Bringing Constrained Random into SoC SW-driven Verification

Alberto Allara
Digital Asic Product
STMicroelectronics
Via Tolomeo 1, Cornaredo (MI), Italy
alberto.allara@st.com

Fabio Brognara
Digital Asic Product
STMicroelectronics
Via Tolomeo 1, Cornaredo (MI), Italy
fabio.brognara@st.com

Abstract—**one of the most used techniques for System-on-Chip (SoC) verification is the so-called SW-driven verification.One drawback of such technique is that the tests developed are in general direct tests. This could be considered a severe limitation since direct tests do not scale very well when the test complexity increases compared to a more modern Coverage Driven Constrained Random Verification (CDCRV). In this paper we propose a lightweight technique that takes advantage of systemVerilog UVM methodology to introduce CDCRV in SW-driven tests.**

Keywords—SoC, Coverage Driven Constrained Random, CDCRV, SW-driven verification,systemVerilog,UVM, VAL

I. Introduction

One of the most used techniques for System-on-Chip (SoC) verification is the so-called SW-driven verification. It consists in taking advantage of the CPU(s) on board of the system to execute software (typically written in C) to test the correct integration of all the IPs embedded in the system or to develop performance and stress-tests directly on the actual system. The SW-driven verification methodology is undoubtedly a powerful technique. Among its advantages there is the easiness of use and, potentially, the possibility to reuse the tests developed for RTL simulation in the context of the silicon validation on emulators. One of its drawbacks is that the tests developed with this technique are in general direct tests. This could be considered a severe limitation since direct tests do not scale very well when the tests complexity increases compared to a more modern Constrained-Random verification methodology. The EDA market is starting to address such limitation with a new class of powerful tools adopting new techniques identified as *Model-based test generation* (see for instance TrekSoc© from BrekerSystems[1] or Infact© from Mentor Graphics[2]) that allow to define a layer on top of the standard SW-driven SoC tests where the user can formulate the randomization of an entire class of scenarios. In this paper we propose a lightweight technique that takes advantage of SystemVerilog (SV) UVM methodology [3] to introduce Coverage Driven Constrained Random techniques in SW-driven tests.

II. Verification Abstraction Layer

The basic interface between embedded processors and internal IPs (e.g., peripheral devices) in a SoC is composed of a set of control and status registers. These registers, that are usually located in the memory space of the system (memory mapped), are part of the IP implementation and represent features of the IP itself. We could think that any external verification IP (VIP) used for the verification of a SoC could be interfaced to an embedded processor through a set of control and status registers in the same way we do with hardware IPs. The problem is that usually the VIPs available do not come with such an interface and, in general, no standards are available.

We closed this gap with a specific layer, called verification abstraction layer (or VAL) dedicated to expose the functionality of the VIPs (e.g., VIP configuration or VIP sequences) in the form of a register map suitable to be controlled by an embedded processor. The VAL is implemented with a structure of verification components as reported in Figure 1. In particular, it is composed of a front-end (VAL-FE) component, specific of the physical interface where it is connected, a bridge component to convert TLM packets generated by the front-end into internal VAL packets and a set of VAL back-end components (VAL-BE) each one devoted to interface a specific VIP or to provide dedicated randomization capabilities. In the UVM terminology the VAL front-end is a monitor and is typically connected at SoC-level to the ports of an embedded static RAM (Figure 2). The embedded RAMs generally guarantees a fast access from CPU and their interface is available both at RTL and gate-level.

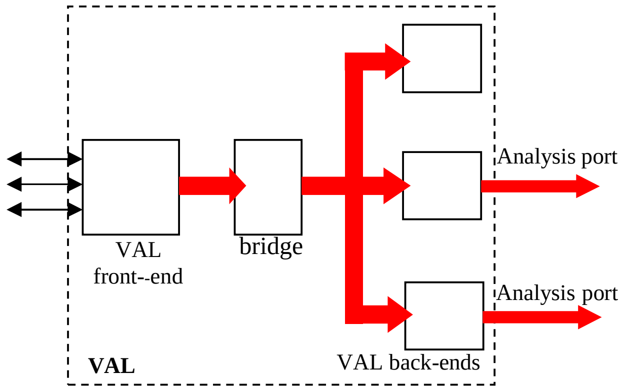

Figure 1 VAL structure

The VAL bridge converts a VAL-FE-specific memory packet generated by the VAL-FE and carrying an address, a data value, and a direction into a VAL-specific memory packet propagating the same information. This bridge enforces the reusability of VAL-BE components in case of change of the VAL-FE.

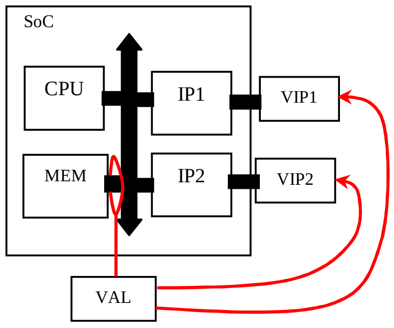

Figure 2 VAL in a SoC

Each memory transaction flowing through the interface where the VAL is connected is captured and forwarded to all the internal VAL-BE components.

Each VAL-BE is associated with a dedicated register map used to enable the interaction between

1. the embedded CPU and

2. either the verification component connected through the VAL-BE or directly the VAL-BE itself.

The register maps are only virtual since they are represented as a data structure mapped in the internal memory of the SoC. Their general layout is reported in Figure 3.

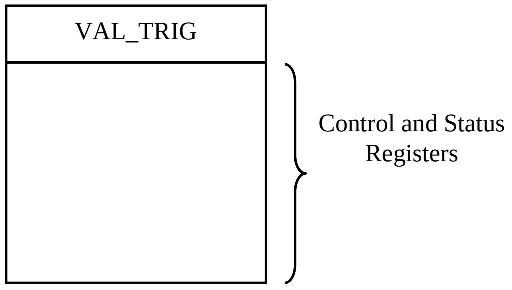

Figure 3 generic VAL-BE register map layout

The first register of the register map is called VAL_TRIG. Each VAL-BE can be programmed to be sensitive to the particular memory location where the VAL_TRIG is located. Every time the SW access the address location of the VAL_TRIG, the data contained in this location is used as a selector to trigger different actions that the associated VAL-BE component can execute. Beside VAL_TRIG the other status and control registers can be used to exchange information between SW and verification environment. For performance reasons, all the data transfers are executed through back-door memory accesses.

Each VAL-BE is the specialization of a UVM SV root class called *val_component* and inherits the following features:

- Configuration knobs: they define the memory address the component is sensitive to, and the base memory address of the memory where the VAL-BE is mapped. Moreover they define the connection between SoC memories and the backdoor API functions
- Memory backdoor API functions: define the common functions to read or write data into memory through backdoor accesses. They include:
  - o read_mem_word(bit[31:0] addr, output bit[31:0] data);
  - o read_mem_dword(bit[31:0] addr, output bit[63:0] data);
  - o write_mem_word(bit[31:0] addr, input bit[31:0] data);
  - o write_mem_dword(bit[31:0] addr, input bit[63:0] data);
- The val_op() callback function that needs to be implemented to specify the behavior of the VAL-BE component

III. VAL BACKEND DESIGN

The design of a VAL-BE consists in the definition of a register map from the SW side and a consistent implementation of a UVM component from the HVL side. To better clarify the required steps for such a process let us consider a VAL-BE called VAL_TUBE. This component implements the UVM reporting mechanisms as well as the end-of-simulation mechanism, both issued by SW.

Beside the VAL_TRIG the register map for this component needs to reserve a space to store the string of messages the SW wants to print in the log file of the SoC simulation. In this register map there are only registers written by SW and used by the verification component.

Figure 4 reports the data structure in C code for this register map

```
typedef struct _val_tube_registers {
  uint32_t VAL_TRIG;
  uint32_t STR_BUFF[50];
} val_tube_registers;
```

Figure 4 data structure of VAL_TUBE register map

A user does not interact with the above register map directly but through a set of API functions, in particular:

- void uvm_info(char* msg);
- void uvm_warning(char* msg);
- void uvm_error(char* msg);
- void global_stop_request(void);

The first three API functions store the string of characters inside the STR_BUFF buffer and then write into the VAL_TRIG a value that represents the severity of the message (0=info, 1=warning, and 2=error respectively). The VAL-BE component will use such information to correctly print the message in the log file

The latter API function stores into the VAL_TRIG a code (in our example the value 4) that instructs the VAL-BE to stop the simulation when it is called.

The complete low-level driver in C for the VAL_TUBE is shown in Figure 5. Please notice that each reporting function (respectively uvm_info, uvm_warning and uvm_error) calls the uvm_report function providing the appropriate severity value. Inside uvm_report the p pointer to the VAL_TUBE register map is initialized with a macro (VAL_TUBE_START) that can be customized according to the SoC project needs.

```
#ifndef VAL_TUBE_START
#define VAL_TUBE_START 0x D8000000
#endif
typedef enum _uvm_severity {
  UVM_INFO =0,
  UVM_WARNING,
  UVM_ERROR,
  UVM_QUIT
} uvm_severity;
...
void uvm_error(const char* str) {
  uvm_report(str,(uint32_t)UVM_ERROR);
}
void uvm_warning(const char* str) {
  uvm_report(str,(uint32_t)UVM_WARNING);
}
void uvm_info(const char* str) {
  uvm_report(str,(uint32_t)UVM_INFO);
}
void uvm_report(const char* str,uint32_t sev) {
  volatile val_tube_registers* p=(volatile
val_tube_registers*)VAL_TUBE_START;

  strcpy((char*)p->STR_BUFF,str);
  *(p->VAL_TRIG) = sev;
}
void global_stop_request(void) {
  volatile val_tube_registers* p=(volatile
val_tube_registers*)VAL_TUBE_START;
  *(p->VAL_TRIG) = (uint32_t)UVM_QUIT;
}
```

Figure 5 low-level driver for the VAL_TUBE

From the HVL-side the steps needed to develop a VAL-BE component are:

- creation of VAL_TUBE as specialization of the val_component class
- Definition of the val_op() function implementing the behavior consistently with the previously defined register map
- instantiation of the VAL-BE inside the environment and connection to the internal analysis port of the VAL bridge and (if needed) to other external verification IPs

Figure 6 reports an example of implementation of the VAL_TUBE. Defined in the lower portion of the Figure the *read_string_from_mem* function is used to read the content of the STR_BUFF with the *read_from_mem* API function provided by the val_component class. The input parameter of the read_string_from_mem function is the base address memory location where the STR_BUFF buffer is mapped expressed relatively to the memory

```
class val_tube extends val_component;
  string s;

  `uvm_component_utils(val_tube)

  function new(string name = "val_tube",
               uvm_component parent);
    super.new(name, parent);
  endfunction

  virtual function void val_op();
    if (VAL_TRIG == 3) begin
     `uvm_info("VAL_TUBE",
     "stop activated from SW",UVM_NONE);
     uvm_test_done.drop_objection(this,
       "Stop triggered by SW");
    end
    else begin
      s=read_string_from_mem(
         (mem_addr - base_mem_addr ) + 4
        );
      case(VAL_TRIG)
        0:
        `uvm_info(“VAL_TUBE”,s,UVM_NONE);
        1: `uvm_warning(“VAL_TUBE”,s);
        2: `uvm_error(“VAL_TUBE”,s);
      endcase
    end
  endfunction : val_op

  virtual function string read_string_from_mem
                          (input bit[31:0] a);
    bit complete =0;
    bit[7:0] ca;
    bit[31:0] w;
    string s1;
    string s ="";
    for (reg[31:0] i=a;
         (i < (a+200)) && !complete;
         i+=4) begin
      read_mem_word(i, w);
      for (int j=0; j < 4; j++) begin
        ca = w[7:0];
        w = {8'h0,w[31:8]};
        if (ca == 8'h00) begin
          complete = 1;
          break;
        end
        else begin
          $sformat(s1,"%c",ca);
          s = {s,s1};
        end
      end
    end
    return s;
  endfunction : read_string_from_mem
endclass : val_tube
```

Figure 6 example of VAL_TUBE implementation

IV. SW DATA RANDOMIZATION THROUGH VAL

The VAL is the key to introduce randomization technique in the embedded SW. Indeed, it is possible to reserve VAL-BEs

to trigger the randomization of SystemVerilog (SV) packets of data (uvm_object) defined inside the components itself. Then, the randomized data can be possibly sent to external verification components through, for instance, a UVM analysis port or be serialized into a SoC memory using back-door memory access mechanisms.
In order to explain in detail the whole mechanism considers the following example. We want to verify the correct integration of a UART IP within a SoC using tests that randomize its configuration, and in particular, the following features:

- Baud rate (e.g. 9300,28800,57600,..)
- Number of stop bits (0, 1 or 2)
- Parity enabled/disabled (0 or 1)
- Size of the sent character (7bit or 8bit)

Figure 7 shows an example of UART configuration coded in UVM

```
typedef enum {
  baud9600,
  baud28800,
  baud57600
} baud_enum;

class uart_cfg extends uvm_object;
  rand baud_enum baudr_id;
  rand bit[1:0] nstopb;
  rand bit parity_en;
  rand bit char_len;

  `uvm_object_utils_begin(uart_cfg)
    `uvm_field_int(baudr_id,UVM_ALL_ON)
    `uvm_field_int(nstopb,UVM_ALL_ON)
    `uvm_field_int(parity_en,UVM_ALL_ON)
    `uvm_field_int(char_len,UVM_ALL_ON)
  `uvm_object_utils_end
  function new(string name = "uart_cfg");
    super.new(name);
  endfunction
  //~~~ Constraints
  constraint baud_rate_c {
     baudr_id inside {[baud9600:baud57600]};
  }
  constraint num_stop_bits_c {
    nstopb inside {[0:2]};
  }
endclass: uart_cfg
```

Figure 7 UART configuration in UVM

We can define a VAL-BE register map associated to the above configuration class as follows (see Figure 8):

```
typedef union {
  struct {
    uint32_t f_NSTOPB   :2;  // [0:1]
    uint32_t f_PARITYEN :1;  // [2]
    uint32_t f_CHARLEN  :1;  // [3]
    uint32_t unused_0   :28; // [4:31]
  } BitI;
  uint32_t RegI;
} UARTMISC_t;

typedef struct _uart_cfg_registers {
  uint32_t   VAL_TRIG;       // @0
  uint32_t   BAUDR_ID_REG;   // @4
  UARTMISC_t MISC_REG;       // @8
} uart_cfg_registers;
```

Figure 8 register map of the UART VAL

The register map is composed by 3 registers:

1. The VAL_TRIG used to trigger the VAL-BE as we have seen in previous paragraphs
2. The baud rate register identifier called BAUDR_ID_REG (notice that it spans a full 32bit word) and finally
3. a Miscellaneous register called MISC_REG that contains as subfields the number of stop bits (f_NSTOPB), the parity enable/disable (f_PARITYEN) and the length of the transmitted character (f_CHARLEN) respectively.

The MISC_REG register is described with a union/struct statement that allow to access either to the register subfields (BitI) or directly to the register as a whole (RegI). For SW compilers that do not handle this representation correctly it is possible to represent MISC_REG with a normal 32bit word and to define a set of bit field masks to extract their internal fields.
Figure 9 shows the low-level driver of the VAL–BE dedicated to the UART:

```
typedef struct _uart_val_desc {
  volatile uart_cfg_registers* regmap;
} uart_val_desc;
void uart_val_init(uart_val_desc* p,
                   uint32_t base_addr) {
  p->regmap = (volatile
uart_cfg_registers*)base_addr;
}
void uart_val_randomize(uart_val_desc* p) {
  (p->regmap->VAL_TRIG) = 0;
}
void uart_val_setparity(uart_val_desc* p,
                        int en) {
  (p->regmap->MISC_REG.BitI.f_PARITYEN = en;
  (p->regmap->VAL_TRIG) = 1;}
```

Figure 9 low-level driver of the UART VAL

We defined in particular 3 API functions used to initialize the register map of the VAL declared within the descriptor of the driver, to randomize the configuration of the UART, and to enable/disable the parity of the UART directly from SW respectively. We have reserved for the VAL_TRIG the value 0 to trigger the randomization and the value 1 to trigger the propagation of the parity enable value to the VAL and subsequently to the VIP.
From the HVL side we have to implement the VAL-BE taking into account the several aspects defined above. Figure 10 shows an example of implementation of a VAL-BE component dedicated to randomize UART configurations

```
class val_uart_cfg extends val_component;
  uvm_analysis_port #(uart_cfg) cfg_port;
  uart_cfg cfgp;
  reg [31:0] a;
  reg [31:0] w;
```

```
  `uvm_component_utils_begin(val_uart_cfg)
    `uvm_field_object(cfgp,UVM_ALL_ON)
  `uvm_component_utils_end
  function new(string name = "val_uart_cfg",
               uvm_component parent);
    super.new(name, parent);
    cfg_port = new("cfg_port", this);
    cfgp = uart_cfg::type_id::create("cfgp",
                                   this);
  endfunction
  ...
  virtual function void val_op();
    case(VAL_TRIG)
      0: begin
       // Randomize the configuration packet
       assert(cfgp.randomize()) else
       `uvm_fatal("VAL_UART",
                  "Randomize failed");
       // Serialize back into memory
       a = (mem_addr - base_mem_addr ) + 4;
       // access to register BAUDR_ID_REG
       w = cfgp.baudr_id;
       write_mem_word(a,w);
       a += 4;  // access to register MISC_REG
       w = {28'b0,cfgp.char_len,
            cfgp.parity_en,cfg.baudr_id};
       write_mem_word(a,w);
      end
      1: begin
       // Read the parity information from Mem
       a = (mem_addr - base_mem_addr ) + 8;
       // access to register MISC_REG
       read_mem_word(a,w);
       cfgp.parity_en = w[2];
      end
    endcase
    // Send the UART configuration  to the VIP
    cfg_port.write(cfgp);
  endfunction: val_op

endclss: val_uart_cfg
```

Figure 10 example of UART specific VAL

In the example the val_uart_cfg component defines a UART configuration object (cfgp) and an analysis port used to propagate the configuration outside to a UART VIP. The component redefines the val_op() callback implementing two different actions: the randomization of the UART configuration and the update of the parity_en field of the configuration object with the content of the f_PARITYEN field written by SW. In both cases the configuration object is issued outside through the analysis port.

It is important to pay attention to the way the parameters of the UART configuration object are serialized. They must be written in the exact position defined by the SW register map (defined in Figure 8).

V. Constraining the Randomization in the VAL

The results of the randomization can be controlled mainly in two ways.The former is the one we could define static. In this case we rely on the UVM factory mechanism to replace one object subject to randomization within the VAL-BE with a specialized version of the same object that add or remove some constraints. The latter way is more dynamic and relies on values passed from SW to control the randomization constraints

Let us assume that we want to limit the UART configurations to those having the parity bit always disabled. The new configuration objects will become:

```
class uart_cfg_no_parity extends uart_cfg;
  `uvm_object_utils(uart_cfg_no_parity)

  function new(string name =
               "uart_cfg_no_parity");
    super.new(name);
  endfunction
  //~~~ Constraints
  constraint no_parity_c { parity_en == 0;}
endclass: uart_cfg_no_parity
```

Figure 11 UART configuration with disabled parity

The replacement of the original configuration object with the new one can be decided at UVM test level.

```
class uart_no_parity_test extends
uart_test;

`uvm_component_utils_begin(uart_no_parity_test
)

  function new(string name =
               "uart_no_parity_test",
               uvm_component parent);
    super.new(name, parent);
  endfunction

  function void buil_phase(uvm_phase phase);
    set_type_override_by_type(
      uart_cfg::get_type(),
      uart_cfg_no_parity::get_type()
    );
    super.build_phase(phase);
  endfunction : build_phase
```

Figure 12 UVM test that define a new UART configuration

The dynamic mechanism requires to reserve extra registers in the register map of the VAL-BE, to provide information that the component can use to constraint the result of the randomization. For instance, if we assume to have the possibility to control the maximum number of stop bit, the register map and the API function of the low-level driver should be restructured as follows:

```
typedef struct _uart_cfg_registers {
  uint32_t   VAL_TRIG;       // @0
  uint32_t   BAUDR_ID_REG;   // @4
  UARTMISC_t MISC_REG;       // @8
  uart32_t   MAX_STOPBIT     // @C
} uart_cfg_registers;

...
void uart_val_randomize_with(uart_val_desc* p,
int max_stopbit) {
(p->regmap->MAX_STOPBIT = max_stopbit;
(p->regmap->VAL_TRIG) = 0;
}
```

Figure 13 new UART VAL register map

In similar way also for code that randomizes the cfgp packet at HVL side is restructured. In particular, the content of the

MAX_STOPBIT register is read before to decide what to randomize. If the value of the register is 0 this constraint is ignored and the UART configuration is randomized as usual, otherwise an extra constraint that define the MAX_STOPBIT as upper bound it is applied.

```
class val_uart_cfg extends val_component;
  int max_stopbits;
  ...
  virtual function void val_op();
    case(VAL_TRIG)
      0: begin
      // Read the max number of
      // stop bit defined by SW
      a = (mem_addr - base_mem_addr ) + 12;
      // access to register MAX_STOPBIT
      read_from_mem(a,w);
      max_stopbits = w;
      if (max_stopbits == 0) begin
        // This additional constraint
        // is not used
        assert(cfgp.randomize) else
        `uvm_fatal(“VAL_UART”,
                   “Randomize failed”);
      end
      else begin
        // Randomize the configuration packet
        assert(cfgp.randomize() with
            {nstopb < max_stopbits;})
        else
          `uvm_fatal(“VAL_UART”,
                     “Randomize failed”);
      end
      // Serialize back into memory
      a = (mem_addr - base_mem_addr ) + 4;
      // access to register BAUDR_ID_REG
      w = cfgp.baudr_id;
      write_mem_word(a,w);
      a += 4;
      // access to register MISC_REG
      w = {28’b0,cfgp.char_len,
           cfgp.parity_en,
           cfg.baudr_id};
      write_mem_word(a,w);
    end
  ...
```

Figure 14 dynamic randomization mechanism at HVL side

VI. VAL CONNECTIVITY

The VAL allows to logically connect the embedded SW of a SoC to all the elements of an external verification environment. This feature is obtained, as we have seen in Figure 2, through a physical connection of the VAL-FE with an embedded memory. Externally, the VAL can be connected to the verification environment components using mainly two mechanisms: UVM analysis ports and SV interfaces.

The TLM connection implemented through analysis ports are typically adopted to propagate:

- Configurations to UVM agents
- Data to scoreboard components for data checking
- UVM objects to control sequences generated by UVM components

FIGURE 15 depicts a scenario that shows all the described possibilities

On the contrary, the connection through SV interface is used when there is the need to drive (or read) few physical signals that do not justify the implementation of a full fledge verification component, or to drive (or read) a legacy HDL-based verification component (e.g. , verilog BFM) that cannot interact with UVM specific TLM connections.

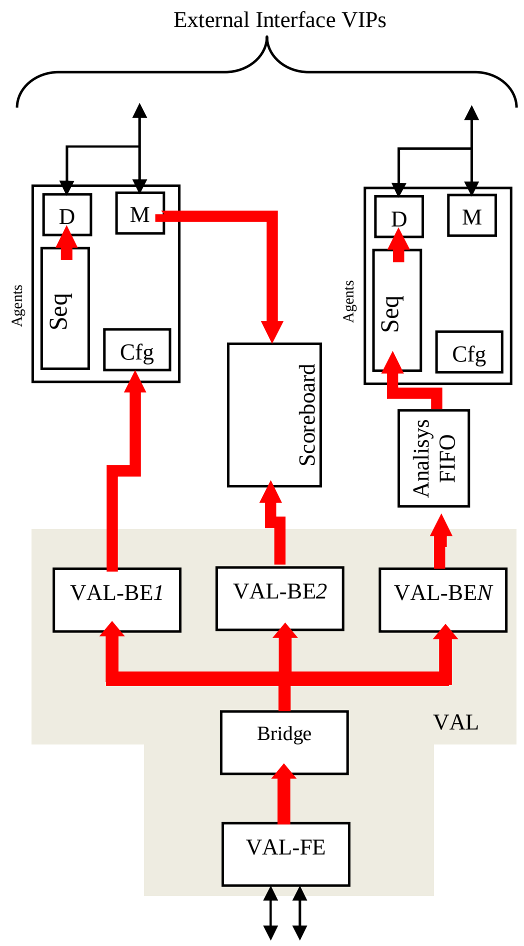

Figure 15 VAL-BE connected to UVM verification components through analysis ports

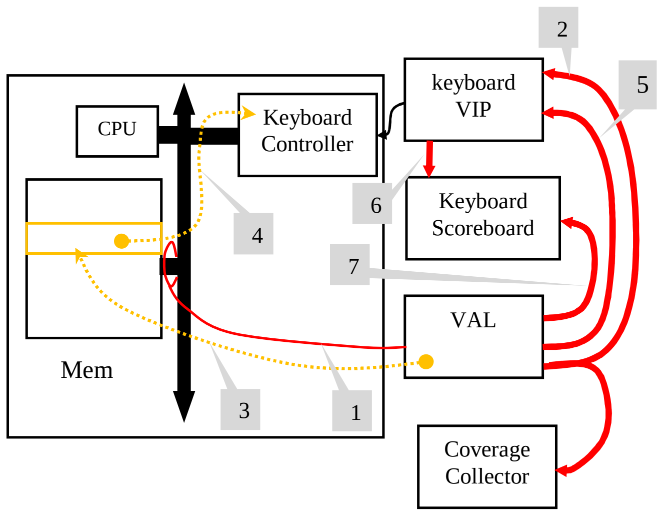

Figure 16 usage model with unidirectional (Rx) IPs

VII. EXAMPLES OF VAL USAGE MODEL

Considering the characteristics of the SoC IPs, we developed a set of VAL-centric verification patterns with the aim to guide our designers and verification engineers to be more efficient in the SoC tests development. In particular we have identified verification patterns for Bi-directional IPs, Unidirectional IPs (both Rx and Tx), Embedded RAM and ROM, DDR memories, Co-processors, Miscellaneous registers driving side-band signals, Reset and Clock generators and so on.
In this paragraph we present a couple of cases as example.
As a first example consider the case where a Designer/Verification Engineer needs to implement tests to verify the correct integration of a keyboard controller within a SoC (a typical example of a Unidirectional Rx IP). The scenario is depicted in Figure 16.

In this case the test starts triggering the randomization of the configuration of the external VIP through VAL (1). The result of the randomization is sent to the VIP over an analysis port (2) and stored back in memory through backdoor memory access (3). The analysis port is connected also to an external coverage collector component to measure functional coverage. The C test uses the result of the randomization stored in memory to configure in a consistent way the keyboard controller (4). Once IP and VIP are both configured, the SW test triggers a new operation of the VAL to instruct the VIP to simulate the pressing of a key (5) and then do polling on a status variable of the keyboard controller driver waiting for a key pressed event. The VIP generates a key pressing and sends the high-level packet to a scoreboard for later comparison (6). When the keyboard controller detects a key pressed it triggers an Interrupt Service Routine (ISR) that reads the content of an internal register of the controller and store the coded value of the key into a memory location, asserting the status variable of the keyboard driver. After the assertion of the variable the SW test can proceed sending through VAL the received scan key code to an external scoreboard for comparison (7).
Figure 17 shows an example of C test implementing the described scenario.

```
int main(void) {
  // test initialization
  // (e.g.connects internal
  // pins with external pads)
  <ip>_test_init();
  prepare_interrupts();
  // Randomize a VIP configuration
  <ip>_vip_randomize();
  // Use the randomized data to
  // configure the IP
  populate_<ip>_desc_from_val();
  <ip>_config()
  // Starts the IP. Now it is
  // ready to receive data
  start_<ip>();
  while (1) {
    // wait for an interrupt...
    while(!<ip>_desc.int_asserted);
    // sends back the received data to
    // external scoreboard through VAL
    send_received_data_to_sb();
  }
}
```

Figure 17 Example of C code for unidirectional (Rx) IP tests

Another example of randomization with VAL is reported in Figure 18. In this case we have to verify the correct integration of a clock generator within a SoC

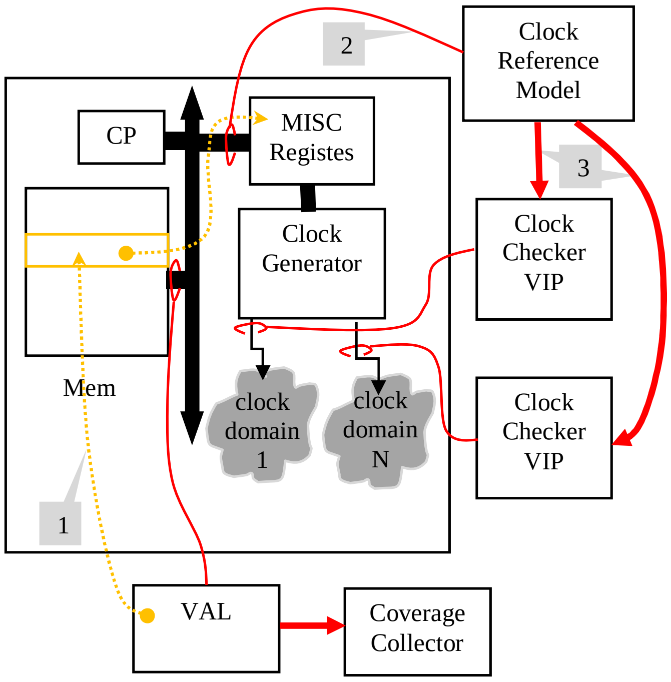

Figure 18 VAL usage model for clock generators

In this case the VAL is used to randomize all the possible combinations of clock enables and clock frequencies for any clock present in the SoC. The SW test triggers the randomization and the result of the operation is both sent outside through an analysis port to a coverage collector and it is serialized back into the memory of the SoC (1). This information will be used by the SW test to program the miscellaneous registers used to control the clock generator. The self-checking capability of the environment is composed by a reference model of the clock generator and a set of clock checker VIPs (one for each clock source). The reference model reads the same information provided by the miscellaneous register (2) and computes the expected frequencies for all the clocks of the system. Such information is forwarded to configure the clock checkers, each one dedicated to monitor a single line of clock (3).

VIII. Conclusions and Future works

In this paper we have described a methodology to bring Coverage Driven Constrained Random capabilities in SW-driven SoC tests. The key element of this approach is a verification component we call VAL. Around this component we developed a complete flow that allows to access the external verification environment directly from SW running on embedded CPU. Through VAL the embedded SW can trigger the randomization of any meaningful data for tests (e.g. ,VIP configurations, IP configurations, payload data, etc.) allowing the designers/verification engineers to exploit the benefits of randomization techniques to simplify the development of integration tests as well as to improve their overall quality.

In the SoCs we developed so far VAL demonstrated its versatility on enabling the possibility to combine together the integration tests of the different IPs into single stress tests dedicated to qualify the robustness of the system, or create complex scenario where several IPs interact together.

Currently the VAL and its randomization capabilities have been experimented mostly in HDL simulations. We are currently working on the creation of synthesizable versions of the VAL-FE to map its concepts on Emulators and bring Constrained Random technique in SW-driven Soc tests within Co-emulation environments.

Acknowledgment

We would like to thanks and acknowledge Matteo Barbati and Lanfranco Salinari for their contributions to the technical implementation of the VAL.

References

[1] TrekSoc product page, http://www.brekersystems.com/products/treksoc
[2] Infact product page, http://www.mentor.com/products/fv/infact
[3] Universal Verification Methodology, http://www.uvmworld.org/