# The STIL generator is a uvm\_subscriber extended component which collects STIL information from drivers through analysis ports and writes them out according to time stamp. To get STIL information, it’s necessary to let drivers take charge of all pad toggle, no any direct pad connection in test bench. Pads are divided into five categories which correspond to jtag\_driver, clock\_diver, reset\_driver, scan\_chain\_driver and pad\_driver. Since clock pads and reset pads connection are usually simple, they get toggle information from configurations directly others are from sequecne\_items.

# Each type of pads has a specific driver extends from uvm\_driver to interact with them. In Figure1, jtag\_driver drives JTAG interface, scan\_driver drives scan interface, clock\_driver drivers clock pads, reset\_driver drivers reset pads and pad\_drivers drives other pads.

# Firstly, stil\_info\_transaction sequence item is defined as Figure 2.

# class stil\_info\_transaction extends uvm\_sequence\_item;

# string stil\_info;

# string comment\_info;

# ……

# endcalss

# stil\_info is the pad drive or sample information at a certain time. For example,

# Secondly, add an analysis port, which is an object of uvm\_analysis\_port class specialized with stil\_info\_transaction type, in each driver, except for scan\_driver. Scan test pattern generation, which are normally generated by ATPG tools, is not the topic of this paper.

# Thirdly, a component named stil\_generator, which extends from uvm\_subscriber class specialized with stil\_info\_transaction type, is created. Add correspondent analysis export for each driver in stil\_generator. Since uvm\_subscriber class has only one built-in analysis export, the `uvm\_analysis\_imp\_decl macro need to be used to declare analysis imp export and its associated write() method for the rest analysis export[2].

# Lastly, connect analysis port and analysis export pair as show in Figure 1 in orange lines.

### Abstract

# The DFT design is becoming more and more complex accompany with the scale increasing of SoC. How to verify DFT logic completely in simulation and supply test patterns to ATE test with highly coverage is important for post-silicon debug and yield increase.

# While verification methodology is evolving, innovating and entering UVM era, DFT verification need to keep in pace to leverage the advantage of UVM, therefore to increase test reusability, extendibility and function coverage and etc.

# This paper presents a general UVM based DFT verification environment, which is usable from modular DFT verification to SoC DFT verification, and it can generate STIL test patterns for ATE test during SoC simulation.

# This paper also presents a method to model hierarchically networked DFT TDR in register abstract level to let test writers focus on test sequences not caring the details of TDR read and write.

### Introduction

# In DFT (Design For Testability) domain, the test patterns running on an ATE (Automatic Test Equipment) can be categorized into two types: scan related and non-scan related. The former can be generated using ATPG (Automatic Test Pattern Generation) tools, while the latter cannot. These non-scan DFT function tests, like other function tests, are normally created by design verification engineers using languages such as System Verilog or C++. However, ATEs need test patterns described by STIL (Standard Test Interface Language) or other test languages.

# To fill the gap, there is usually a dedicated team to transfer function simulation to ATE test environment, or alternatively in-house automation flows are developed to enforce complex rules on test writing and register specification documentation, which are specific for a given environment and difficult to migrate.

# This paper provides a universal and more efficient solution by introducing a UVM (Universal Verification Methodology) based DFT verification environment that naturally generates test patterns in STIL format during simulation and can be plugged into any UVM based environment. This method applies to other formats that ATEs need.

# For ultra-large-scale SoC (System on Chip), IEEE 1149 protocol alone cannot satisfy DFT design requirements, so IEEE 1687 and 1500 protocols are usually adopted to enable modular and hierarchical DFT test access, leading to challenges when writing test sequences at register abstract level, as different protocol TDRs (Test Data Register) are hierarchically located in a network connected via IEEE 1687. To access a TDR one or more levels 1687 SIBs (Segment Insertion Bit) have to be set and the length of DR (Data Register) chain varies with SIB values. The author also comes up with a general way to model complex DFT test access network.

# …point issues need to resolve…

# … review previous work…

# … Important to resolve these issues…

# … What this paper did …

### Proceed of This Paper

# This paper is intended to divide into two parts. The first part is about how to build a UVM based DFT verification environment that can generate STIL test patterns innately. The second part is focus on how to lift DFT TDR to register abstract level. The third part is how to verify the generated STIL pattern is workable. The fourth part is about result discussion and conclusion.

# In the first and second part, each of them will be proceeded as following sections.

# Idea overview.

# Implementation details.

### UVM Based DFT Verification Environment

#### Idea Overview

# The STIL test pattern describes test stimulus using vectors which specify pad toggle and measurement information (called STIL information hereinafter) in a time period.

# A UVM test usually contains one or several sequences; the UVM sequences are finally break down into streams of UVM sequence items (a.k.a transactions) and passes to UVM drivers. UVM drivers are in born the best supplier of STIL information.

# If we restrict any pad drive and sample should be controlled through a UVM driver which enforce no directly pad connection in test bench, collect all STIL information from drivers and then write them out according to the time stamp of STIL information, we can get a complete test vectors of a certain UVM test after simulation finish.

# So we divide pads of a SoC as following groups for DFT functional simulation.

# IEEE 1149.1 compliance on-chip TAP (Test Access Port). Hereinafter it’s simply called JTAG (Joint Test Action Group) interface, which is the most significant interface for DFT design.

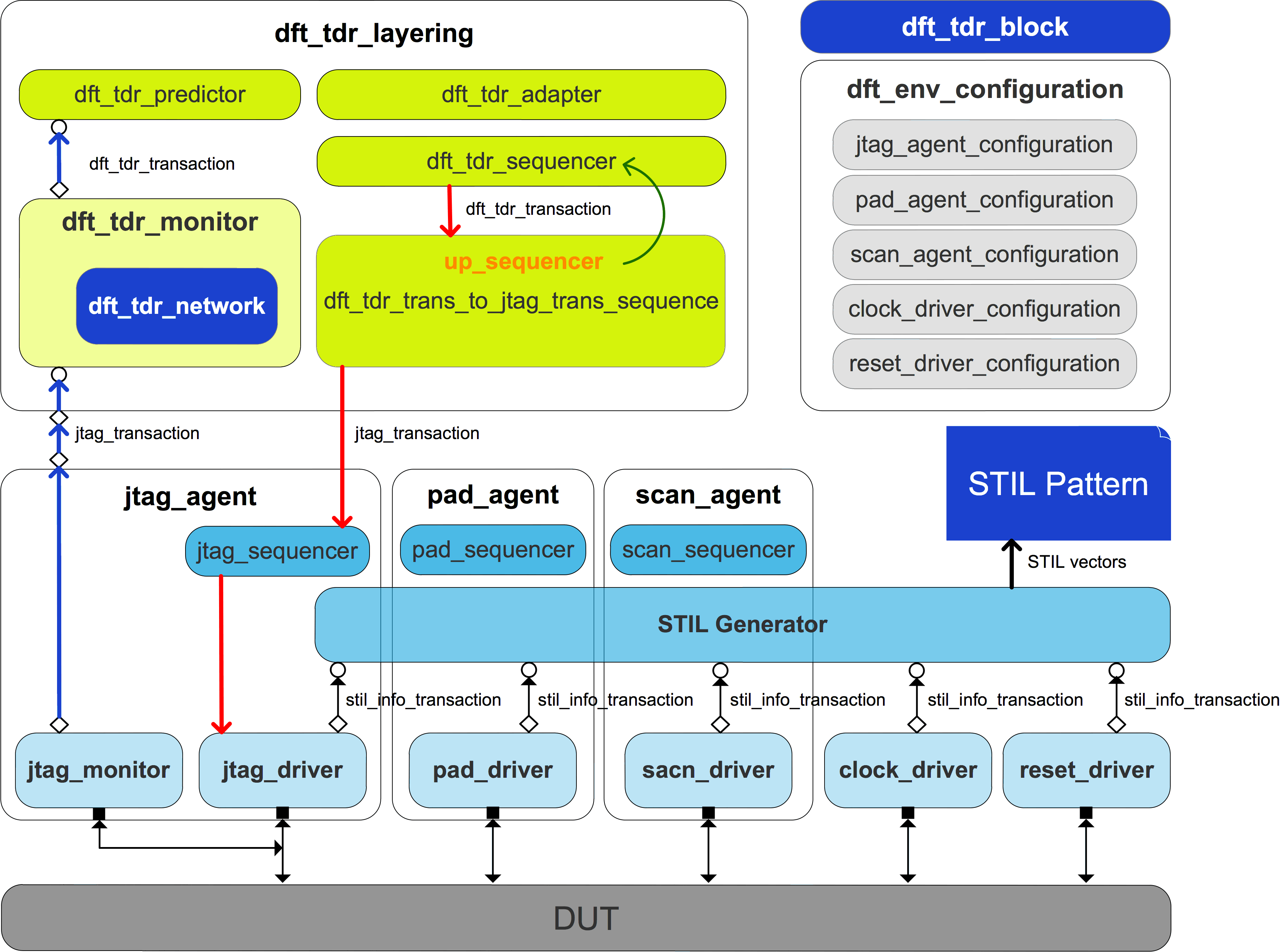
# Scan related pads. Hereinafter it’s simply called scan interface, which is a group of pads used as scan control signals and scan channels when a chip is in scan test mode.

# Clock pads. It’s the clocks need toggling in DFT functional simulation. Please see BOZO for more description.

# Reset pads. It’s the asynchronous resets of a chip.

# Other pads. Except for type 1-4 mentioned above, the rest pads are categorized as one type. These pads may need have initial value or be toggled once in a while in simulation.

# In Figure 1, jtag\_driver, scan\_driver, clock\_driver, reset\_driver and pad\_driver are correspond to above pad group from 1 to 5.The STIL generator collects STIL information from these drivers and writes them out.



### jtag\_agent Implementation

##### jtag\_agent\_configuration

##### jtag\_transaction class

o\_ir is a dynamic array to store instruction operation code(a.k.a OPCODE) to be sent to DUT JTAG 1149.1 FSM IR(Instruction Register).

o\_dr is a dynamic array to store data to be sent to DUT’s IEEE 1149.1 compliance FSM DR(Data Register).

chk\_ir\_tdo and chk\_dr\_tdo are flags to indicate jtag\_driver whether to check TDO during shift IR state or shift DR state.

##### jtag\_driver class

In jtag\_driver, IEEE 1149.1 protocol is implemented. It fetches jtag\_transaction sequence items from jtag\_sequencer and toggle the JTAG interface.

If the gen\_stil\_file knob is on, jtag\_driver writes

### DFT TDR Abstraction

#### Idea Overview

### STIL Test Pattern Verification

### Conclusion