves to the state q_{t_i} and updates the values of variables according to $a_{t_i}(x)$. In this case, $r_{t_i}(\sigma', \sigma)$=1 and $p_{t_i}(x)$=1 are two necessary conditions of the transition t_i. We call the conjunction of r_{t_i} and p_{t_i}, i.e., $r_{t_i} \wedge p_{t_i}$, the *transition condition (TC)* of t_i. While M moves to the state s, the next state transition must be within its outgoing-transition set $T_{next}(s)$= $\{t \mid t = (s_t, q_t, r_t, p_t, u_t) \in T$ and $s_t = s\}$.

In our approach, we use this CEFSM model to describe the interface protocol specification from a checker's viewpoint. Σ is identical to the set of all interface signals ($I_{DUV} \cup O_{DUV}$). Each state indicates certain interface status, and the set of transitions represents all possible status transformations. Compared with previous approaches, two different features make this CEFSM model easier to specify an interface protocol. The first feature is that the transition conditions are written as functions to make the modeling capability more flexible and powerful. Especially when the interface protocol specification contains the following behaviors:

- **Retention behavior**: A retention behavior means that a signal must retain its previous value. For example, in AMBA AHB protocol, while a binary signal *HREADY* is low, the multi-bit signal *HTRANS* must hold its previous value.

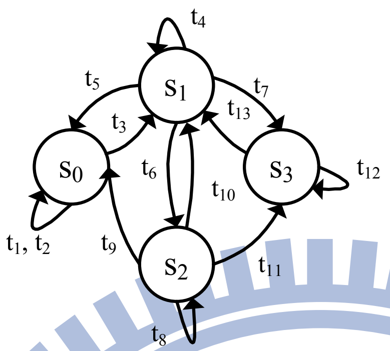
- **Arithmetic behavior**: An arithmetic behavior means that the current value of a signal can be arithmetically calculated. For example, during a burst transfer, the value of the multi-bit signal *HADDR* is incremented by one at each subsequent cycle.

These two behaviors can be written in the relation functions, " $(HREADY == 1) \wedge (HTRANS == HTRANS')$ " and "$(HADDR == HADDR' + 1)$", respectively. Obviously, these kinds of behaviors can not be easily represented with pure Boolean formulae or regular expressions. The second feature is that the predicate and action functions on variables can significantly reduce the number of required states and transitions. For example, the address of a fixed-length burst transfer can be described with a counter variable to make the state machine more compact.

Similarly, according to the protocol introduced in Section 3.3, we build the corresponding CEFSM as shown in Fig. 4.1. It has 4 states, 4 signals, one internal variable, and 13 transitions. The bottom of Fig. 4.1 lists the transition conditions.

4.2 CEFSM-based Correctness Checker

In this section, we show the role an CEFSM-based checker play during simulation. A CEFSM-based checker can be easily generated according to a given CEFSM model. As



CEFSM
$\sum$: O_r, I_a, I_b
$\sum'$: O_r', I_a', I_b'
State: S_0, S_1, S_2, S_3
Variable: x_1

Transition definitions:
$t=(s_t, q_t, r_t, p_t, a_t)$

$t_1=(S_0, S_0, O_r==0, -, -)$
$t_2=(S_0, S_0, (O_r==1)\wedge(I_b==0), -, -)$
$t_3=(S_0, S_1, (O_r==1)\wedge(I_b==1), -, x_1=3)$
$t_4=(S_1, S_1, (O_r==1)\wedge(I_a==I_a'+1)\wedge(I_b==1), x_1>0, x_1=x_1-1)$
$t_5=(S_1, S_0, (O_r==1)\wedge(I_a==I_a'+1)\wedge(I_b==1), x_1==0, -)$
$t_6=(S_1, S_2, (O_r==0)\wedge(I_a==I_a'+1), -, -)$
$t_7=(S_1, S_3, (O_r==1)\wedge(I_a==I_a'+1)\wedge(I_b==0), x_1>0, x_1=x_1-1)$
$t_8=(S_2, S_2, (O_r==0)\wedge(I_a==I_a'), -, -)$
$t_9=(S_2, S_0, (O_r==1)\wedge(I_a==I_a')\wedge(I_b==1), x_1==0, -)$
$t_{10}=(S_2, S_1, (O_r==1)\wedge(I_a==I_a')\wedge(I_b==1), x_1>0, x_1=x_1-1)$
$t_{11}=(S_2, S_3, (O_r==1)\wedge(I_a==I_a')\wedge(I_b==0), x_1>0, x_1=x_1-1)$
$t_{12}=(S_3, S_3, (O_r==1)\wedge(I_a==I_a')\wedge(I_b==0), -, -)$
$t_{13}=(S_3, S_1, (O_r==1)\wedge(I_a==I_a')\wedge(I_b==1), -, -)$

Figure 4.1: A CEFSM example.

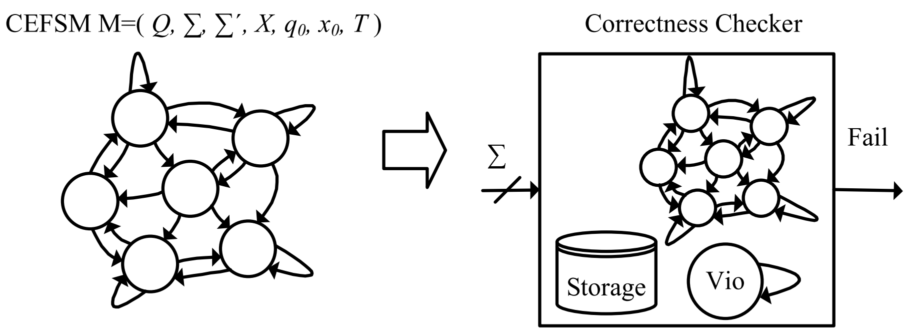


Figure 4.2: CEFSM to correctness checker.

shown in Fig. 4.2, the checker embeds a CEFSM that has similar states and transitions to the given CEFSM model. Furthermore, an extra state, "*Vio*", is added into the embedded CEFSM for indicating the violation status. The checker takes all interface signals (Σ) as its input, and produces only one output signal, "*Fail*", representing the checking result. The value of *Fail* is associated with the current state. While the embedded CEFSM moves to the state *Vio*, *Fail* is asserted to 1. Otherwise, *Fail* remains 0 indicating no violations. There is also one dedicated storage element for each interface signal to keep its value in the previous cycle. During simulation, state transitions are performed based on the evaluation of the transition conditions. While the interface signal values obey the specification, the embedded CEFSM makes the corresponding transition to reach a valid state. On the contrary, if no single transition condition can be satisfied after evaluation, the machine moves to *Vio*. CEFSM. The violation alert can be observed through the signal *Fail* which is only asserted at *Vio*.

Now we illustrate the correctness checking process with the corresponding checker translated from the CEFSM in Fig. 4.1. The simulation data is shown as waveforms in

Fig. 4.3. At the cycle *C1*, initially the current state is S_0 and the current value of x_1 is 0. Among all transitions in $T_{next}(S_0)$, only the transition condition of t_3 is satisfied. As a result, the checker moves to S_1 and sets the value of x_1 to 3 according to a_{t_3}. By repeating this kind of iteration, the state changes in the order "$S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_1 \rightarrow S_3$". Since the state transitions are all kept within the valid states during the first five cycles, the signal *Fail* holds low. However, at *C5*, no single transition whose condition can be satisfied so that the checker moves to *Vio* and then asserts *Fail*. It means a certain protocol violation occurs and the current simulation task should be stopped and the debugging process should then begin.

4.3 CEFSM-based Stimulus Generator

In this section, we introduce the CEFSM-based stimulus generation scheme. The primary function of the stimulus generator is to assign valid values to I_{DUV} according to the observed values of O_{DUV}. Before describing the details, we use the example in Fig. 4.3 to give the idea of our stimulus generation. In this example, I_a, I_b, and I_d belong to I_{DUV}, and O_r belongs to O_{DUV}. That is, the valuses of I_a, I_b, and I_d are produced by the stimulus generator while the value of O_r is driven by the DUV. Consider the state transition of the corresponding CEFSM at *C2*. After evaluating the transition conditions with $O_r = 0$, $I_b = 1$, and $x_1 = 3$, the CEFSM takes a transition t_6 to move to S_2 and retains the value of x_1 simultaneously. The updated variable value is exactly the evaluation value for the state

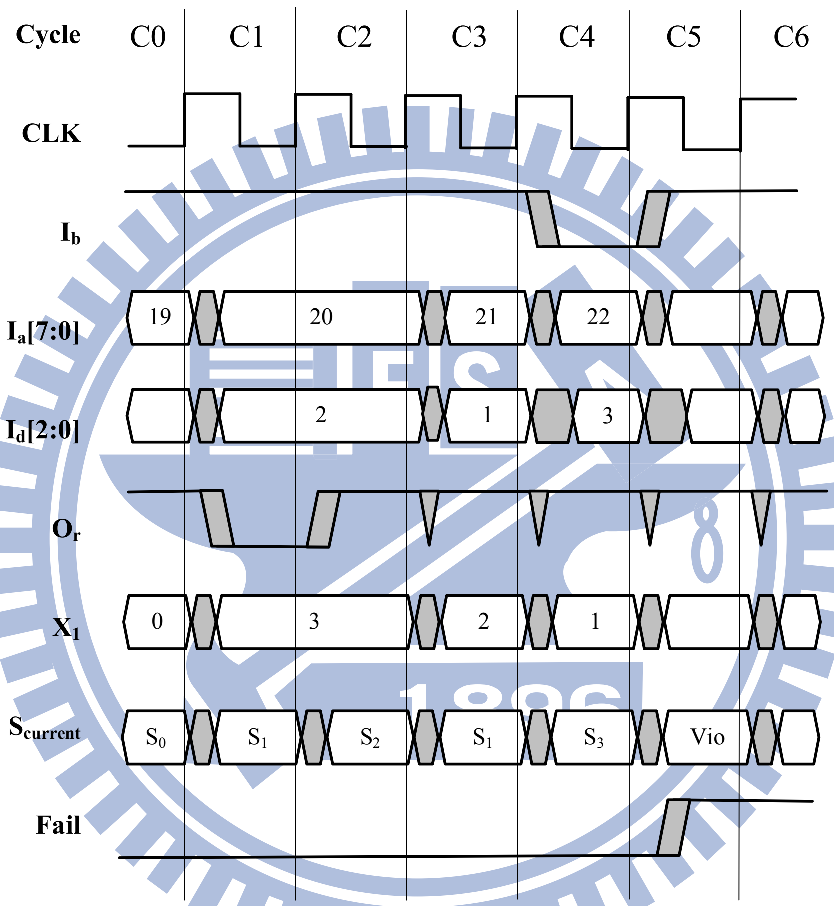


Figure 4.3: A simulation example.

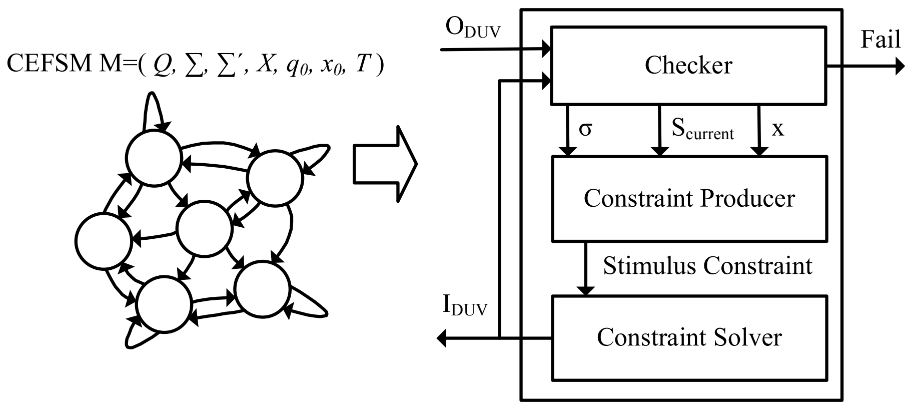


Figure 4.4: CEFSM to stimulus generator.

transition at *C3*. Since the updated variable value can determine the results of those predicate functions of transitions in T_{next}, some impossible next transitions can be eliminated in advance according to the variable value. The set of remaining transitions is the *next transition candidate set (NTCS)*. In this example, $T_{next}(S_2)$ is $\{t_8, t_9, t_{10}, t_{11}\}$ while the NTCS of S_2 at *C2* is $\{t_8, t_{10}, t_{11}\}$. Identifying the NTCS of the current state is the preprocess of the stimulus generation in our approach. Subsequently, the generator must assign values to I_{DUV} at *C2* for interacting with the DUV. These values become the evaluation values of I_a, I_b, and I_d for state transition at *C3*. Note that arbitrary stimulus assignments may cause certain protocol violations. For example, if we make the assignments I_a = 19 at *C2*, the embedded CEFSM can not find any valid transition at *C3*, and thus a protocol violation occurs. In other words, to generate a valid stimulus, the relation functions in NTCS must be taken into consideration.

The CEFSM-based stimulus generator consists of a CEFSM-based checker, a constraint producer, and a constraint solver as shown in Fig. 4.4. In the first stage, the checker performs certain state transition and updates the values of all variables. In addition, the checker passes necessary information to the constraint producer. The information includes the current signals' values (σ), the current state ($S_{current}$), and the current variables' values (x).

In the second stage, the constraint producer generates the constraint indicating a valid solution space for the stimulus generation. The constraint can be obtained by applying three operations over the relation functions of transitions in $T_{next}(S_{current})$ as shown in Fig. 4.5. Since the values of O_{DUV} are determined by the DUV, the existential quantification operations are first performed to make those relation functions of transitions in $T_{next}(S_{current})$ independent of O_{DUV}. Suppose the relation function of the transition t is shown as Equation (4.1). In the beginning, the input of the relation function r_t is the union of Σ' and Σ_t. Illustrated in Equation (4.2), the *simplified relation function (SR)* of the transition t is the result after performing the existential quantification operation of r_t with respect to O_{DUV}. Since the dependence of O_{DUV} is eliminated, the input of SR_t reduces to the union of Σ' and $\Sigma_t \cap I_{DUV}$ (note that $\Sigma_t \cap I_{DUV}$ = Σ_t - O_{DUV}). The second operation is the constraint evaluation. This operation applies σ and x to the SRs and the predicate functions of the transitions in $T_{next}(S_{current})$, respectively. On the one hand, since the SRs regard the next state transition, σ are treated as the previous values at the next state transition. The evaluation of SR can further simplify the original function into the *stimulus constraint function (SC)* as shown in Equation (4.3). The remaining input in

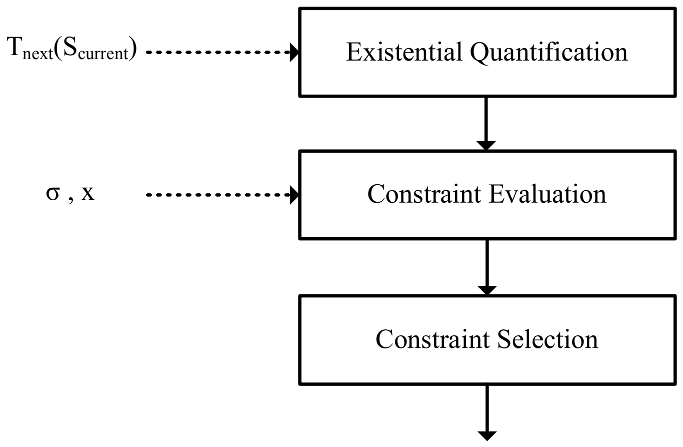


Figure 4.5: The operation flow of a constraint producer.

SC_t further reduces to the set of $\Sigma_t \cap I_{DUV}$, which is only a subset of I_{DUV}. The SC_t is exactly the constraint for I_{DUV} when the transition t is selected as the next transition. On the other hand, the evaluation of the predicate functions can discard some invalid next transitions and obtain the NTCS.

$$r_t(\Sigma', \Sigma_t) = \bigwedge_{for\ each\ b_i\ \in\ \Sigma_t} (b_i == f_{t_i}(\Sigma')) \tag{4.1}$$

$$\begin{aligned} SR_t(\Sigma', \Sigma_t \cap I_{DUV}) &= \exists O_{DUV}\ (r_t(\Sigma', \Sigma_t)) \\ &= \bigwedge_{for\ each\ b_i\ \in\ \Sigma_t \cap I_{DUV}} (b_i == f_{t_i}(\Sigma')) \end{aligned} \tag{4.2}$$

$$\begin{aligned} SC_t(\Sigma_t \cap I_{DUV}) &= SR_t(\sigma, \Sigma_t \cap I_{DUV}) \\ &= \bigwedge_{for\ each\ b_i\ \in\ \Sigma_t \cap I_{DUV}} (b_i == f_{t_i}(\sigma)) \end{aligned} \tag{4.3}$$

Actually, the entire solution space for the current generated stimulus is the conjunction of SCs in the NTCS of the current state: $\bigvee_{t_i \in NTCS} SC_{t_i}$. Intuitively, we could directly solve this constraint to obtain a valid stimulus. However, in our approach, only one SC from the NTCS is picked and passed to the constraint solver. This strategy comes with the following two benefits:

- Simplify the constraint solving process.
- Enable the transition biasing (discuss later).

In the last stage, the constraint solver receives an SC from the constraint producer and then generates a valid stimulus accordingly. As described, the SC is in the form:

$$(I_{DUV_1} == k_1) \wedge (I_{DUV_2} == k_2) \wedge \ldots \wedge (I_{DUV_N} == k_N),$$

$$k_i\ maps\ to\ a\ specific\ value\ within\ the\ value\ set\ of\ I_{DUV_i}$$

As a result, the constraint solver first performs a series of assignments, including "$(I_{DUV_1} = k_1)$", "$(I_{DUV_2} = k_2)$", $\ldots$, and "$(I_{DUV_N} = k_N)$". This process assigns a set of valid values for the constrained signals. After these assignments, remaining unconstrained signals in I_{DUV} can be assigned with any values in their own value set by the constraint solver.

Like the GEFSM-based stimulus generator, the CEFSM-based stimulus generator also support various stimulus biasing options. Remind that the constraint producer only selects a single stimulus constraint instead of the union of all stimulus constraints derived from the NTCS. For transition-level biasing, each transition t_i is given an individual weight w_{t_i}. Then, the probability of selecting a candidate constraint SC_{t_i} is defined as:

$$P_{SC_{t_i}} = \begin{cases} \frac{w_{t_i}}{\sum_{t_j \in NTCS} w_{t_j}}, & t_i \in \text{NTCS} \\ 0, & \text{otherwise} \end{cases}$$

A candidate constraint with a larger weight has a higher probability to be chosen. In other words, the transition trend can be biased by this mechanism. Moreover, if previous simulation results do not exercise certain states or transitions, the related transition weights can be increased accordingly to raise their appearance probabilities. We can also define a meaningful transaction in terms of a sequence of transitions and then bias all these transitions simultaneously.

Note that even if a specific SC_{t_i} is selected, it is not guaranteed that the corresponding transition t_i is exactly the next transition. This is because the solution space of the selected SC may overlap with the solution spaces of other SCs. An example shown in Fig. 4.6 demonstrates two kind of relations of solution spaces shown in Fig. 4.3. SP_i denotes the solution space of SC_{t_i}. Fig. 4.6(a) shows that the overlapping relation for solution spaces of SCs at *C1*. SP_4 is a subset of SP_6. That is, even if SC_{t_4} is selected for stimulus generation, the next transition could be t_6 instead of t_4. Another example is shown in Fig. 4.6(b), the two non-overlapping solution spaces mean that the selected SC can directly determine the next transition. Obviously, in either case, the transition-level

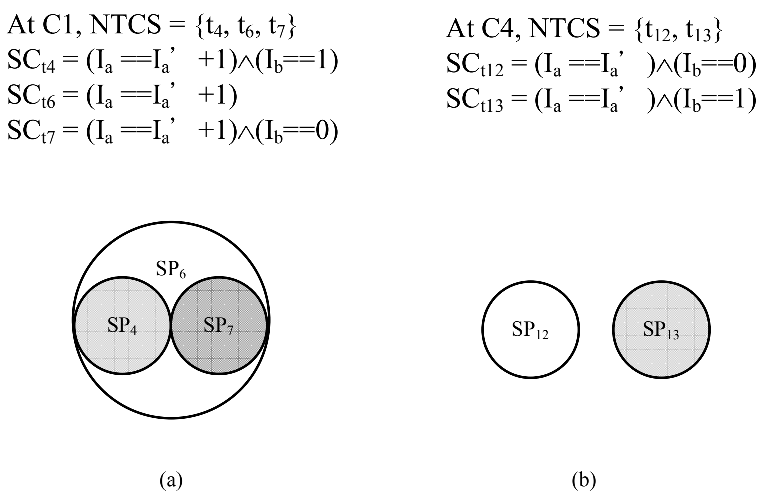

At C1, NTCS = {t_4, t_6, t_7}
$SC_{t4} = (I_a == I_a' + 1) \wedge (I_b == 1)$
$SC_{t6} = (I_a == I_a' + 1)$
$SC_{t7} = (I_a == I_a' + 1) \wedge (I_b == 0)$
At C4, NTCS = {t_{12}, t_{13}}
$SC_{t12} = (I_a == I_a') \wedge (I_b == 0)$
$SC_{t13} = (I_a == I_a') \wedge (I_b == 1)$
SP_6
SP_4
SP_7
SP_{12}
SP_{13}
(a)
(b)

Figure 4.6: Two different kinds of relations.

biasing can increase the probability of exercising certain transitions.

4.4 Automatic Translation

We implement a translator that can read in a CEFSM model and then automatically produce the stimulus generator and correctness checker. To meet specific simulation requirements, these components can be translated into either a high-level language or a native HDL. That is, it is possible to use only a native HDL simulator to simulate all the simulation components and the DUV. This can save a lot of simulation time compared to those approaches using PLI.

Furthermore, our translator is capable of generating simulation components in synthesizable HDL on one condition – only synthesizable operators are allowed in the transition conditions of the given CEFSM. In our experiences, the set of synthesizable operators are generally sufficient to model most interface protocols. Fig. 4.7 illustrates the block diagram of the proposed CEFSM-based stimulus generator. The upper part is the embedded correctness checker. The checker consists of several Flip-Flops (FFs) and a transition evaluator which can be implemented as a combinational circuit. The architecture of the correctness checker is similar to the traditional FSM with extended variables. Unlike the traditional FSM that makes a state transition only depending on the current state and current input values, in our approach, the previous input values and current variable values can also affect the state transition. The transition evaluator implements the transition conditions and action functions to determine the next state and update the values of the variables.

The lower part of Fig. 4.7 shows the elements for stimulus generation. The constraint evaluator and selector accomplish the flow in Fig. 4.5. Note that the existential quantification is done by our translator in the preprocess so that the stimulus generator does not need to do this operation during simulation. The constraint evaluator can determine the NTCS of the current state according to the predicate functions in $T_{next}(S_{current})$. Since the NTCS may contain more than one transition, a constraint selector is required. The easiest way to build a constraint selector is to use the round-robin method. If the transition-level biasing is desired, the selector with a weighted random selection engine [40] can be adopted alternatively. The weighted random selection engine can randomly choose one item from the candidates based on the candidates' weights, and thus realize the transition-level biasing.

According to the selected constraint, the constraint solver can assign valid values to

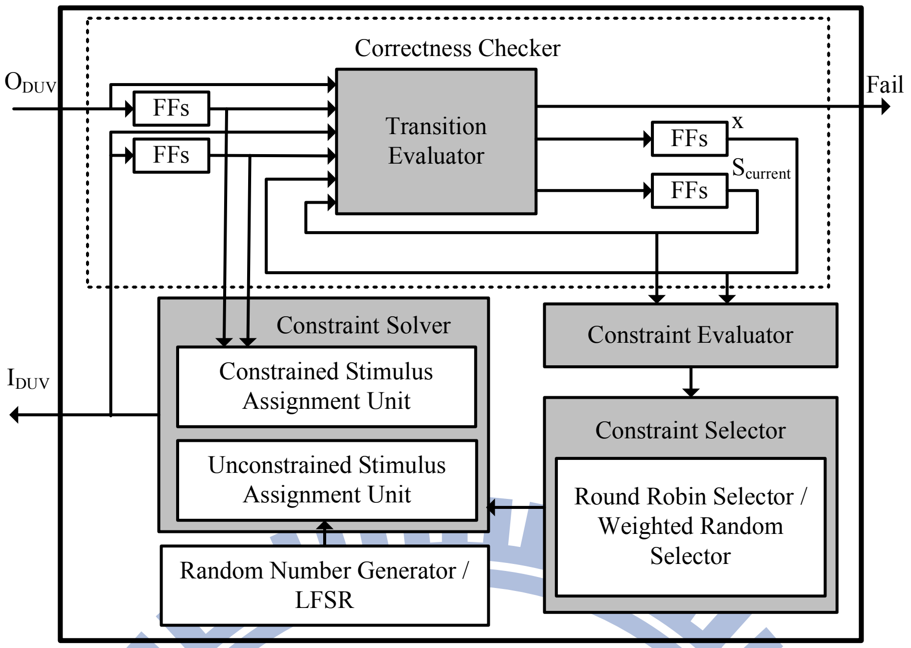

Correctness Checker
O_DUV
FFs
FFs
Transition Evaluator
Fail
FFs
x
S_current
FFs
I_DUV
Constraint Solver
Constrained Stimulus Assignment Unit
Unconstrained Stimulus Assignment Unit
Random Number Generator / LFSR
Constraint Evaluator
Constraint Selector
Round Robin Selector / Weighted Random Selector

Figure 4.7: The block diagram of the proposed CEFSM-based stimulus generator.

I_{DUV}. This solver contains two units for stimulus assignment. One unit makes a series of stimulus assignments with constrainted values, and the other one makes a series of stimulus assignments with random values. Typically, random numbers can be obtained from a software call provided by most HDL simulators. For hardware implementation, an LFSR circuit can be adopted to produce random numbers. Furthermore, if the word-level biasing is required, weighted random number generators can replace the pure random number generators. All the elements in our stimulus generator can be implemented with either behavioral or synthesizable HDL easily. Hence, our approach enables the truly hardware-based stimulus generator and correctness checker that can be realized in hardware acceleration environments.

4.5 Coverage Metrics

4.5.1 The Basic Coverage Metrics

Generally, various coverage metrics are used to evaluate the simulation completeness. In our approach, since the EFSM is used to represent the interface protocol specification, some basic coverage metrics can be easily derived from the EFSM model:

- **State coverage:** the coverage of all states in the EFSM.
- **Transition coverage:** the coverage of all transitions in the EFSM.
- **Transition-pair coverage:** the coverage of feasible transition-pairs in the EFSM.

Intuitively, these three coverage metrics can be observed by adding flags to the states and transitions of the embedded EFSM of the correctness checker. In other words, the EFSM-based correctness checker can also provide basic coverage analysis.

For interface protocol verification, how many transaction types are examined should be the major concern. However, a transaction specified in an interface protocol specification usually contains a series of state transitions. Obviously, basic state or transition coverage can not directly represent the transaction coverage. Many meaningful transaction types may not be exercised yet even if all states and transitions are covered during simulation. For this reason, we propose a transaction-level functional coverage methodology to enhance the coverage measurement.

4.5.2 The Transaction-level Functional Coverage

In order to provide a method for specifying transactions simply, we develop a transaction description language, SOL, mainly based on the Property Specification Language (PSL) [45]. Because PSL provides a richer set of expressive and readable language constructs than typical regular-expression-based approaches do, SOL adopts most PSL constructs used to describe temporal sequences. The syntax of SOL is based on the following principles:

- Since a transaction is defined as a specific sequence of state transitions, states are used as basic elements to describe sequences.
- Extra signals can be included in additional to the states while defining a transaction.
- A sequence can be defined once as a named sequence and then be reused later. The assignment operator is used to define a named sequence. The left-hand-side of the assignment operator becomes a synonym for the sequence on the right-hand-side.
- Sequence name is enclosed in braces when referred.
- A sequence set comprises one or more sequences. Sequences are enclosed in angle brackets and separated by commas.

The syntax of SOL is briefly introduced below. We use the CEFSM shown in Fig. 4.1 as an example again to demonstrate the operators in SOL.

1. Concatenation (;): Two sequences can be concatenated into one by the concatenation operator.

   Example 1: In Fig. 4.1, T_1 is a transaction with the state transitions that starts from S_1, then moves through S_2, S_3, and ends at S_1.

   $T_1 : S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_1$

   SOL $T_1 = \{S_1; S_2; S_3; S_1\};$

2. Extra signal qualification (" "): Extra signals can be qualified while making a state transition. The expression built from the extra signals should be enclosed in double quotes.

   Example 2: In Fig. 4.1, T_2 is another transaction with the same state transitions sequence as T_1 while the value of the extra signal x_1 must be 2 when moving from S_1 to S_2.

   $T_2 : S_1 \overset{x_1==2}{\longrightarrow} S_2 \rightarrow S_3 \rightarrow S_1$

   SOL $T_2 = \{S_1$"x_1==2"; $S_2; S_3; S_1\};$

3. Repetition ([ ]): The repetition operators are used to describe repeated concatenations of a sequence. There are three types of the repetition operators: consecutive repetition ([*]), non-consecutive repetition ([=]), and goto repetition ([$\rightarrow$]).

   (a) Consecutive repetition ([*]):

   Example 3: In Fig. 4.1, T_3 is a transaction with the state transitions that starts from S_1, moves to S_2, and stays at S_2 for three consecutive cycles, then ends at S_1.

$T_3 : S_1 \rightarrow S_2 \rightarrow S_2 \rightarrow S_2 \rightarrow S_1$

SOL $T_3 = \{S_1; S_2[*3]; S_1\};$

Example 4: In Fig. 4.1, T_4 is a transaction with the state transitions that starts from S_1, moves to S_2, and stays at S_2 for one to five consecutive cycles, then ends at S_1.

$T_4 : S_1 \rightarrow S_2\ (1 \sim 5\ cycles) \rightarrow S_1$

SOL $T_4 = \{S_1; S_2[*1:5]; S_1\};$

(b) Non-consecutive repetition ([=]):

Example 5: In Fig. 4.1, T_5 is a transaction with the state transitions that starts from S_1, and then visits S_2 three times. The visits of S_2 need not to be in consecutive cycles. In addition, T_5 holds after the third S_2 is visited and still holds before the forth S_2 appears.

$$\overbrace{S_1 \rightarrow \ldots \rightarrow S_2 \rightarrow \ldots \rightarrow S_2 \rightarrow \ldots \rightarrow S_2 \rightarrow \ldots}^{T_5} \rightarrow S_2 \rightarrow \ldots$$

SOL $T_5 = \{S_1; S_2[=3]\};$

(c) Goto repetition ([$\rightarrow$]):

Example 6: In Fig. 4.1, similar to T_5, T_6 is also a transaction with the state transitions that starts from S_1, and then moves to S_2 three times (can be non-consecutive). In addition, T_6 holds only at the cycle in which the third S_2 is visited.

$$\overbrace{S_1 \rightarrow \ldots \rightarrow S_2 \rightarrow \ldots \rightarrow S_2 \rightarrow \ldots \rightarrow S_2}^{T_6} \rightarrow \ldots \rightarrow S_2 \rightarrow \ldots$$

SOL $T_6 = \{S_1; S_2[\rightarrow 3]\};$

4. Sequence AND (&&): A transaction which combines two sequences with the sequence AND operator holds only if both sequences hold and complete at the same cycle.

   Example 7: In Fig. 4.1, similar to T_6, T_7 is also a transaction with the state transitions that starts from S_1, and then visits S2 three times (can be non-consecutive). However, S_3 is not allowed showing up in the sequence T_7 strictly.

   $T_7 : S_1 \rightarrow \ldots (!S_3) \rightarrow S_2 \rightarrow \ldots (!S_3) \rightarrow S_2 \rightarrow \ldots (!S_3) \rightarrow S_2$

   SOL $T_7 = \{S_1; \{S_3[=0]\}\&\&S_2[\rightarrow 3]\};$

5. Sequence OR (|): A transaction which combines two sequences with the sequence OR operator holds if one of the two alternative sequences holds.

   Example 8: In Fig. 4.1, T_8 is a transaction shown below:

   $T_8 : S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_1$ OR $S_1 \rightarrow S_2 \rightarrow S_2 \rightarrow S_2 \rightarrow S_1$

   SOL $T_8 = \{\{S_1; S_2; S_3; S_1\}|\{S_1; S_2[*3]; S_1\}\};$

   Note that above two sequences are previously defined as T_1 and T_3. Hence, T_8 can also be defined in terms of these named sequences.

   $T_8 = \{\{T_1\}|\{T_3\}\};$

6. Sequence fusion (:): Similar to the concatenation operator, a sequence fusion operator concatenates two sequences overlapping by one cycle.

   Example 9: In Fig. 4.1, T_9 is a transaction shown below:

   $T_9 : S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_1 \rightarrow S_2 \rightarrow S_2 \rightarrow S_2 \rightarrow S_1$

   SOL $T_9 = \{S_1; S_2; S_3; S_1; S_2[*3]; S_1\};$

   T_9 can also be treated as two sequences that overlap each other for one cycle as shown below:

   SOL $T_9 = \{\{S_1; S_2; S_3; S_1\} : \{S_1; S_2[*3]; S_1\}\};$

   Again, T_9 can also be defined in terms of T_1 and T_3.

   $T_9 = \{\{T_1\} : \{T_3\}\};$

7. Sequence set cross (**): A sequence set cross operator is used to represent a set of back-to-back consecutive transactions.

   Example 10: Assume the following 8 transactions are interested:

   $\{\{T_1\} : \{T_3\} : \{T_8\}\}; \{\{T_2\} : \{T_3\} : \{T_8\}\};$

   $\{\{T_1\} : \{T_4\} : \{T_8\}\}; \{\{T_2\} : \{T_4\} : \{T_8\}\};$

   $\{\{T_1\} : \{T_3\} : \{T_9\}\}; \{\{T_2\} : \{T_3\} : \{T_9\}\};$

   $\{\{T_1\} : \{T_4\} : \{T_9\}\}; \{\{T_2\} : \{T_4\} : \{T_9\}\};$

   The following expression utilizing the sequence set cross operator provides a much more elegant but equivalent representation for the set of 8 interested transactions.

SOL $T_{10} = \langle \{T_1\}, \{T_2\} \rangle$**$\langle \{T_3\}, \{T_4\} \rangle$ **$\langle \{T_8\}, \{T_9\} \rangle$;

SOL provides an efficient methodology in transaction description for state-based specifications. As mentioned, SOL is a PSL-like language. Except for the "sequence set cross" and "extra signal qualification" operators, other operators in SOL can find their corresponding counterparts in PSL. Hence people who are familiar with PSL can easily use SOL with virtually no extra learning effort. The fundamental conceptual difference between PSL and SOL is that SOL uses states as the atomic elements when defining a transaction. This method can raise the level of abstraction as well as encapsulate the details of the low-level signals. Hence, verification engineers can put more emphasis on the functionality at higher abstract level. Besides, the two new operators can further bring more conveniences at EFSM level. Using "sequence set cross" operations, we can fold lots of back-to-back transactions in a concise manner. The "extra signal qualification" is useful while the transactions is related to some internal variables of EFSM or abstracted signals.

4.6 Experimental Results

We choose the AMBA AHB and WISHBONE protocols to demonstrate our methodology. Experiments are conducted over those designs in Table 3.3. The experimental environment is built as shown in Fig. 2.2. We model the required protocol specification with a CEFSM. The resultant CEFSM of the WISHBONE contains 6 states and 23 transitions while AHB's requires 7 states and 69 transitions. The simulation kit, including the stimulus generator, the correctness checker, and the coverage analyzer, are directly translated from the CEFSM model. With the generated simulation kit, the simulation-based verification process is fully automated. Next, we report detailed experimental results to demonstrate the important features of our methodology.

4.6.1 Coverage Comparison

Here three coverage results (state coverage, state transition coverage, and transaction coverage) are compared for these designs, respectively.

Case I.

We first define the interested transactions of the two protocols as followings:

- WISHBONE: 6 basic read and write transactions, i.e., { SingleRead }, { SingleWrite }, { BlockRead }, { BlockWrite }, { FourBeatBlockRead }, { FourBeatBlockWrite }.

- AHB: 10 basic read and write transactions, i.e., { IncrBeatRead }, { IncrBeatWrite }, { OneBeatRead }, { OneBeatWrite }, { FourBeatRead }, { FourBeatWrite }, { EightBeatRead }, { EightBeatWrite }, { SixteenBeatRead }, { SixteenBeatWrite }.

Meanwhile, the stimulus generator is configured to evenly pick the next transition from the NTCS and assign the unconstrained values in purely random way. That is, no bias strategy is used.

The comparison results are shown in Table 4.1. For those WISHBONE-compliant designs, few cycles are required to reach 100% state/transition coverage. However, while the state/transition coverage are satisfied, the basic transactions are not fully exercised. For the AHB-compliant design RGB2YCrCb, it takes 8/2455/1040 cycles to reach 100% state/transition/transaction coverage. As the state coverage reaches 100%, the transaction coverage is only 10%. In this case, the interested transactions are very simple so that more simulation time is required to achieve 100% transition coverage than 100% transaction coverage. The results from the other two AMBA-compliant designs basically tell the similar story.

Case II.

Besides 10 basic transactions, we make the interested transactions of AMBA designs more complex by adding more transactions:

- 4 transactions with BUSY (i.e., { IncrBeatWithBUSY }, { FourBeatWithBUSY }, etc.).
- 25 consecutive transactions (i.e., ⟨{ IncrBeat }, { OneBeat }, { FourBeat }, { EightBeat }, { SixteenBeat }⟩ **⟨{ IncrBeat }, { OneBeat }, { FourBeat }, { EightBeat

Table 4.1: Coverage comparisons for Case I

Protocol	**Design**	**Coverage type**	**# of cycles to reach 100%**	**Transaction coverage (%)**	
WISHBONE	PTC	State	12	17	(1/6)
		Transition	48	50	(3/6)
		Transaction	245	100	(6/6)
	AC97	State	10	17	(1/6)
		Transition	155	67	(4/6)
		Transaction	2480	100	(6/6)
	SPI	State	21	17	(1/6)
		Transition	229	83	(5/6)
		Transaction	414	100	(6/6)
AHB	RGB2YCrCb	State	8	10	(1/10)
		Transition	2455	100	(10/10)
		Transaction	1040	100	(10/10)
	Convolution	State	84	20	(2/10)
		Transition	3319	100	(10/10)
		Transaction	1427	100	(10/10)
	MAC	State	30	10	(1/10)
		Transition	1823	100	(10/10)
		Transaction	923	100	(10/10)

}, { SixteenBeat }⟩).

- 2 transactions interleaved with BUSY and SEQ transfers (i.e., { FourBeatInter } = { S_{NONSEQ}; S_{BUSY}; S_{SEQ} ; S_{BUSY}; S_{SEQ}; S_{BUSY}; S_{SEQ} }, { EightBeatInter }).

Again, no bias strategy is used here. The comparison results are shown in Table 4.2. For the design RGB2YCrCb, it still takes 8/2455 cycles to reach 100% state/transition coverage. But it takes 732381 cycles to reach 100% transaction coverage. As the state/transition coverage reach 100%, the transaction coverage is only 5%/85%. It is shown that the transaction coverage is much lower than that in Case I as the other two coverage metrics reach 100%.

We get some conclusions from the above 2 cases. While the set of interested transactions becomes larger and more complex, it needs significantly (non-linearly) longer

Table 4.2: Coverage comparisons for Case II

Design	**Coverage type**	**# of cycles to reach 100%**	**Transaction coverage (%)**	
RGB2YCrCb	State	8	5	(2/41)
	Transition	2455	85	(35/41)
	Transaction	732381	100	(41/41)

simulation time to reach 100% transaction coverage. Moreover, even the state/transition coverage reach 100%, the transaction coverage can still be very low. The situation is getting worse when more complicated transactions are concerned. It means that even a stimulus set developed to reach 100% state/transition coverage hardly provides a satisfied functional coverage for real industrial designs. Experimental results show that the classical coverage metrics are not capable of providing enough verification quality.

4.6.2 Stimulus Biasing

After analyzing the coverage report of Case II, it is found that the major reason why so many cycles are required to reach 100% transaction coverage is the seldom occurrence of the transaction { EightBeatInter }. Hence, it is possible to reduce the simulation time by biasing the stimulus generator but still remain the same functional coverage quality. The biasing settings and results are shown in Table 4.3 and Table 4.4, respectively.

In $bias_1$, in which the word-level biasing is used, the weight of 8-beat burst is 10 times to other burst types because the transaction { EightBeatInter } only occurs in an 8-beat burst transfer. This biasing indeed decreases the simulation time to 54295 cycles, which is only 7.4% of the original one. In $bias_2$, in which the transaction-level biasing is applied,

Table 4.3: Biasing settings

	Word-level biasing on HBURST					**Transaction-level biasing**	
Biasing	**SINGLE**	**INCR**	**4-Beat**	**8-Beat**	**16-Beat**	**Every transition leaving or entering the busy state**	**others**
$no\ bias$	1	1	1	1	1	1	1
$bias_1$	1	1	1	10	1	1	1
$bias_2$	1	1	1	1	1	10	1
$bias_3$	1	1	1	10	1	10	1

Table 4.4: Biasing results for Design RGB2YCrCb in Case II

Design	**Bias**	**# of cycles to reach 100% transaction coverage**	**Factor**
RGB2YCrCb	$no\ biasing$	732381	1
	$bias_1$	54295	0.074
	$bias_2$	33861	0.046
	$bias_3$	16551	0.027

the weights of transitions that enter or leave the busy state are intuitively increased. This biasing can reach 100% transaction coverage in only 33861 cycles. Applying both word-level and transaction-level biasing in $bias_3$, the simulation time can be further reduced to 16551 cycles, which is only 2.7% of the original one. The results show that the coverage information can help bias the stimulus generator to create more effective stimuli and help verify the DUV in much shorter time. This technique is extremely useful while developing a regression verification environment in which the compact and effective stimulus suites are crucial to minimize the required simulation time. That is, the proposed methodology can increase the efficiency of the regression verification process.

4.6.3 Error Detection

To verify the effectiveness of the correctness checker, we intentionally inject protocol-related errors into designs. The experimental results show that the checker is fully capable of capturing these kinds of design errors. Once an error is detected, the error-occuring state/transion/transaction are reported so that it would be very helpful to reason and locate where the root error source is.

4.6.4 Synthesis Results

As mentioned, our simulation component can be mapped to real hardware through a commercial logic synthesizer. We use Synoposys Design Vision as the synthesizer under UMC 0.18 technology. For WISHBONE protocol, the synthesizer reports that 0.7K and 2.0K NAND2-equivalent gates are required to implement a correctness checker and a stimulus generator with an embedded checker, respectively. For AMBA AHB protocol, the synthesizer reports that 3.0K and 6.8K NAND2-equivalent gates are required to implement a correctness checker and a stimulus generator with an embedded checker, respectively. This result clearly shows that the proposed correctness checker and stimulus generator can be easily and cost-effectively integrated into a high-performance emulator-based verification flow.

4.7 Summary

In this chapter, we introduce the CEFSM-based unified framework. A translator is built to automatically translate the given CEFSM model into a complete simulation kit which is more flexible to apply in various simulation environments. In addition, we also propose a state-oriented language SOL, which can help develop the transaction-level functional coverage metric to achieve even better verification quality. This coverage metric can provide useful information for further stimulus biasing. The experimental results demonstrate that the proposed methodology can indeed improve the verification quality as well as speed up the verification process.

Chapter 5

Conclusions and Future Works

5.1 Conclusions

Designing a reusable IP component with a standard interface protocol is a trend in the SOC era. It is very important to guarantee that the integrated IP conforms to the specific interface protocol. Therefore, how to correctly and efficiently perform the interface compliance verification becomes a big issue for SOC integration.

In this dissertation, we propose a unified methodology for interface compliance verification. We formulate an interface protocol specification with an EFSM model. Two different kinds of EFSM models are developed for different specification viewpoints. The EFSM model can then be translated into a correctness checker, a stimulus generator, and a coverage analyzer as the essential simulation components. These components can also be implemented in synthesizable HDL to enable hardware acceleration. For steering the simulation direction, the generated stimuli can be guided through transition-, transaction-, and word-level biasing to provide better stimulus controllability. In addition, we also propose a state-oriented language SOL, which can help develop the transaction-level functional coverage metric to achieve even better verification quality. This coverage metric can provide useful information for further stimulus biasing. The experimental results demonstrate that the proposed methodology can indeed improve the verification quality as well as speed up the verification process. Therefore, we believe the proposed EFSM-based verification flow is an efficient and effective solution for interface compliance verification.

5.2 Future Works

There are some improvements could be done in the future:

An automatic biasing tuner: Though our methodology provides multiple stimulus biasing options, the stimulus biasing setting is still based on users' knowledge. Obviously, it would be better to mathematically link the stimulus biasing options for a specific coverage metric, especially transaction-level coverage, to automate the biasing process. In addition, our work only focuses on the behaviors of interface signals, that is, the internal architecture of the DUV is treated as a black box. If the internal information of DUV is collected and analyzed during simulation, it can further guide the stimulus generation and improve the debugging process.

Modeling ability: Though the proposed EFSM models are appropriate for common interface protocol modeling, they still have limitations in some characterization. For example, current EFSM models can not directly describe a protocol with multiple clock signals. More powerful structure may be added into our EFSM models to enhance the modeling ability.

Specification debugging tool: While the EFSM modeling style brings a more powerful describing ability, its structure is more complex in checking if there is any consistence problem in the EFSM model. The transition functions of the EFSM model should be carefully used to avoid the inconsistences between them which may make certain states unreachable. Some previous researches [46] [47] focus on this issue but they can not directly apply for our modified EFSM models. If we can develop algorithms for debugging the proposed EFSM models, it will be helpful in the protocol modeling phase.

Bibliography

[1] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Transactions on Programming Languages and Systems*, pp. 244–263, April 1986.

[2] I. Beer, S. Ben-David, C. Eisner, Y. Engel, R. Gewitzman, and A. Landver, "Establishing PCI compliance using formal verification: A case study," *Proceedings of the 14th International Phoenix Conference on Computation and Communications*, pp. 373–377, March 1995.

[3] S. Campos, E. M. Clarke, W. R. Marrero, and M. Minea, "Verifying the performance of the PCI local bus using symbolic techniques," *Proceedings of the 1995 International Conference on Computer Design: VLSI in Computers and Processors*, pp. 72–78, January 1995.

[4] K. Kaufmann, A. Martin, and C. Pixley, "Design constraints in symbolic model checking," *Proceedings of the 10th International Conference on Computer Aided*

Verification, pp. 477–487, June 1998.

[5] P. Chauhan, E. M. Clarke, Y. Lu, and D. Wang, “Verifying ip-core based system-on-chip designs,” *Proceedings of the 12th Annual IEEE International ASIC/SOC Conference*, pp. 27–31, September 1999.

[6] A. Roychoudhury, T. Mitra, and S. R. Karri, “Using formal techniques to debug the AMBA system-on-chip bus protocol,” *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pp. 1530–1591, March 2003.

[7] K. L. McMillann, “Symbolic model checking,” *Kluwer Academic Publishers*, 1993.

[8] K. Shimizu, D. L. Dill, and A. J. Hu, “Monitor-based formal specification of PCI,” *Proceedings of the 3th International Conference on Formal Methods in Computer-Aided Design*, pp. 335–353, November 2000.

[9] H.-M. Lin, C.-C. Yen, C.-H. Shih, and J.-Y. Jou, “On compliance test of on-chip bus for SOC,” *Proceedings of the Asia and South Pacific Design Automation Conference*, pp. 328–333, January 2004.

[10] M. T. Oliviera and A. J. Hu, “High level specification and automatic generation of IP interface monitors,” *Proceedings of the 39th Design Automation Conference*, pp. 129–134, June 2002.

[11] A. J. Hu, J. Casus, and J. Yang, "Efficient generation of monitor circuits for GSTE assertion graphs," *Proceedings of the 2003 IEEE/ACM International Conference on Computer-Aided Design*, pp. 154–159, November 2003.

[12] T. Larrabee, "Test pattern generation using boolean satisfiability," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 4–15, January 1992.

[13] J. Yuan, K. Shultz, C. Pixley, H. Miller, , and A. Aziz, "Modeling design constraints and biasing in simulation using BDDs," *Proceedings of the 1999 IEEE/ACM International Conference on Computer-Aided Design*, pp. 584–589, November 1999.

[14] K. Shimizu and D. L. Dill, "Deriving a simulation input generator and a coverage metric from a formal specification," *Proceedings of the 39th Design Automation Conference*, pp. 801–806, June 2002.

[15] J. Yuan, A. Aziz, K. Albin, and C. Pixley, "Simplifying boolean constraint solving for random simulation-vector generation," *Proceedings of the 2002 IEEE/ACM International Conference on Computer-Aided Design*, pp. 123–127, November 2002.

[16] J. Yuan, K. Albin, A. Aziz, and C. Pixley, "Constraint synthesis for environment modeling in functional verification," *Proceedings of the 40th Design Automation Conference*, pp. 296–299, June 2003.

[17] J. Yuan, C. Pixley, A. Aziz, and K. Albin, "A framework for constrained functional verification," *Proceedings of the 2003 IEEE/ACM International Conference on Computer-Aided Design*, pp. 142–145, November 2003.

[18] M. A. Iyer, "RACE: A word-level ATPG-based constraints solver system for smart random simulation," *Proceedings of the 2003 International Test Conference*, pp. 299–308, September 2003.

[19] C.-H. Shih, J.-D. Huang, and J.-Y. Jou, "Stimulus generation for interface protocol verification using the non-deterministic extended finite state machine model," *Proceedings of the 10th IEEE International Workshop on High Level Design Validation and Test*, pp. 87–93, November 2005.

[20] S. Inc., "Constrained-random test generation and functional coverage with vera," *Technical report*, February 2003.

[21] N. Kitchen and A. Kuehlmann, "Stimulus generation for constrained random stimulation," *Proceedings of the 2007 IEEE/ACM International Conference on Computer-Aided Design*, pp. 258–265, November 2007.

[22] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient sat solver," *Proceedings of the 38th Design Automation Conference*, pp. 530–535, June 2001.

[23] N. Een and N. Sorensson, “An extensible sat-solver,” *International Conference on Theory and Applications of Satisfiability Testing*, pp. 502–518, May 2003.

[24] S. B. Akers, “Binary decision diagrams,” *IEEE Transactions on Computers*, pp. 509–516, June 1978.

[25] P. Goel, “An implicit enumeration algorithm to generate tests for combinational logic circuits,” *IEEE Transactions on Computers*, pp. 215–222, March 1981.

[26] H. Foster, A. Krolnik, and D. Lacey, “Assertion-based design,” *Kluwer Academic Publishers*, 2004.

[27] J. Bergeron, “Writing testbenches: Functional verification of HDL models,” *Kluwer Academic Publishers*, 2003.

[28] D. Drako and P. Cohen, “HDL verification coverage,” *Integrated System Design Magazine*, pp. 46–52, June 1998.

[29] F. Fallah, S. Devadas, and K. Keutzer, “OCCOM: Efficient computation of observability-based code coverage metrics for functional verification,” *Proceedings of the 35th Design Automation Conference*, pp. 152–157, June 1998.

[30] P. A. Thaker, V. D. Agrawal, and M. E. Zaghloul, “Validation vector grade (VVG): A new coverage metric for validation and test,” *Proceedings of the IEEE VLSI Test Symposium*, pp. 182–188, April 1998.

[31] B. Min and G. Choi, “ECC: Extended condition coverage for design verification using excitation and observation,” *Proceedings of the Pacific Rim International Symposium on Dependable Computing*, pp. 183–190, December 2001.

[32] T.-Y. Jiang, C.-N. J. Liu, and J.-Y. Jou, “Observability analysis on HDL descriptions for effective functional validation,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1509–1521, August 2007.

[33] G. V. Bochmann and J. Gecsei, “A unified method for the specification and verification of protocols,” *Proceeding of the International Federation for Information Processing Congress ’77*, pp. 229–234, August 1977.

[34] K. Ara and K. Suzuki, “A proposal for transaction-level verification with component wrapper language,” *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pp. 82–87, March 2003.

[35] M.-Y. Su, C.-H. Shih, J.-D. Huang, and J.-Y. Jou, “FSM-based transaction-level functional coverage,” *Proceedings of the Asia and South Pacific Design Automation Conference*, pp. 448–453, January 2006.

[36] K.-T. Cheng and A. S. Krishnakumar, “Automatic functional test generation using the extended finite state machine model,” *Proceedings of the 30th Design Automation Conference*, pp. 86–91, June 1993.

[37] D. Lee and M. Yannakakis, “Optimization problems from feature testing of communication protocols,” *Proceedings of the 4th International Conference on Network Protocols*, pp. 66–75, October 1996.

[38] C. Besse, A. Cavalli, and D. Lee, “An automatic and optimized test generation technique applying to TCP/IP protocols,” *Proceedings of the 14th International Conference on Automated Software Engineering*, pp. 73–80, October 1999.

[39] A. Limited, “AMBA specification (rev. 2.0),” May 1999.

[40] K. Lahiri, A. Raghunathan, and G. Lakshminarayana, “Lotterybus: A new high-performance communication architecture for system-on-chip design,” *Proceedings of the 38th Design Automation Conference*, pp. 15–20, June 2001.

[41] P. H. Bardell, W. H. McAnney, and J. Savir, “Built-in test for vlsi: Pseudorandom techniques,” *New York: John Wiley & Sons, Inc.*, 1987.

[42] O. Organization, “Specification for the: Wishbone system-on-chip (soc) interconnection architecture for portable IP cores, rev. b.3,” September 2002.

[43] [Online]. Available: http://www.opencores.org/

[44] F. Somenzi, “CUDD: Colorado university decision diagram package.” [Online]. Available: http://vlsi.colorado.edu/~fabio/CUDD/

[45] "Property specification language–language reference manual version 1.1." [Online]. Available: http://www.eda.org/vfv/docs/PSL-v1.1.pdf

[46] M. U. Uyar and A. Y. Duale, "Resolving inconsistencies in EFSMs using simultaneous reachability analysis," *Proceedings of IEEE Military Communications Conference*, pp. 135–139, October 1999.

[47] B. Karacali, K. C. Tai, and M. A. Vouk, "Deadlock detection of EFSMs using simultaneous reachability analysis," *Proceedings of International Conference on Dependable Systems and Networks*, pp. 315–324, June 2000.