

# Designing Generic Framework for IP Verification

Major Project Report

By

**Sahu Akruti Ashokbhai**

**09MEC002**



Department OF Electronincs and Communication Engineering

Ahmedabad-382481

May - 2011

# Designing Generic Framework for IP Verification

## Major Project

Submitted in partial fulfillment of the requirements

For the degree of

**Master of Technology in Electronics and Communication**  
(VLSI Design)

By

**Sahu Akruti Ashokbhai**

**09MEC002**

### External Guide

**Mr. Kishor Kulkarni**

Component Design Engineer,  
Intel Technology India Pvt. Ltd.  
Bangalore

### Internal Guide

**Dr. N. M. Devashrayee**

PG-Coordinator(VLSI Design),  
Elect. and Comm. Engineering,  
Institute of Technology,  
Nirma University, Ahmedabad



**Department OF Electronincs and Communication Engineering**

**Ahmedabad-382481**

**May - 2011**

## Declaration

This is to certify that

- i) The thesis comprises my original work towards the degree of Master of Technology in Electronics and Communication Technology at Nirma University and has not been submitted elsewhere for a degree.
- ii) Due acknowledgement has been made in the text to all other material used.

**Sahu Akruti Ashokbhai**

## Certificate

This is to certify that the Major Project entitled "**Designing Generic framework for IP Verification**" submitted by **Sahu Akruti Ashokbhai (09MEC002)**, towards the partial fulfillment of the requirements for the degree of Master of Technology in Electronics and Communication Engineering (VLSI Design) of Nirma University, Ahmedabad is the record of work carried out by her under my supervision and guidance. In my opinion, the submitted work has reached a level required for being accepted for examination. The results embodied in this major project, to the best of my knowledge, haven't been submitted to any other university or institution for award of any degree.

**Mr. Kishor Kulkarni**

External Guide

Component Design Engineer,  
Intel Technology India Pvt. Ltd.  
Bangalore

**Dr. N. M. Devashrayee**

Internal Guide

PG-Coordinator(VLSI Design),  
Elect. and Comm. Engineering,  
Institute of Technology,  
Nirma University, Ahmedabad

**Prof. A. S. Ranade**

Head of Department,  
Dept. of Electrical Engineering,  
Institute of Technology,  
Nirma University, Ahmedabad

**Dr. K Kotecha**

Director,  
Institute of Technology,  
Nirma University, Ahmedabad

## Acknowledgements

I express my gratitude and appreciation for all those with whom I worked and interacted at Institute of Technology, Nirma University, Ahmedabad and INTEL Technology (INDIA) Pvt Ltd, Bangalore thanks to all of them for their help and co-operation.

It has been great pleasure for me in doing a Major Project work "Designing Generic Framework for IP Verification " under guidance of Mr. Kishor Kulkarni and Mrs. Srilatha Pachanathan. I am very grateful to those who assigned me a project under their expert guidance, without their invaluable guidance the work would have not been possible. They always inspire to put best efforts to achieve the goal.

I am especially thankful to Mr. Manoj Velayudha, Mr. Krishnan Subramoniam and Mr. Priyank Vashisth from Intel Corporation for their guidance, encouragement, support and confidence throughout the internship as a part of curriculum. I am especially grateful to them for giving me the opportunity to work on such an exciting project.

I greatly appreciate the generosity of Dr. N. M. Devashrayee, PG Coordinator (VLSI Design), Institute of Technology, Nirma University for his valuable and inspiring guidance, patience and support at every moment, and for giving me permission and providing the facilities for completing this project.

I would also like to thank my colleagues and friends for the all that they have taught me. My greatest thanks are to all who wished me success especially my parents.

- Sahu Akruiti Ashokbhai

09MEC002

## Abstract

The rapid and dynamic information and knowledge transfer between designers during the conceptual phase of designing verification environment for any Reusable IP can result in time consuming and erroneous environment. This thesis describes the development and verification of a structured framework, which has been generated to aid and support conceptual design. The development of framework for IP verification is based on OVM (Open Verification Methodology). For designing verification environment in this directory case study of various project was carried out and summary of study observed was drawn in table II. From that conceptual design of framework is generated, which was followed by creation of hardcoded fragments for each component. In this project work, a tool is developed for generating the Framework. Script takes input from User, in GUI form and depending on the input, existence of reuse layer in verification environment and placement of components is decided at run time. Existence of reuse layer control some hierarchical references in the code, and then connection of various component in respective hierarchy was otherwise a tedious task is completed through the script with less efforts. Various other components were designed like bus interface tracker, which is useful for debugging and performance analysis. Quality has been tested by ensuring the framework is compile clean. Various configuration files were developed for compiling the framework with a generic frontend simulation environment. This framework will save efforts as well as time for creating basic essential components in the environment and is less prone to errors as the framework itself would be pre-validated. This framework, as the name suggests is the fundamental layer created for any generic IP validation and thus not only enforces the same look-n-feel across all IP environments but also ensures the quality discipline among the team members.

# Contents

<b>Declaration</b>	<b>iii</b>
<b>Certificate</b>	<b>iv</b>
<b>Acknowledgements</b>	<b>v</b>
<b>Abstract</b>	<b>vi</b>
<b>List of Tables</b>	<b>ix</b>
<b>List of Figures</b>	<b>x</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Introduction and Motivation . . . . .	1
1.2 Objectives of Thesis . . . . .	2
1.3 Thesis Organization . . . . .	2
<b>2 OVM Verification Environment &amp; Mechanics</b>	<b>4</b>
2.1 Verification Principles . . . . .	4
2.1.1 Verification Basics . . . . .	4
2.1.2 Two questions . . . . .	5
2.1.3 Two loop flow . . . . .	6
2.1.4 Concentric Testbench Organization . . . . .	7
2.2 OVM components . . . . .	8
2.2.1 Transactors . . . . .	8
2.2.2 Operational Components . . . . .	8
2.2.3 Analysis Components . . . . .	9
2.2.4 Controller . . . . .	10
2.3 Components and Hierarchy . . . . .	10
2.4 Connectivity . . . . .	11
2.5 Phases . . . . .	12
2.6 Config . . . . .	14
2.7 Factory . . . . .	16
2.8 Shutting down the testbench . . . . .	18

2.8.1	Global Function . . . . .	18
2.8.2	Timeout . . . . .	19
2.9	Tests and Testbenches . . . . .	19
2.10	Summary . . . . .	20
<b>3</b>	<b>Generic Simulation Environment</b>	<b>21</b>
3.1	System Verilog Package . . . . .	21
3.2	VCS (verilog compiler simulator) . . . . .	22
3.3	Configuration files . . . . .	23
3.3.1	Setting Environment . . . . .	23
3.3.2	Perl-syntax formatted file . . . . .	23
<b>4</b>	<b>Designing Generic Framework</b>	<b>24</b>
4.1	Creating Templates . . . . .	24
4.1.1	Case study projects . . . . .	24
4.1.2	Observations from case study data . . . . .	25
4.2	Flow of the tool . . . . .	27
4.3	Summary . . . . .	31
<b>5</b>	<b>Designing Bus Interface Tracker</b>	<b>32</b>
5.1	Template creation for tracker . . . . .	33
5.1.1	Flow of script . . . . .	35
5.2	Summary . . . . .	37
<b>6</b>	<b>Other Productive Contribution</b>	<b>38</b>
6.1	OVM Compliance checking for various IPs . . . . .	38
6.2	Added Coverpoint for mapping multi interrupt agents to single interrupt pin or line . . . . .	41
6.3	Written various test code . . . . .	41
6.3.1	For checking full condition for memory storage element . . . . .	41
6.3.2	For checking clock dutycycle of certain block . . . . .	41
6.3.3	For multiple msi . . . . .	42
6.4	Trainings attended . . . . .	42
<b>7</b>	<b>Conclusion</b>	<b>44</b>
	<b>References</b>	<b>45</b>



# List of Tables

I	Table of phases . . . . .	13
II	Configuration functions . . . . .	16
I	observation from case study . . . . .	26
I	Ovm component Tracker . . . . .	35
I	IP grading column . . . . .	39
II	ovm-ip-grading . . . . .	40

# List of Figures

2.1	Two-loop Flow . . . . .	6
2.2	Concentric Testbench Organization . . . . .	7
2.3	Simple Hierarchy of components . . . . .	10
2.4	small design showing traversing . . . . .	11
2.5	Connecting an initiator to a target . . . . .	12
2.6	Each component has a database of configuration items . . . . .	15
4.1	Concept of Reuselayer . . . . .	25
4.2	Flow of the tool . . . . .	27
4.3	Input to script through GUI . . . . .	28
4.4	Flow chart . . . . .	30
5.1	Tracker output . . . . .	32
5.2	Tracker flow . . . . .	34
5.3	Tracker script flow . . . . .	36

# Chapter 1

## Introduction

### 1.1 Introduction and Motivation

The conceptual phase of any design project is potentially the most vibrant, dynamic and creative stage of the overall design process. It is at early stage that designers need to interact freely to achieve optimal verification environment that eliminate or reduce the need for compromise of design at a later, more critical period of the process. Currently, for developing verification environment for any Reusable IP, all components are to be created from scratch which is time consuming and is prone to error. This thesis is based on actual work designing generic framework for IP verification and is motivated by these several factors. As generic framework include all the necessary components with its minimal functionality without including project specific data for any generic IPs. These will save efforts as well as time for creating basic essential components in the environment and is less prone to errors as the framework itself would be pre-validated. Thus this framework will help effective deployment of new IP environments in future projects.

This project is divided into two distinct phases Designing and Validating. The first phase (Designing) involves creating templates for basic components according to OVM methodology of verification. Second phase (validating) involves compiling and elabo-

rating the templates by developing configuration files for generic frontend simulation environment.

## 1.2 Objectives of Thesis

The main objectives of the thesis are:

- 1) Auto creating Templates for basic component in verification environment using a script.
- 2) Ensure the whole framework is self sufficient in terms of compilation and elaboration of the templates.

## 1.3 Thesis Organization

The rest of the thesis is organized as follows.

**Chapter 2**, *OVM Verification Environment and OVM Mechanics*, describes the overall verification process focusing on two questions. This chapter introduces and briefly describes the OVM components for testbench organization and also explains the mechanics of OVM, illustrating how to build hierarchies of class-based verification components and connect them with transaction-level interfaces.

**Chapter 3**, *Generic Simulation Environment*, describes the *template simulation environment* required to compile and simulate any IP verification environment. This chapter includes packages in system verilog and configuration files for simulation environment.

In **chapter 4**, *Designing Generic Framework*, Presents the actual work carried out during the project. This chapter describes the case study carried out, based on this study, using a PERL script generation of templates is carried out. This chapter also includes designing of configuration files for validating the environment. The whole templates are compiled and elaborated using VCS simulator from Synopsys®.

In **chapter 5**, *Designing Bus interface Tracker*, Presents the concept of tracker. Actual work done for generating generic template for tracker is also described in this chapter. The inputs to the tracker generating script and flow of the script are also presented.

In **chapter 6**, *Other Productive contribution*, describes the other productive contribution carried out during the period of internship, which includes OVM Compliance checking for various IPs, adding coverpoint to coverage, writing various test code and attending various trainings.

Finally, in **chapter 7** concluding remarks is presented.

## Chapter 2

# OVM Verification Environment & OVM Mechanics

The OVM (Open Verification Methodology) is the result of joint development between Cadence and Mentor Graphics to facilitate true SystemVerilog interoperability with a standard library and a proven methodology. Completely open source, it combines the best of the Cadence Incisive Plan-to-Closure Universal Reuse Methodology (URM) and the Mentor Advanced Verification Methodology (AVM). The OVM is a programming environment built upon SystemVerilog. It is designed to enable the development of complex testbenches.

## 2.1 Verification Principles

### 2.1.1 Verification Basics

Functionally verifying a design means comparing the designer's intent with observed behavior to determine their equivalence. Every testbench has some kind of reference model and a means to compare the function of the design with the reference. The reference can take many forms, such as a document describing the operation of the DUT, a golden model that contains a unique algorithm, or assertions that represent

a protocol.

### 2.1.2 Two questions

Verifying a design involves answering two questions: Does it work? and Are we done? The first question comes from the essential idea of verification where the major role is to determine functional correctness of the design. The second asks if we have satisfactorily compared the design and intent to conclude whether the design does indeed match the intent, or if not, why not. Are-we-done questions are also call for functional coverage, questions that relate to whether the design is sufficiently covered by the test suite in terms of design features. The most important part in the whole verification process is the approach. When a good approach is followed to design the test benches, one does not have to go back and forth in developing / checking the functionality. Here comes the need for a common, reusable and easily pluggable verification environment framework. Although different designs may function differently, the verification framework needed for those designs are abstracted at a higher level and are common and reusable. Hence the efforts in this project is to concentrate on developing such generic framework with less / minimal efforts and help all IP teams with a easy launch for the verification environment.

### 2.1.3 Two loop flow

The process for answering the does-it-work and are-we-done questions can be described in a simple flow diagram as shown in figure 2.1. Everything is driven by the functional (architectural) specification for the design. From the functional specification, the design and the verification plan can be derived. The verification plan drives the testbench construction.

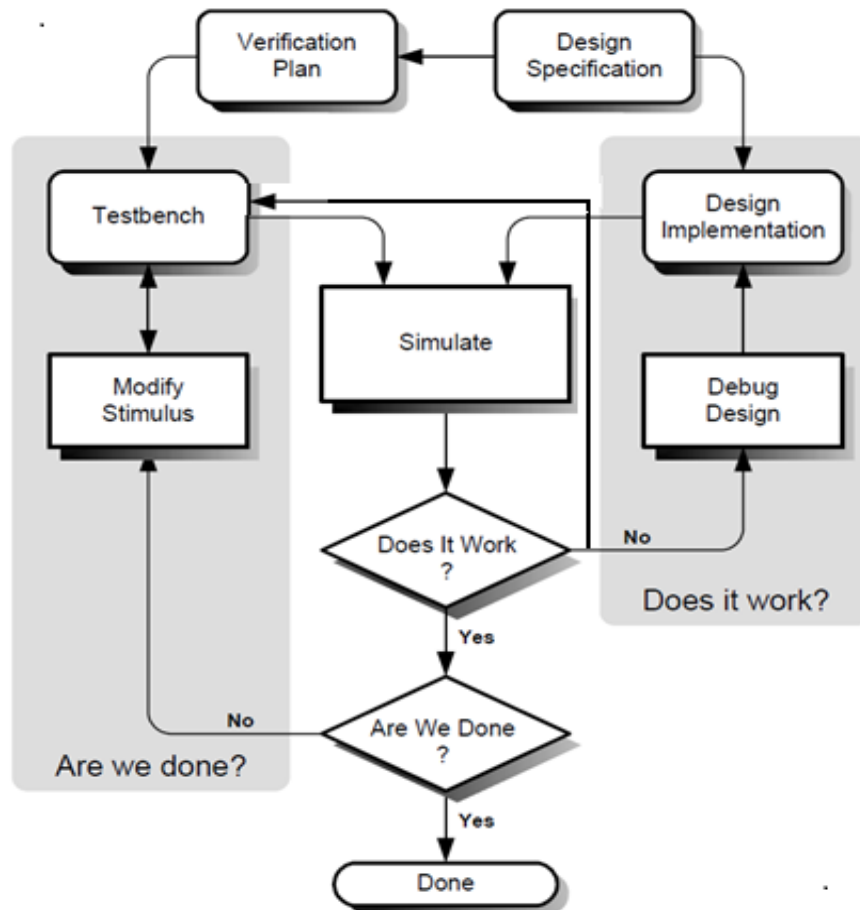


Figure 2.1: Two-loop Flow



### 2.1.4 Concentric Testbench Organization

The OVM defines verification components, their structure, and interfaces. OVM testbenches are organized in layers. The innermost layer is the DUT, an RTL device with pin-level interfaces. Above that is a layer of transactors, devices that convert between the transaction-level and pin-level worlds. The components in the layers above the transactor layer are all transaction-level components. Figure 2.2 illustrates the Concentric testbench organization.

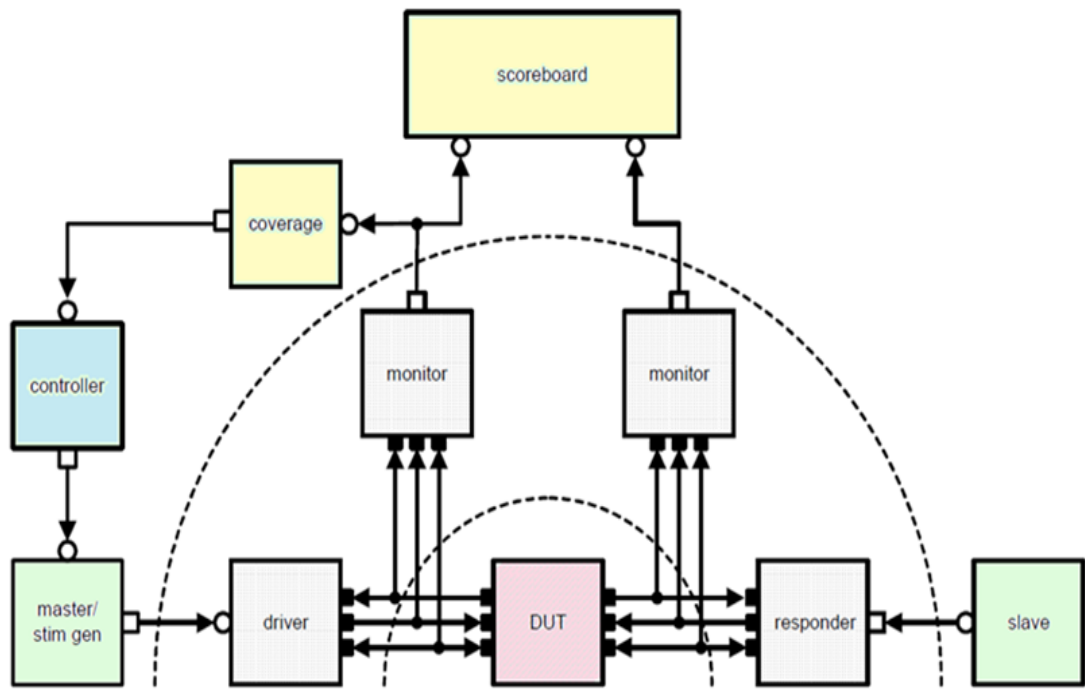


Figure 2.2: Concentric Testbench Organization

## 2.2 OVM components

### 2.2.1 Transactors

The role of a transactor in a testbench is to convert a stream of transactions to pin-level activity or vice versa. Transactors are characterized by having at least one pin-level interface and at least one transaction-level interface.

**Monitor.** A monitor, as the name implies, monitors a bus. It watches the pins and converts their wiggles to a stream of transactions. Monitors are passive, meaning they do not affect the operation of the DUT in any way.

**Driver.** A driver converts a stream of transactions (or sequence items) into pin-level activity.

**Responder.** A responder is much like a driver, but it responds to activity on pins rather than initiating any activity.

### 2.2.2 Operational Components

The operational components are the set of components that fulfill all the requirements the DUT needs to operate. The operational components are responsible for generating traffic towards the DUT. They are all transaction-level components and have only transaction-level interfaces (called TLM). The ways to generate stimulus are as varied as the kinds of devices there are to verify.

**Stimulus Generator.** Stimulus generators create a stream of transactions passed on to the driver to be able to convert to pin level activities and drive towards the 'DUT'. Stimulus generators can be of varied types, such as random, directed, or directed random (also called pseudo random); they can be free running or have controls; and they can be independent or synchronized. The simplest stimulus generator randomizes the contents of a request object and sends that object to a driver through a TLM connection.

**Master.** A master is a bidirectional component that sends requests and receives responses. Masters initiate activity. Like stimulus generators, they can send individual randomized transactions or sequences of directed or directed-random transactions. Masters may use the responses to determine their next course of action. Masters can also be implemented in terms of sequences.

**Slave.** Slaves, like masters, are bidirectional components. They respond to requests and return responses (in contrast to masters, which send requests and receive responses).

### 2.2.3 Analysis Components

Analysis components receive information about what's going on in the testbench and use that information to make some determination about the correctness or completeness of the test. Two common kinds of analysis components are scoreboards and coverage collectors.

**Scoreboard.** Scoreboards are used to determine functional correctness of the DUT, to answer does-it-work questions. Scoreboards are connected to the DUT activities through transactor components like monitor using TLM connections and help determine if the DUT is behaving correctly to its stimulus.

**Coverage Collector.** Coverage collectors count things. They tap into streams of transactions and count the transactions or various aspects of the transactions. The purpose is to determine verification completeness by answering are-we-done questions. The particular things that a coverage collector counts depend on the design and the specifics of the test. Coverage collectors can also perform computations as part of a completeness check.

### 2.2.4 Controller

Controllers form the main thread of a test and orchestrate the activity. Typically, controllers receive information from scoreboards and coverage collectors and send information to environment components. For example, a controller might start a stimulus generator running and then wait for a signal from a coverage collector to notify it when the test is complete. The controller, in turn, stops the stimulus generator. More elaborate variations on this theme are possible. In an example of a possible configuration, a controller supplies a stimulus generator with an initial set of constraints and starts the stimulus generator running. When a particular ratio of packet types is achieved, the coverage collector signals the controller. Rather than stopping the stimulus generator, the controller may send it a new set of constraints.

## 2.3 Components and Hierarchy

The primary structure for building testbench elements is the component. OVM is responsible for creating the component instances and assembling them into hierarchies. Figure 2.3 illustrates a simple hierarchy of components.

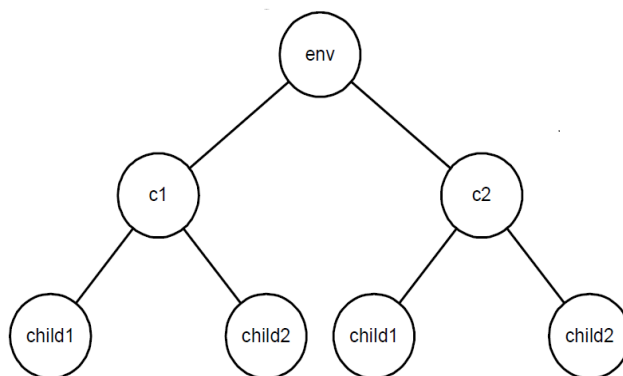


Figure 2.3: Simple Hierarchy of components

The top-most node, `env`, is the root. The root is distinguished by the fact that it has no parent. All other nodes have exactly one parent. Each node has a name. The location in the hierarchy of each node can be identified by a unique `full_name` (path), which is constructed by stringing together the names of all the nodes between the root and the node in question, separating them with a hierarchy separator, dot (`.`). The children of a component are stored in an associative array. This array is not directly accessible, but it can be accessed through a hierarchy API.

```

+ env
|  + env.c1
|  |  env.c1.child1
|  |  env.c1.child2
|  + env.c2
|  |  env.c2.child1
|  |  env.c2.child2

```

Figure 2.4: small design showing traversing

## 2.4 Connectivity

Components are connected to each other through TLM ports and exports. Ports and exports provide a means for components, or more accurately, processes in components, to synchronize and communicate with each other. Ports and exports are objects that form a binding point to enable intercomponent communication. Exports provide functions and tasks that can be called by ports shown in figure 2.5.

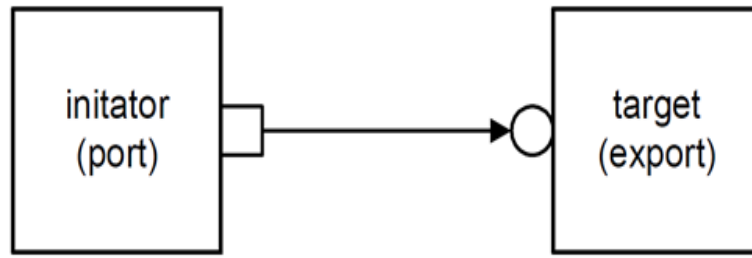


Figure 2.5: Connecting an initiator to a target

The connect method on ports and exports is used to bind the two together.

```
initiator_port.connect(target.export)
```

This method creates an association, or a binding, between the port and export so that the port can now call tasks and functions on the export. For the connection to be made successfully, the types of the port and export must match. That is, the interface types must be the same, and the type of the object being transferred in the interface must be the same.

## 2.5 Phases

Traditional Verilog modules rely on the simulator to elaborate the complete design and kick off its execution. Since OVM components are classes, they are instantiated and connected, and their execution is initiated outside of the Verilog elaborator. Components come into existence by calling class constructor `new()`, which allocates memory and performs initializations. Rather than the Verilog run-time engine managing instantiation, elaboration, and execution of class-based components, component functionality is broken into phases, and the OVM phase controller manages their execution.

phase name	function/ task	order
new	function	top-down
build	function	top-down
connect	function	bottom-up
end_of_elaboration	function	bottom-up
start_of_simulation	function	bottom-up
run	task	bottom-up
extract	function	bottom-up
check	function	bottom-up
report	function	bottom-up

Table I: Table of phases

Each phase is represented in the component as a virtual method (task or function) with a trivial default implementation. These phase callbacks are implemented by the component developer, who supplies appropriate functionality. The phase controller ensures that the phases are executed in the proper order. The set of predefined phases is shown in the table I. Each phase has a specific purpose. Component builders must take care to ensure that the functionality implemented in each phase callback is appropriate to the phase definition.

**new** is not technically a phase, in that it's not managed by the phase controller. However, for each component, the constructor must execute and complete in order to bring the component into existence. Therefore, `new()` must run before `build()` or any other subsequent phases can execute.

**build** is the place where new components, ports, and exports are instantiated and con-

figured. This is also the recommended place for calling `set_config_*` and `get_config_*`. (refer section 2.6)

**connect** is where components, ports, and exports created in `build()` are connected. **end\_of\_elaboration** is where you can make configuration changes, knowing that elaboration is complete. That is, you can assume that all components are built and connected.

**start\_of\_simulation** executes just before time 0.

**run** is the only pre-defined task phase. All of the run tasks are forked to run in parallel. Each run task continues until its locus of control passes the `endtask` statement or it is explicitly shut down. Later in this chapter, we will discuss how to shut down test-benches.

**extract** is intended for collecting information relating to coverage or other information about how to answer the testbench questions.

**check** is where any correctness checking or validation of extracted data is done.

**report** is where final reports are produced.

## 2.6 Config

To increase reusability of components, it's desirable to sprinkle them liberally with parameters that can be externally configured. The config facility provides a means to do just this. It is based on a database of name-value pairs called configuration items that is organized hierarchically. Each component contains a configuration table of configuration items as shown in Figure 2.6 and, since components are arranged in a tree, each element in the database can be uniquely located by the location of the component and the name of the configuration item.



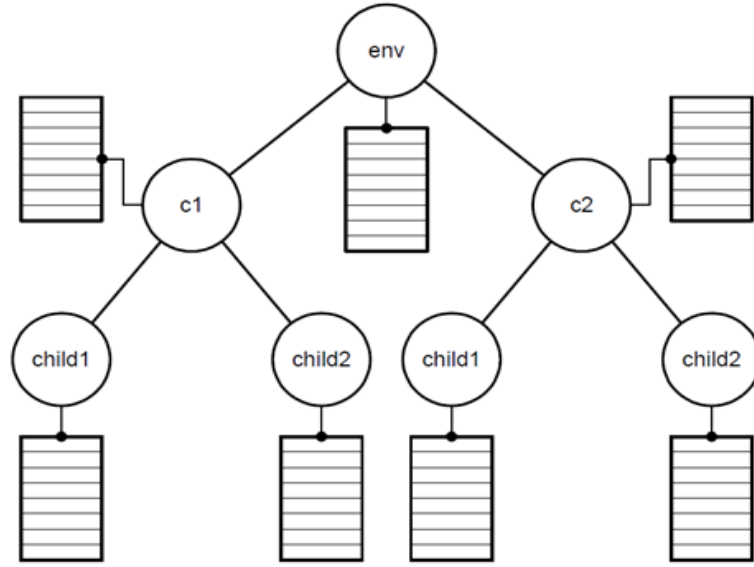


Figure 2.6: Each component has a database of configuration items

The `ovm_component` class contains two sets of methods for putting configuration items into the database and for retrieving them later. These are `set_config_*` and `get_config_*`. The Table II below shows both sets. The `set_config_*` functions place an item in the configuration database in the current component, that is, in the component instance in which the function is called. These functions each take three arguments, `name`, `field_name`, and `value`. The argument `name` is a path name that represents the scope of the components that are to accept this configuration item. `name` is used in `get_config_*` to locate items in the configuration database. `field_name` is the name of the field and must be unique within the current configuration database. `value` is the value part of the name-value pair and its type can be string, int, or `ovm_object`, depending on which function is being called. In addition, `set_config_object` takes a `clone` argument to indicate whether the object being passed in as the value should be cloned before it is put into the configuration database.

Configuration Database Access Functions
<code>set_config_int(string name, string field_name, int value)</code>
<code>set_config_string(string name, string field_name, string value)</code>
<code>set_config_object(string name, string field_name, ovm_object object, bit clone);</code>
<code>get_config_int(string field_name, inout int value);</code>
<code>get_config_string(string field_name, inout string value);</code>
<code>get_config_object(string field_name, inout ovm_object object, input bit clone);</code>

Table II: Configuration functions

The `get_config_*` functions retrieve items from the configuration database. These functions take only two arguments, a field name and an inout variable that contains the value of the item located. They also return a bit to indicate whether the requested item was successfully located. The `get_config_*` functions do not take a path name argument like their `set_config_*` counterparts because they use the path of the current component as the point of reference to locate configuration items. They are designed to inquire as to the value of a configuration item for the current context, that is, the component in which the `get_config_*` function is called.

## 2.7 Factory

The structure of a testbench is determined by the organization of the components into a hierarchy and the way these objects are connected. The behavior of the testbench

is determined by the procedural code in the phase callbacks `build`, `connect`, `run`, and so forth. There are times when it is desirable to modify the behavior or part of the structure externally, that is, at run time, without touching the testbench code. For example, to inject errors into a system, you may want to replace the normal driver with an error driver, one that intentionally injects errors. Instead of re-coding the environment to use a different driver, you can use the factory to do the substitution automatically.

The factory provides a means for substituting one object for another without having to use your text editor to modify the testbench. Instead of creating the object using `new()`, you invoke a `create` function in the factory. The factory keeps a list of registered objects and, optionally, a set of overrides associated with each one. When you create an object using the factory, the list of overrides is consulted. If one is present, then the override object is returned. Otherwise, the registered object is returned.

The factory is an OVM data structure. It is global in scope, and only one instance exists (that is, it's a singleton). It serves as a polymorphic constructor, a single function that lets you build a variety of different objects. It provides a means for registering objects and for specifying overrides. Objects registered as overrides must be derived from the object they are overriding. To have a single function return multiple objects, each of those objects must be derived from a common base class.

An essential component of the factory is the wrapper, a class that wraps the object we wish to register with the factory. The factory data structure is a table of wrappers indexed by a key. The wrapper has a `create()` function that delegates to the constructor of the wrapped object.

Using the factory involves three steps: registration, setting overrides, and creation. In the first step, you register an object with the factory. In the second step, you add an override to a registered object. In the third step, you create an object with the factory that will return either the originally registered object or an override, depending on whether an override was registered for the requested object.

## 2.8 Shutting down the testbench

### 2.8.1 Global Function

The easiest way to shut down an OVM testbench is to call the global function `global_stop_request()`. This requests that the testbench shut down. If there is no reason not to shut down, then the testbench will terminate. What reasons would there be to not allow a shutdown? Every component has a virtual task `stop()`. When you call `global_stop_request()`, this task is called for each component whose member `enable_stop_request` is set to 1. When all of the stop tasks have returned, then the testbench shuts down. The stop task can be used to clean things up, tell the DUT to shut down, serve as a shutdown objection, or anything else you'd like to do before completing the current phase. Since it is a task, `stop()` can consume time. A `stop()` task can disallow the shutdown by blocking. It can wait for some condition to be set or delay a fixed time. The `stop()` task services the shutdown request. When `stop()` returns, it allows the request to be granted. The following example illustrates how the stop mechanism works. This example consists of two producers sending transactions to a consumer through a FIFO. Each producer runs independently of the other. We want to make sure that both producers finish their respective jobs. When one finishes, the other continues until it is done. Upon starting, the task immediately calls `global_stop_request()`, which causes the `stop()` tasks to be called in the producers (because `enable_stop_interrupt` is set to 1 in each). In turn, each producer blocks until its respective done flag is set. When the producer with the smallest number of iterations finishes, it triggers its local done flag and its stop task returns. However, because there are outstanding blocked stop tasks, the simulation continues. Only when all of the stop tasks complete will the simulation terminate.

### 2.8.2 Timeout

It's possible that a simulation can deadlock when a bug in a stop task prevents it from returning, a blocked call never unblocks, or a forever loop never breaks. To prevent the simulation from hanging indefinitely, OVM provides two watchdog timeout mechanisms. One is for task phases, and the other is for stop tasks. `ovm_root` contains two variables, `phase_timeout` and `stop_timeout`. Their type is the Verilog type `time`, and their values can be set by `set_global_timeout` and `set_global_stop_timeout`. The default value for both variables is 0, which means timeout is disabled. When a task-based phase is executed, such as `run()`, and `phase_timeout` has been set to a value greater than zero, then a separate watchdog process is spawned that simply waits until the timeout expires. A `fork/join_any` construct is used to spawn these tasks, so if the run tasks finish before the timeout expires, then the timeout is ignored. On the other hand, if the timeout expires first, then it will initiate a shutdown.

## 2.9 Tests and Testbenches

Through the proper use of configuration, the factory, and the phased build process, you can create a verification testbench that allows you to randomize more than just the generated stimulus. For example, if a testbench is written to allow the number of drivers on a bus to be configurable, then the same testbench can be reused across multiple tests, each of which might specify a different (possibly random) number of drivers. As you can see, the flexibility of OVM allows you to run each of these different tests without having to modify the testbench itself.

The OVM also provides an explicit `ovm_test` class as a container for tests. Typically, the top-level module will instantiate an `ovm_test`, which in turn configures and instantiates the testbench. Additional tests can then be written as extensions of the base test that include new configuration and factory directives, making the tests themselves relatively short, well-defined, and easy to maintain. In actuality, the `ovm_test` is simply another extension of `ovm_component`. Since tests and testbenches

are simply components, they too can be created and overridden via the factory.

## 2.10 Summary

In this chapter we looked at how to structure an overall verification process. The process is based on two fundamental questions, Does it work? and Are we done? Testbench organization with all OVM components are also discussed in this chapter. The rest of the chapter explains OVM testbench components that answer these questions and shows how to connect them to form testbenches. An understanding of the concepts discussed in this chapter enables to construct the essential elements of a testbench using the OVM. This chapter describes how to create arbitrary hierarchies of class-based components, connect them, configure them, run them, and shut them down.

# Chapter 3

## Generic Simulation Environment

UNIX HDL simulation front-end environment is flexible and easy to use. It provides a common environment to many projects while providing project-specific customization.

### 3.1 System Verilog Package

Packages provide ways to have common code to be shared across multiple modules. SystemVerilog provides package support to help share following among multiple System Verilog modules, interfaces and programs.

- parameters
- data
- type
- task
- function
- sequence

- property

Above can be shared across SystemVerilog modules, macromodule interfaces and programs. Few rules that should be followed with packages are.

- Packages can not contain any assign statement.
- Variable declaration assignments within the package shall occur before any initial, always, always\_comb, always\_latch, or always\_ff blocks
- Items within packages cannot have hierarchical references.
- Assign statement on any net type is not allowed.

A package is to group a type declaration and a set of tasks or functions that operate on that type. In this scenario, a module could then declare objects of the type and use the package tasks/functions to operate on the object data.

A package is to define a utility for common use. This sort of package often makes use of persistent state and non-reentrancy in its implementation.

## **3.2 VCS (verilog compiler simulator)**

VCS is a high-performance, high-capacity Verilog® simulator that incorporates advanced, high-level abstraction verification technologies into a single open native platform. VCS enables you to analyze, compile, and simulate Verilog design descriptions. It also provides you with a set of simulation and debugging features to validate your design. These features provide capabilities for source-level debugging and simulation result viewing. VCS supports all levels of design descriptions, but is optimized for the behavioral and register transfer levels. VCS accelerates complete system verification by delivering the fastest and highest capacity Verilog simulation for RTL



functional verification. In addition, VCS supports Synopsys DesignWare IP, the VCS Verification Library, VMC models, and the Vera testbench tool.

## **3.3 Configuration files**

### **3.3.1 Setting Environment**

Before simulator runs, the environment must be setup. This includes any tools that will run (e.g., the simulator, a wave viewer..) and environment variables. Simulator itself does not set any environment, and this separation of concerns provides for a more transparent operation, and easy reproducibility of results. The main benefit of using the dispatcher is that users needn't worry about explicitly re-sourcing the environment when updating their local work area from the repository.

### **3.3.2 Perl-syntax formatted file**

In order to provide a converged validation environment, and aid in tool maintainability, the tool changes should be minimal. In other words, all projects should use the same methodology and use the same code. However, there are differences in projects that should be accounted for. For example, the RTL source code from one project to another is different. Perl-syntax formatted text file sets a number of project specific variables. This text file contain commands of the sort that might have been invoked manually once the simulation started running but are to be executed automatically each time the simulation starts up.

# Chapter 4

## Designing Generic Framework

The development of the framework involved a bottom-up approach to grouping the design tasks from the various case study projects. In essence the approach involved the generation of a framework hierarchy, with the project-specific tasks acting as the basis of development, and the clustering of these tasks in relation to their combined objective representing the generic group characteristics. This hierarchy represented a generic building design framework in which each of the case study projects and the designing together process descriptions could be contained.

### 4.1 Creating Templates

#### 4.1.1 Case study projects

Various projects of designing IP verification environment were examined during this research period in order to gain an appreciation of the similarities and differences in the tasks undertaken during the conceptual design of environment. The case studies represented an investigation of the ways in which differing factors can influence the activities involved in the conceptual design process.

### 4.1.2 Observations from case study data

The various sources of data, relating to each case study project, were compiled and cross-referenced. In some project concept of reuselayer was used. All the well defined, essential components are encapsulated in this reuselayer. As shown in fig 4.1, Top layer "Env" is root and "Reuse layer" is defined in this environment.

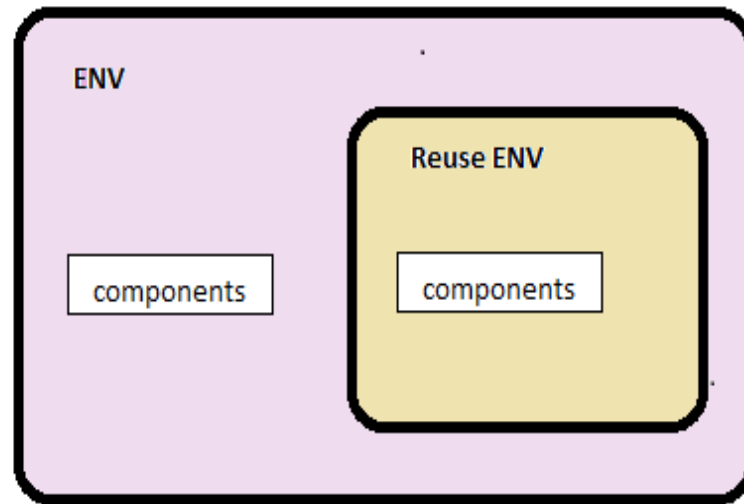


Figure 4.1: Concept of Reuselayer

A synopsis of each case study was generated, in addition to a list of the design tasks that were involved in each of the respective projects. Subsequently, these data was passed to those individuals who provided the information for verification of data. In this manner, errors in the descriptions were highlighted and amended quickly and efficiently. This procedure allowed the generation of a robust and detailed description of the various design tasks involved during the conceptual phase of each of the projects. This allowed a number of general observations to be made, the most germane being :

- There was a variation whether reuse layer environment is present or all the components are there in upper layer environment only.

Project	Reuse layer	total no of components	no of component in Reuse layer	no of component in Top env	no of tasks not project specific
A	yes	7	7	0	8
B	no	7	-	7	7
C	yes	7	5	2	7
D	no	6	-	6	5

Table I: observation from case study

- There was a variation in the number of components present in verification environment identified in each project II. These variation depends on project complexity.
- From II it is seen that also there is variation in no of component in Reuse layer as well as upper layer.
- There was a variation in the number of tasks that were identified in each project. This difference was recognised as being the result of number of components involved in environment.
- Within the high level details of tasks it was apparent that some were common to all projects, while others were very much project specific.

## 4.2 Flow of the tool

Template generating tool is developed using Perl script as shown in fig 4.2. This PERL Script gets input from user in GUI form. Initially users use to enter inputs in text file, for getting templates of various components according to requirements. Now the code template generation script has been enhanced using TK, to create GUI (Graphics User Interface). So that it is easy for user to enter the project data.

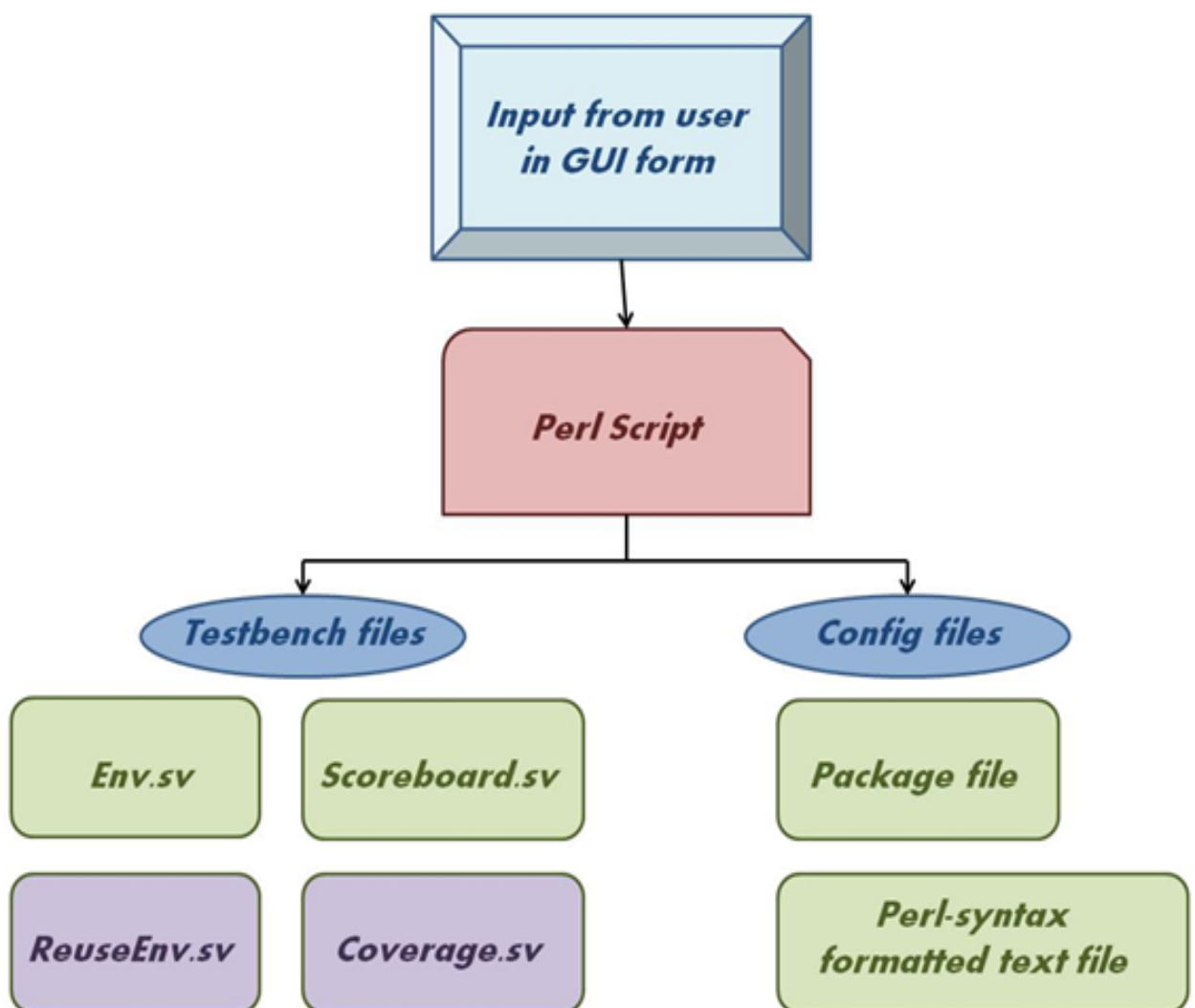
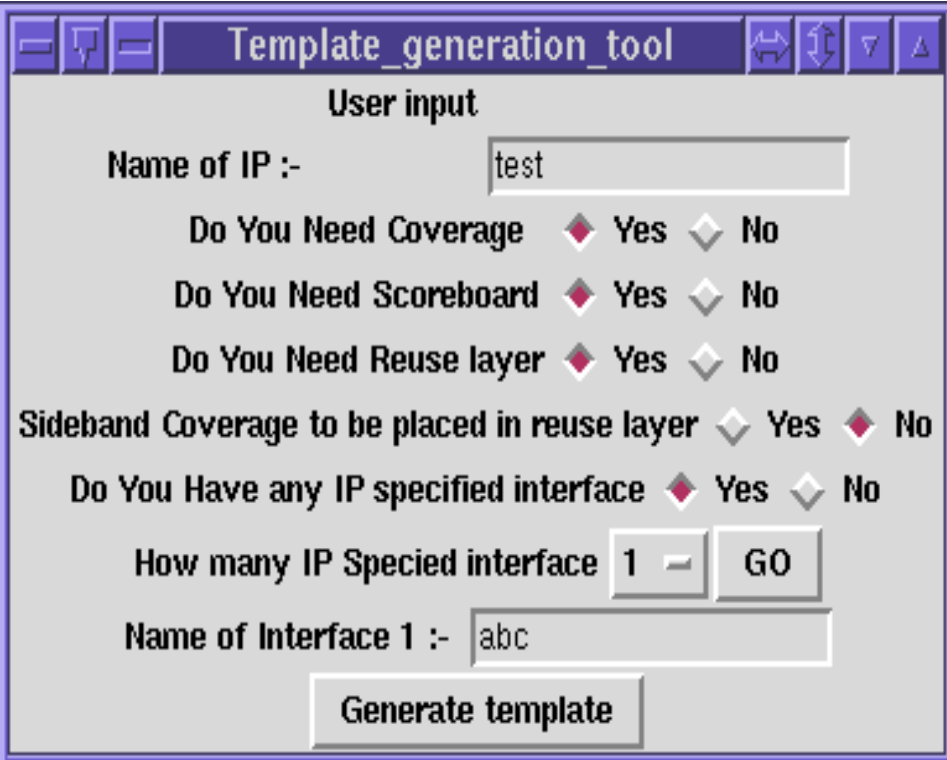


Figure 4.2: Flow of the tool

Here as shown in fig 4.3 user need to enter the name of his IP, all the templates will be generated using IP name. User has flexibility to choose various components for their project (IP), and also the location of this component, whether in top Env or in Reuse layer. Number and name of IP specified interface can also be mentioned, template will contain placeholders for this ip specific interfaces.



The screenshot shows a GUI window titled "Template\_generation\_tool". It contains a "User input" section with the following elements:

- Name of IP :-** A text box containing the word "test".
- Do You Need Coverage** with radio buttons for **Yes** (selected) and **No**.
- Do You Need Scoreboard** with radio buttons for **Yes** (selected) and **No**.
- Do You Need Reuse layer** with radio buttons for **Yes** (selected) and **No**.
- Sideband Coverage to be placed in reuse layer** with radio buttons for **Yes** (selected) and **No**.
- Do You Have any IP specified interface** with radio buttons for **Yes** (selected) and **No**.
- How many IP Specied interface** with a text box containing "1" and a **GO** button.
- Name of Interface 1 :-** A text box containing "abc".
- A **Generate template** button at the bottom.

Figure 4.3: Input to script through GUI

Output of the script is templates for testbench files and the configuration files which are used to pre-validate testbench environment templates. With this tool, essentially env and scoreboard template are generated, while generation of reuseenv and coverage template depends on the input provided by user in addition to all configuration files.

- Testbench files
  - Env.sv and ReuseEnv.sv
    - \* Class declaration
    - \* Function definitions (not project specific)
    - \* Phases (OVM)
    - \* Task of idle detection
    - \* Placeholders for project specific function and task
  - Scoreboard.sv and Coverage.sv
    - \* Class declaration
    - \* Phases (OVM)
    - \* Placeholders for project specific function and task
- Configuration files
  - Package files
    - \* Includes all testbench files
  - Perl syntax formatted text file

**Flow of Script:-**

Refer Figure - 4.4 In this script, initially get inputs from user through GUI as shown in figure. If answer for the field do you need component say (coverage) is yes than component\_flag (coverage\_flag) is set to 1, else set to 0. Then component\_list file is get opened and name of component is read, if the flag for that component is set than it will open the fragment list and create template for that component.

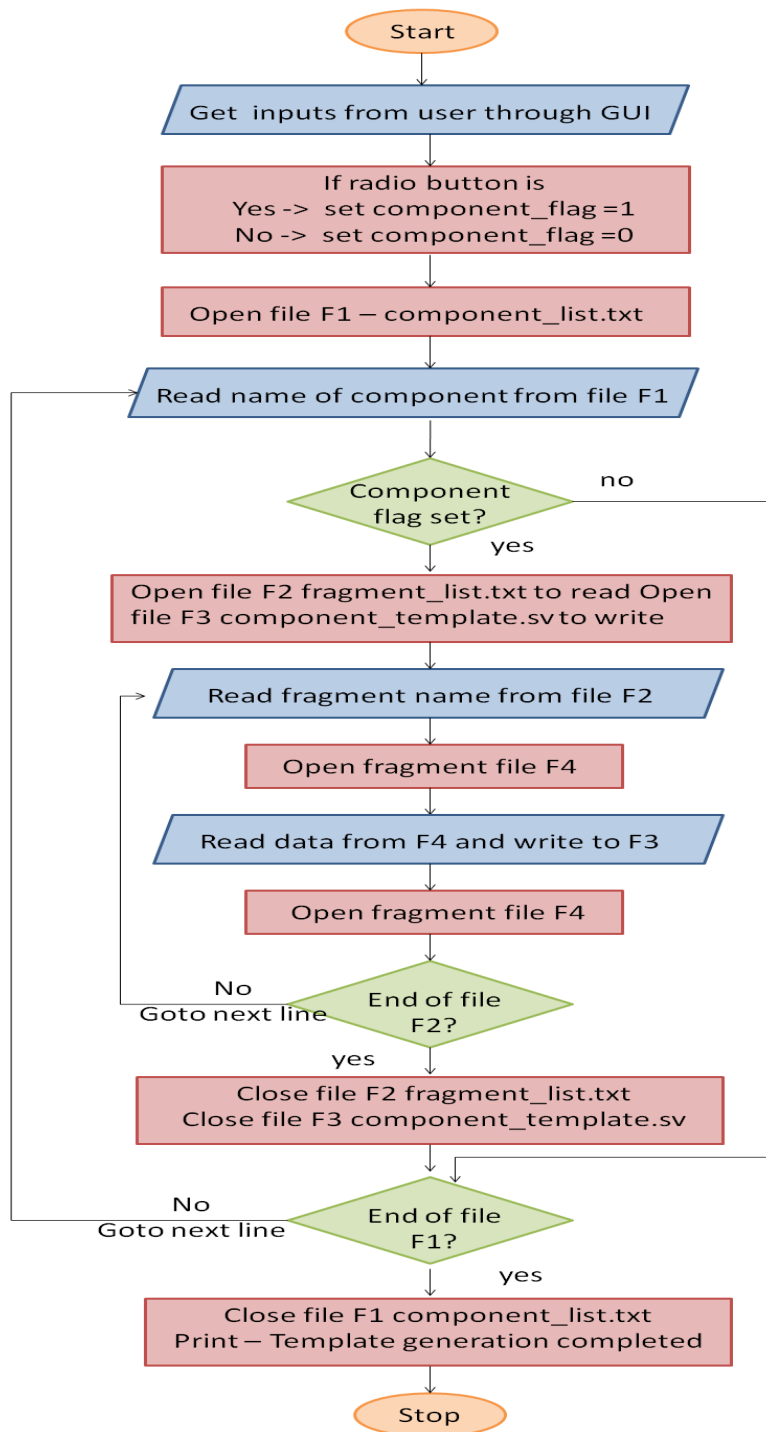


Figure 4.4: Flow chart



### 4.3 Summary

Actual work starting from designing of preliminary framework to creation of templates is presented in this chapter. Input to script was generated after studying various project, and full flow of script is presented.

# Chapter 5

## Designing Bus Interface Tracker

There are number of tools available to validation and design Engineers for debugging test failures. The most detailed information available in a Waveform viewer. However, it is very cumbersome to debug waveform by visual inspection. Bus, interface trackers are tabular representation of waveforms as shown in figure 5.1.

1						
2	TIME	data	addr	sel	re	bu
3					ad	rs
4					y	t
5						
6	=0=0=0=0=0=0=0=0=0=0=0=0=0=0=0=					
7	1 Reset activated					
8	=0=0=0=0=0=0=0=0=0=0=0=0=0=0=0=					
9	10000	0000_0000	0000_0000	0	0	0
10	20000					
11	=0=0=0=0=0=0=0=0=0=0=0=0=0=0=0=					
12	160001 Reset De-activated					
13	=0=0=0=0=0=0=0=0=0=0=0=0=0=0=0=					
14	1020000	b248_f635	8576_eb50			1
15	1610000	0000_0000	ff11_c480		3	
16	1620000	b248_f635	8576_eb50		0	
17	1650000	0000_0000	ff11_c400		3	
18	1660000	b248_f635	8576_eb50		0	
19	1750000			7		
20	1810000			0		
21	2200000	2804_b8ed	ff11_c480		3	
22	2210000	b248_f635	8576_eb50		0	
23	2220000					
24	2240000	0000_0000	ff11_c400		3	
25	2250000	b248_f635	8576_eb50		0	
26	2590000			1		
27	2680000			0		
28	3140000	0000_0000	ff11_c480		3	
29	3150000	0000_0000	0000_0000		0	
30	3180000	0000_0000	ff11_c400		3	
31	3190000	0000_0000	0000_0000		0	
32	3200000					
33	3680000			7		
34	3810000			0		
35	4190000					

Figure 5.1: Tracker output

Bus interface tracker is useful tool for debug and performance analysis. Tracker, monitors changes on an interface and dumps out in to a table as shown below. Information like timestamp, clk signal, reset signal, and various interface signals.

## 5.1 Template creation for tracker

Bus interface Trackers is not dependent on project specific details. As tracker monitors the changes in the interfaces, generic tracker template can be generated. I have developed Perl script that gets inputs from user through GUI and generate Ovm Component tracker.sv. as shown in fig 5.2. While doing actual simulation, this traker.sv file will gets input from interface file, and will generate tracker output as shown in fig 5.1. This interface file contains all the signal for that particular interface, using clocking block module. Interface is instantiated virtually in the ovm component tracker.

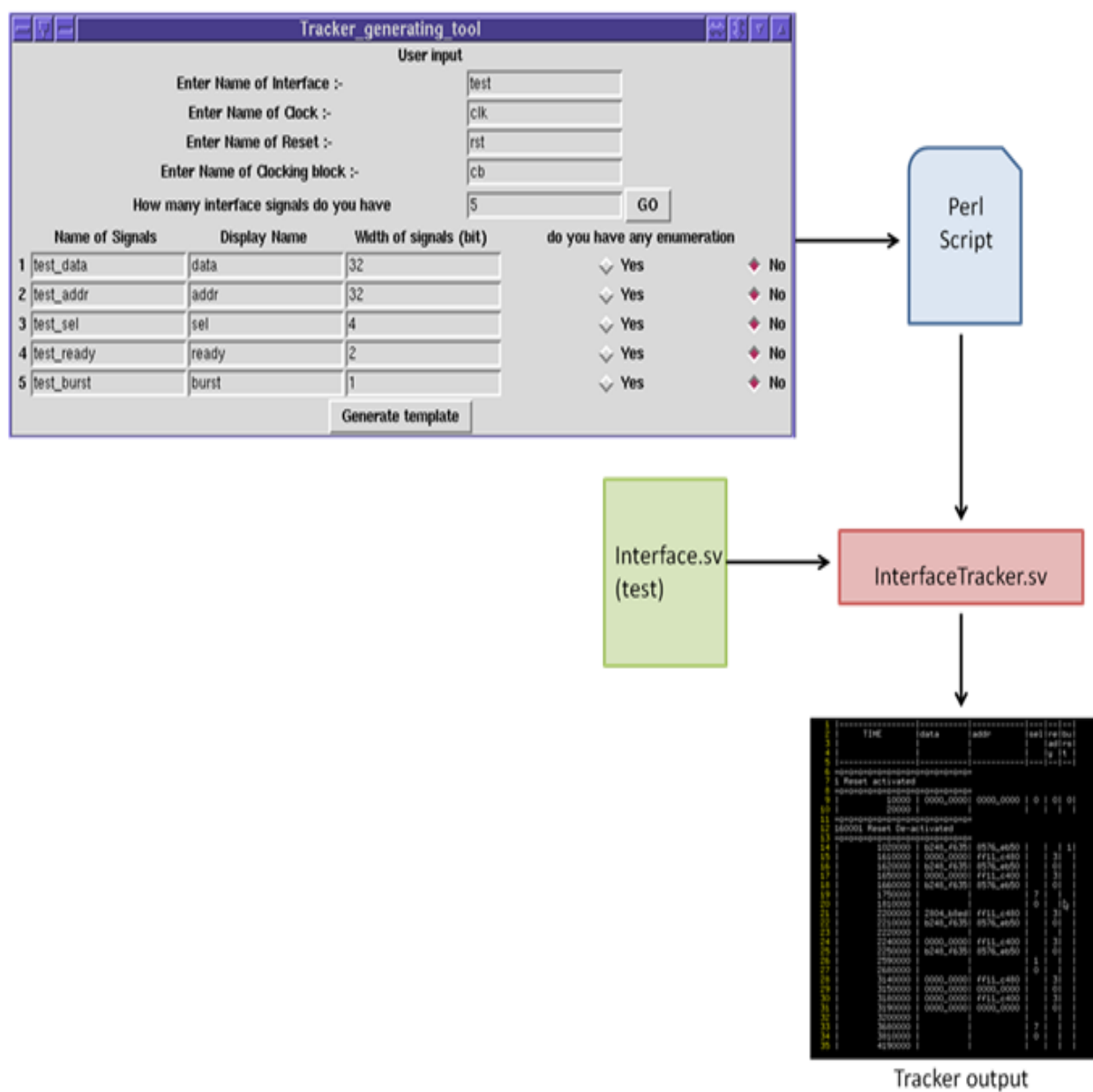


Figure 5.2: Tracker flow

Ovm Component Tracker includes interface instantiation, various OVM phases, tasks and functions shown in table - I :-

Interface instantiation	
Ovm phases	
new	
run	
report	
task	description
reset_value	reset all signal variable values to 'hx, signal_empty_str to and set the positioning of the signals in tracker output
print_header	print signal names with proper alignment
set_report_file	open file to write (tracker output)
watch_reset	print reset activated and deactivated at negedge and posedge respectively
capture	capture the current values of signal
start_track	start tracking whether any signal is changed or not
storelatestvalues	storing the current value to the variable
print_intf	print interface value when anything changes
function	description
ischange()	check whether any signal is changed or not
get_signal()	get current value of signal after change

Table I: Ovm component Tracker

### 5.1.1 Flow of script

In this script, initially get inputs from user through GUI as shown in fig 5.3. Name of interface for which tracker needs to be developed should be given to the script. Name of all the signals , with their width and display name in tracker output to be entered by user. With all this data, script will create ovm componet Tracker.sv file.

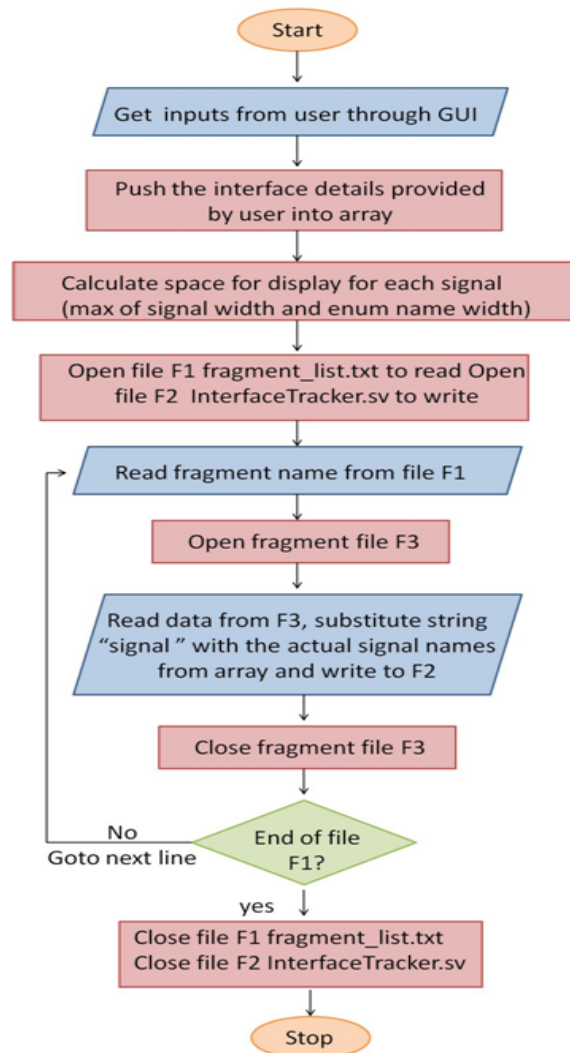


Figure 5.3: Tracker script flow

## 5.2 Summary

In this chapter, the concept of tracker is discussed. Actual work done for generating generic template for tracker is also described in this chapter. The inputs to the tracker generating script and flow of the script are presented.

# Chapter 6

## Other Productive Contribution

### 6.1 OVM Compliance checking for various IPs

Currently, verification environment for large number of reusable IPs are developed using OVM (Open Verification Methodology). For n number of IPs there are n numbers of IP owner, each of them having their own coding style, their own perception and understanding regarding methodology. There was no checking guidelines available for IPs to be checked, whether they are fully OVM compliant or not. Within Intel, one of the groups came up with ovm compliance checklist. This checklist constitutes various checklist rules that are to be followed by IP owner while designing verification environment such as architecture compliance (These checks require simulation to instantiate the OVC and check its architecture.). I have been involved in checking the OVM compliance for each existing IP with following steps:-

- Studied, understood and discussed the checklist rules that were defined in OVM compliance checklist.
- Developed input table to gather essential information from IP owner regarding testbench files for that particular IP.
- Graded one IP for this checklist item according to the inputs provided by IP owner. This grading process includes checking compliance for each checklist



item and also providing suggestion to make the non-complaint things to be complaint according to OVM compliance checklist.

- In the process of IP grading, we created an Excel sheet which includes following column

column number	column name	description
1	Checklist item	Description of that particular checklist rule
2	Priority	Priority of that checklist item either HIGH or MEDIUM
3	Grade	Grade of that particular checklist item depending on the priority
4	Name of file/s	Name of the file/s for that particular checklist item to be checked in testbench environment provided by owner as an input.
5	Content of file	Content of the files in column-4 which is to be checked for that checklist item.
6	Compliant	From the content of files, compliancy can be decided whether yes or no
7	Content to be modified	Content to be modified to make that file compliant
8	Grade obtained	if column-6 is <b>yes</b> then grade obtain is the same as column 3 if column-6 is <b>no</b> then grade obtain is zero

Table I: IP grading column

- After grading an IP ABC, I found that there are various tasks that can be automated, because various checklist items depend on searching particular content from the input files that IP owner provides. So I developed a Perl Script for automating the IP grading process.

- Output of the script is Excel sheet with the data for column 1 to column 5 for every checklist item. Rest of data is to be filled manually.
- Using this script, almost 80% of the manual task is automated.
- I graded total 14 Ips, the result is as follows

IP	current grades
IP1	40.00%
IP2	65.45%
IP3	43.64%
IP4	34.55%
IP5	54.55%
IP6	52.73%
IP7	81.82%
IP8	63.64%
IP9	70.91%
IP10	70.91%
IP11	70.91%
IP12	69.09%
IP13	47.27%
IP14	65.45%

Table II: ovm-ip-grading

## **6.2 Added Coverpoint for mapping multi interrupt agents to single interrupt pin or line**

All the interrupt agents are mapped to one of the pin of 8 pin interrupt line, there can be large number of agents, so more than one agent is mapped to single line, and send interrupt. To check whether multi interrupt agents are mapped to single interrupt pin or not, I added coverpoint in relevant covergroup of Coverage component. To add this coverpoint data was taken from scoreboard, and using that data at any instance of time, how many agents are mapped to a particular pin can be found out. Coverpoint was added to count the number of agent mapped should be more than 2.

## **6.3 Written various test code**

### **6.3.1 For checking full condition for memory storage element**

Message Signaled Interrupts, in PCI 2.2 and later and PCI Express, are an alternative way of generating an interrupt. Traditionally, a device has an interrupt pin which it asserts when it wants to interrupt the host CPU. While PCI Express does not have separate interrupt pins, it has special messages to allow it to emulate a pin assertion or deassertion. Message Signaled Interrupts allow the device to write a small amount of data to a special address in memory space. There was a test, in which one assert message is sent, and once msi (message signaled interrupts) arrives then de assert was sent. I have enhanced this test to check if, 2 interrupts are send then weather the memory storage element on the other side which receives this message gets full or not.

### **6.3.2 For checking clock dutycycle of certain block**

There was a test which was checking the clock period for a certain block. In this test, period from posedge to posedge was counted for particular frequency. I have

enhanced this test to count duty cycle, by taking posedge to negedge difference.

### 6.3.3 For multiple msi

There was a test for getting single msi (message signaled interrupts), I have written new test for getting multiple msi. My learning from this is, before receiving another msi, end of data for first one should be sent, that is once the first message is completed, and another can be received. I have used Clocking block feature of system verilog, in this test.

## 6.4 Trainings attended

During the period of internship, I have attended various trainings. These training helped me in understanding and developing different the concepts of System verilog and OVM.

- **Attended SV training and practices labs on**

- Basic Datatypes, enumerated Data type,
- Arrays, Structure
- Interfaces
- Classes and inheritance
- Static Class Properties and Copying Objects
- Parameterized Classes
- Virtual Classes
- Random Constraints
- Generate Construct

- **Attended OVM training and practices labs on**

- Testbench organization
- Defining Transaction
- Designed Driver and sequencer components
- Designed monitor and agent components to observe a bus transaction
- Define a covergroup within a coverage-collector to measure stimulus coverage.
- Create a user-defined sequence extending from `ovm_sequence`.
- Use the factory override feature to use a new driver component

- **Attended Advanced SystemVerilog Tips Including OVM & UVM**

**Tips** This training was presented by World-Renowned Verification Expert Cliff Cummings, hosted by Cadence Design Systems and QLogic India.

Topics Overview:-

- New UVM 1.0 overview and comparison to OVM
- Important OVM and UVM phasing
- Secrets in mastering OVM and UVM
- Graceful termination of tests in OVM and UVM with emphasis on the objection mechanism
- Some of Cliff's favorite SystemVerilog tips and tricks
- Some early UVM techniques and best practices

# Chapter 7

## Conclusion

The generic framework intends to improve the design of organizational processes for IP verification. The approach is to develop system verilog compile clean templates for all components needed in verification, with all basic necessary code including placeholders for project specific code according to OVM methodology. This project is divided into two distinct phases Designing and Validating. The first phase (Designing) involves creating templates for basic components according to OVM methodology of verification. Second phase (validating) involves compiling and elaborating the design by developing configuration files for generic frontend simulation environment.

Script was demonstrated to the Intel team, and no specific enhancement was mentioned, script will be shortly released for the deployment within some new IP projects. The script enables the immediate and easy use of framework which leads to lower costs, less time consumption and a higher quality of the Design for IP verification. Thus this project will save efforts as well as time for creating basic essential components in the environment and is less prone to errors as the framework itself would be validated.

## References

1. Ovm\_open-verification-methodology\_cookbook.pdf
2. perl\_tut.pdf
3. SystemVerilog\_3.1a.pdf
4. Janick Bergeron, Writing Testbenches: Functional Verification of HDL Models, Second edition, Kluwer Academic Publishers, 2003.