2015 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

Web Template Mechanisms in SOC Verification

Alberto Allara, STMicroelectronics, Digital & Mixed Processes Asic Division,
Via Tolomeo 1, Cornaredo (Milano), Italy (*alberto.allara@st.com*)
Rinaldo Franco, STMicrolelectronics, Digital & Mixed Processes Asic Division,
Via Remo de Feo 1, Arzano (Napoli), Italy (*rinaldo.franco@st.com*)

***Abstract*—The SW-driven verification technique is widely used in the context of the System-on-Chip (SoC) verification. In order to optimize such activities it is common practice to use small virtual platforms developed in SystemC representing the abstraction of an actual SoC. This divide-and-conquer approach allows the development of complex SoC scenarios and verification components in a simplified environment before porting them in actual SoCs. The evident need is to keep the porting process as simple and straightforward as possible. To do that, we need to guarantee that the SW layers and the verification infrastructures are built in a way that the differences between a simplified virtual platform and an actual SoC are completely hidden to the test developers. In order to address the mentioned problem, in this paper we propose a new technique borrowed from the Web development community. The approach consists in automatically generating the SW layer and the verification infrastructures for a SoC using a Template language for Python called "jinja2".**

Keywords—Web template; SoC; UVM; SW-driven verification; VAL

I. INTRODUCTION

The SW-driven verification technique is widely used in the context of System-on-Chip (SoC) verification. In order to optimize such activity it is common practice to use small virtual platforms developed in SystemC representing the abstraction of an actual SoC. Such Lightweight Virtual Platforms (or LVP) are typically composed of an embedded CPU like the FastModel of ARM, an internal memory, a DMA model and the transactor components toward external BUS protocols, like, for instance, the AMBA protocol of ARM.

An LVP can be instantiated along with a DUT (e.g., a single IP or a more complex SoC subsystem), a Verification Abstraction Layer component (VAL) [1], used to allow the SW running on the embedded CPU to interact with the external verification environment and any other necessary Verification IPs. The resulting system represents an ideal platform for any test developer involved in the creation of SoC integration tests for its characteristics of speed and easiness of debug (for instance, the platform allows to connect the onboard CPU to a SW debugger and to easily perform an HW/SW co-verification) [3]. Once the tests are ready at LVP level, the test developer needs to port what has been developed directly at top-level in the actual SoC.

This divide-and-conquer approach has the evident need to keep the porting process as simple and straightforward as possible. To do that, we need to guarantee that the SW layers and the verification infrastructures are built in a way that the differences between a simplified LVP and an actual SoC are completely hidden to the test developers.

One possibility that we explored consists in keeping the information and the data relevant to distinguish a platform from another (the "model") completely separated from a layer representing the SW implementation of the functionalities (either HW or of the verification infrastructure) accessible by the test developers (we can call it the "view").

The technique is widely used in the context of SW engineering to implement data-driven Web applications. In such SW applications the relevant data are typically stored into Relational Databases and the Web applications respond to requests coming from browsers by generating dynamic HTML pages populated with database data. The Web application is usually based on an architectural pattern known as MVC (Model-View-Controller) or, in the Web context, as MTV (Model-Template-View), where the data ("Model") are separated from the way they are presented to the user (the "View" through the "Template").

In this paper we propose a new technique borrowed by a mainstream approach in the Web development to generate the SW layer and the verification infrastructures for a SoC using a Template language for Python called "jinja2".

II. TEMPLATE MECHANISM

The Template languages are tools used to simplify the dynamic generation of Web pages. One of such languages is Jinja2 [2], a modern and designer-friendly template language for Python; A Jinja template is a text file and can generate any text-based files as output (generally it is used to generate HTML, JS or CSS files but it can generate also C, HDL or HVL). A template contains variable and/or expressions, which get replaced with values coming from a context dictionary in Python during rendering. The template includes also tags to control the logic of the output generation.

Figure 1 reports an example of a minimal template coming directly from the documentation page of the Jinja2 site. The template can generate an HTML page that includes a list of HTML anchors and a heading title followed by the content of a variable.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>My Webpage</title>
</head>
<body>
    <ul id="navigation">
    {% for item in navigation %}
        <li><a href="{{ item.href }}">{{ item.caption }}</a></li>
    {% endfor %}
    </ul>

    <h1>My Webpage</h1>
    {{ a_variable }}

    {# a comment #}
</body>
</html>
```

Figure 1: A simple example of Jinja template

Analyzing the content of the template above, a reader can identify the {% for %}{% endfor %} tag statement. It is used to unroll its content based on the information of the "navigation" variable which in the example is a list of elements containing a href (the URL of a web resource) and a caption. Other supported tag statements in Jinja are {% if %}, {% macro %}, {% filter %}, {% set %}, {% include %}, {% import %}, and few others. The details of their behavior are reported directly in the Jija2 documentation page. The example shows also three cases of variable content printing: "item.href" and "item.caption" inside the {% for %} tag, and "a_variable". The variable content printing is enabled by enclosing the variable within a double braces delimiter ({{ … }}), while accessing the variable inside tags does not require any braces around it.

The Python APIs used to render a template are based on a central object called the template **Environment**. Instances of this class are used to store global data and load templates from the file system. The result is a **Template** object that provides a method to render the template files based on data expressed as Python dictionary.

III. TEMPLATE IN A SOC CONTEXT

Our proposal is to apply the Jinja2 template mechanism in the context of a SoC verification to generate a SW view and a HVL view in a consistent manner based on high level descriptions of a platform expressed in a JSON format (JavaScript Object Notation) [4].

JSON is a language independent open format using human-readable text. It is composed by data objects consisting of attribute-value pairs. The choice of using JSON format w.r.t. other formats more common in the

SOC context like, for instance, the IPXACT based on XML is justified by the fact that JSON it is extremely more compact than XML, aspect that simplifies the insertion and the manipulation of data in comparison with an equivalent model represented in XML. In addition, Python comes with a standard Library that allow to easily parse a JSON file and to convert it into a Python dictionary, a data structure directly used by JinJa to render a template. Finally, the most common metadata currently available in IPXACT files are associated to register map and pin-level connectivity, two aspects very low level w.r.t. the information we plan to insert in the JSON file for the platform description.

The overall proposed flow is reported in Figure 2.

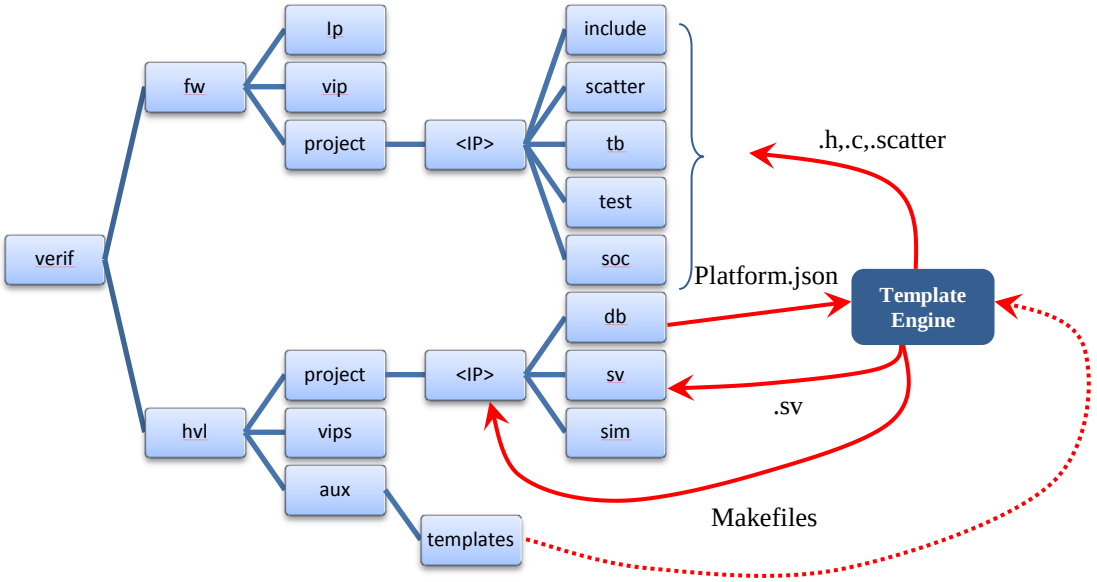

Figure 2: Web template flow applied to a SoC directory infrastructure

The figure shows a directory infrastructure for a SoC separated between firmware (identified by the "fw" subtree) executed by the embedded CPU of the SoC and the testbench environment (identified by the "hvl" subtree) needed for the verification.

The firmware subtree includes:

- ip: containing a set of directories representing the IPs present in the SoC. Each IP includes information regarding the register map description of the component, the low-level drivers in C and the platform independent integration tests in C.
- vip: containing a set of subdirectories representing the VAL backend components accessible from SW through a set of API functions. For each VAL backend there is a description in C of a "virtual" register map of the component and its associated low-level driver.
- project: containing a set of "platforms" that in Figure 2 are identified with "<IP>". Each platform can be an IP or a subsystem implemented with an LVP or assembled as a full SoC. A single platform contains header files for memory map of IPs and VIPs present in the platform, header file for interrupt lines and DMA request lines, mapping mechanism of Hardware Abstraction Layer (HAL) services to the actual APIs of the platform, ARM scatter files related to the memory layout of the platform etc.

On the other side, the "hvl" subtree includes:

- aux: containing all the stuff needed for the verification infrastructure like Makefiles, scripts and so on. It includes also the "templates" directory devoted to store all the Jinja templates.

- vips: containing the UVM systemverilog code that implements the different VAL backends. Each component matches an equivalent SW view in the "fw/vip" subtree.
- Project: containing a set of "platform" elements identified in Figure 2 with "<IP>". Each platform is matched with the SW view in the "fw/project/<IP>" subtree and contains the directory "db" dedicated to store the **platform.json** file with the high level details on how the platform is organized.

Every time there is a change in the platform.json file the Template engine script is called to regenerate from the JSON and from the template files the set of C code (.h and .c), scatter file, systemverilog TB code, and Makefiles illustrated in Figure 2.

IV. TEMPLATE ENGINE

The Template engine is a Python script that receives as input the platform.json file (*-f <json file>* switch) and the jinja template (*-i <template-file>* switch) and generates a rendered output (-o <generated-file> switch). Figure 3 reports the details of the code. In particular at lines 25-26 the json file is read and converted into a Python dictionary. Then, the *gen_template()* function is called. The function reads a template (line 11) and creates a template object through a Jinja2 Environment (line 12). Finally, the template is rendered based on the content of the Python dictionary and the output is written into an output file (lines 13-15).

```
1)  #!/usr/bin/env python

2)  import os
3)  from jinja2 import Environment, FileSystemLoader
4)  import json
5)  import argparse

6)  PATH = os.path.dirname(os.path.abspath(__file__))
7)  env = Environment(loader=FileSystemLoader(os.path.join(PATH, '.')),
              i.  trim_blocks=True)
8)  env.lstrip_blocks = True
9)  env.trim_blocks = True

10) def generate_template(data,templ_tb,gen_tb):
11)   f=open(gen_tb,'w')
12)   template=env.get_template(templ_tb)
13)   sv=template.render(data)
14)   f.write(sv)
15)   f.close()


16) def main():
17)   parser = argparse.ArgumentParser(description='Template generator')
18)   parser.add_argument('--cfg',"-f", action="store", dest="cfg",
19)   help="specify the configuration file in JSON",default="platform.json")
20)   parser.add_argument('--otb',"-o", action="store", dest="otb",
21)   help="define the file name of the generated file")
22)   parser.add_argument('--itb',"-i", action="store", dest="itb",
23)   help="define the file name of the template file")
24)   args = parser.parse_args()
25)   cfg_h = open(os.path.join(PATH,".",args.cfg),"r")
26)   data = json.load(cfg_h)
27)   generate_template(data,args.itb,args.otb)

28) if __name__ == "__main__":
29)   main()
```

Figure 3: Python code of the Template engine

V. THE PLATFORM DESCRIPTION FORMAT

The platform description file in the current implementation is characterized by the following information:

- IP section: at LVP level the platform can include one IP, while at SoC level it is usually composed of several IPs. Each IP includes information about the memory base address, the number of instances, the IP memory offset, the number of the interrupt lines connected with the IP (if any), and information regarding the DMA request identifier (if present).

- Memory map section: it contains information about the static RAMs present in the system with details on the memory base address, size of the memories, number of cut and size of a single cut. It includes also a set of regions present in the system, each one identified by a name, a base address, and a size.
- TB section: it defines the content of the verification environment. It is organized with the following information:
    - Clocks: at LVP levels the clock are modelled by UVM verification components controlled by a VAL while in an actual SoC they are mapped on real resources of the system. It is possible to specify the number and the characteristics of clocks; in particular, for each of them it is possible to specify if it is a direct clock or a derived one as well as its frequency rate (or its division factor in case of derived clocks). Such information can be used or not, depending on the characteristics of the platform (e.g., LVP vs. SoC).
    - Resets: in a similar way to clocks, the resets at LVP level are modelled by UVM components, while in a real SoC they are directly implemented in the system. For resets it is possible to define the number of reset lines and their level during initialization.
    - VAL front-end (VAL FE): it is possible to specify the number of VAL front-ends in the platform. Usually, at LVP level we have only one VAL FE, while at SoC level it is possible to have several of them. This happen for instance in case of SoCs with a number of power islands. In Power aware simulation it can happen that the internal memory where the VAL is connected goes in power down and having the possibility to access another VAL located in a region not in power down it is very useful. For each VAL FE it is reported the base address, the trigger base address and the trigger size (i.e. the three information needed to configure a VAL FE). In addition, a list of basic VAL backend components is specified. The supported VAL backend are the VAL timer, the VAL clock generator, and the VAL reset generator. The latters are needed at LVP level if the platform specifies clocks and resets. Finally, the VAL FE includes a list of UVM env. Each environment is associated with an IP or subsystem and it is connected directly to the VAL FE through an analysis port. The environment contains all the verification components (VAL backend, scoreboard, coverage collector, UVC, etc.) needed for the verification of the IP/subsystem. At LVP level typically only one environment is defined with the associated VAL backend, while at SoC the number of UVM environments is related to the number of verified IPS/subsystems

An example of platform expressed in JSON format is reported in Figure 4.

```
{
  "platform": {
    "name": "uart_at_lvp",
    "ip": [
     { "name": "uart",
       "n_instance": 2,
       "base_addr": "0x53000000",
       "instance_offset": "0x00001000",
       "int_line_n": [[117, 118],
                      [135, 138]],
       "dma_connection_tx": [["DMAC_DRV_DST_REQ_55"],
                             ["DMAC_DRV_DST_REQ_92"]],
       "dma_connection_rx": [["DMAC_DRV_SRC_REQ_55"],
                             ["DMAC_DRV_SRC_REQ_92"]]
     }

    ],
    "memory_map": {
      "esram": [
         {"name":"ESRAM_0", "base_addr": "F8280000", "n_cut":4, "cut_size": "8000"},
         {"name":"ESRAM_1", "base_addr": "F82A0000", "n_cut":4, "cut_size": "8000"}
       ],
      "regions": [
         {"name": "IPC_MEMORY_AREA", "base_addr": "ESRAM_0_CUT3_BASE_ADDR",
          "size": "0x1000"}
       ]
```

```
        }
       "tb": {
        "clock": [
             {"id": "0", "fref": "10000000", "duty": "50",
              "clockphase": "0", "commonstart": "1"},
             {"id": "1", "fref": "50000000", "duty": "50",
              "clockphase": "0", "commonstart": "1"},
             {"id": "2", "id_ref": "0", "div": "5", "clockphase": "0"},
             {"id": "3", "id_ref": "0", "div": "10", "clockphase": "0"},
             {"id": "4", "id_ref": "1", "div": "7", "clockphase": "0"}
        ],
        "reset": [
             {"id": "0", "active": "1", "commonstart": "1"},
             {"id": "1", "active": "1", "commonstart": "1", "length": "400"},
             {"id": "2", "active": "0", "commonstart": "0"}
        ]
        "val_fe": [
          {
            "base_addr": "B2000000",
            "trig_base_addr":"D8000000",
            "trig_size":"0010",
            "val_base": [
             { "name": "val_timer", "size" : "100" },
             { "name": "val_clkgm", "size" : "100" },
             { "name": "val_reset", "size" : "100" }
            ],
            "env": [
              "name": "uart"
              "val_be":[
                {"name": "val_ip", "instance_name": "val_ip","n_instance": 1, "size" : "100" }
              ]
            ]
          }
        ]
      }
     }
    }
```

Figure 4: example of a platform at LVP level

VI. EXAMPLE OF SOC TEMPLATES

In this section we will present two examples of Templates snippet and their resulting rendered outputs obtained using in both cases the platform description reported in Figure 4.

Figure 5 reports as first example the Template file to generate the memory map header file for IPs, RAMs and other relevant regions present in the platform. The rendering is called memory_map.h file and it is reported in Figure 6.

```
   #ifndef _MEMORY_MAP_H_
   #define _MEMORY_MAP_H_

   /* ATTENTION this file is automatic generated! DO NOT MODIFY BY HAND!   */
   /*  ESRAM MEMORY MAP  */

   {% for esram in platform.memory_map.esram %}
   #define {{ esram.name }}_BASE_ADDR             0x{{ esram.base_addr }}
   #define {{ esram.name }}_SOC_COMMON_BASE_ADDR  0x{{ esram.base_addr }}
   #define {{ esram.name }}_CUT_SIZE              0x{{ esram.cut_size  }}
   {% for cut in range(esram.n_cut) %}
   {% if loop.first %}
   #define {{ esram.name }}_CUT{{ cut }}_BASE_ADDR  {{ esram.name }}_BASE_ADDR
   {% else %}
   #define {{ esram.name }}_CUT{{ cut }}_BASE_ADDR  ({{ esram.name }}_CUT{{ cut-1 }}_BASE_ADDR
+ {{ esram.name }}_CUT_SIZE)
   {% endif %}
   {% endfor %}
   {% endfor %}

   ...
   /*  IPs MEMORY MAP  */
```

```
   {% for ip in platform.ip %}
   {% if (ip.n_instance > 1) %}
   {% for n_inst in range(ip.n_instance) %}
   {% if loop.first %}
   #define   {{ ip.name.upper() }}_{{ n_inst }}_BASE_ADDR 0x{{ ip.base_addr }}
   {% else %}
   #define {{ ip.name.upper() }}_{{ n_inst }}_BASE_ADDR  ({{ ip.name.upper() }}_{{ n_inst-1
}}_BASE_ADDR + 0x{{ ip.instance_offset }})
   {% endif%}
   {% endfor %}
   {% else %}
   #define   {{ ip.name.upper() }}_BASE_ADDR 0x{{ ip.base_addr }}
   {% endif%}
   {% endfor %}

   #endif // _MEMORY_MAP_H_
```

Figure 5: IP memory map Template

```
   #ifndef _MEMORY_MAP_H_
   #define _MEMORY_MAP_H_

   /* ATTENTION this file is automatic generated! DO NOT MODIFY BY HAND!   */
   /*  ESRAM MEMORY MAP  */

   #define ESRAM_0_BASE_ADDR             0xF8280000
   #define ESRAM_0_SOC_COMMON_BASE_ADDR  0xF8280000
   #define ESRAM_0_CUT_SIZE              0x8000
   #define ESRAM_0_CUT0_BASE_ADDR  ESRAM_0_BASE_ADDR
   #define ESRAM_0_CUT1_BASE_ADDR  (ESRAM_0_CUT0_BASE_ADDR + ESRAM_0_CUT_SIZE)
   #define ESRAM_0_CUT2_BASE_ADDR  (ESRAM_0_CUT1_BASE_ADDR + ESRAM_0_CUT_SIZE)
   #define ESRAM_0_CUT3_BASE_ADDR  (ESRAM_0_CUT2_BASE_ADDR + ESRAM_0_CUT_SIZE)
   #define ESRAM_1_BASE_ADDR             0xF82A0000
   #define ESRAM_1_SOC_COMMON_BASE_ADDR  0xF82A0000
   #define ESRAM_1_CUT_SIZE              0x8000
   #define ESRAM_1_CUT0_BASE_ADDR  ESRAM_1_BASE_ADDR
   #define ESRAM_1_CUT1_BASE_ADDR  (ESRAM_1_CUT0_BASE_ADDR + ESRAM_1_CUT_SIZE)
   #define ESRAM_1_CUT2_BASE_ADDR  (ESRAM_1_CUT1_BASE_ADDR + ESRAM_1_CUT_SIZE)
   #define ESRAM_1_CUT3_BASE_ADDR  (ESRAM_1_CUT2_BASE_ADDR + ESRAM_1_CUT_SIZE)

   /*  REGIONS MEMORY MAP  */

   ...

   /*  IPs MEMORY MAP  */

   #define UART_0_BASE_ADDR 0x53000000
   #define UART_1_BASE_ADDR (UART_0_BASE_ADDR + 0x00001000)

   #endif // _MEMORY_MAP_H_
```

Figure 6: memory_map.h file

The second set of files is related to the verification environment and represents the top UVM environment (called top_sve.sv). Figure 7 shows the Template used for the platforms at LVP level. The reader can notice that only one VAL FE is enforced. Figure 8 reports its rendered output.

```
   `ifndef _TOP_SVE
   `define _TOP_SVE

   ...

   `include "{{ tb.env.name }}_env.sv"

   class top_sve extends ip_env;

     val_env val;
     clkgm_env  clock_env;
     {{ tb.env.name }}_env  {{tb.env.name }}env;

     ...

     // build
     virtual function void build_phase(uvm_phase phase);
```

```
       string inst_name;
       super.build_phase(phase);

       if (inst_val) begin
         val = val_env::type_id::create("val",this);
       end

       ...

       set_config_int("{{ tb.env.name }}env", "trigger_addr", `{{ tb.env.name.upper()
}}_TRIG_ADDRESS-`VAL_TRIG_BASE);
       set_config_int("{{ tb.env.name }}env", "mem_addr",     `{{ tb.env.name.upper()
}}_MEM_ADDRESS);
       {{ tb.env.name }}env = {{ tb.env.name }}_env::type_id::create("{{ tb.env.name
}}env",this);
     endfunction : build_phase

     function void connect_phase(uvm_phase phase);
       super.connect_phase(phase);

       ...
   {% for vip in tb.env.val_be %}
   {% if loop.first %}
       val.ahb_fe.ahb_conv.send_item.connect({{ tb.env.name }}env.{{ tb.env.name
}}_env_export);
   {% endif%}
   {% endfor %}

     endfunction: connect_phase

   endclass : top_sve
   `endif
```

Figure 7: UVM top sve Template for LVP

```
   `ifndef _TOP_SVE
   `define _TOP_SVE

   ...

   `include "uart_env.sv"

   class top_sve extends ip_env;


     val_env val;
     clkgm_env  clock_env;
     uart_env  uartenv;

     ...
     // build
     virtual function void build_phase(uvm_phase phase);
       string inst_name;
       super.build_phase(phase);

      if (inst_val) begin
         val = val_env::type_id::create("val",this);
       end

       ...

       set_config_int("uartenv", "trigger_addr", `UART_TRIG_ADDRESS-`VAL_TRIG_BASE);
       set_config_int("uartenv", "mem_addr",     `UART_MEM_ADDRESS);
       uartenv = uart_env::type_id::create("uartenv",this);

     endfunction : build_phase

     function void connect_phase(uvm_phase phase);
       super.connect_phase(phase);

       ...
       val.ahb_fe.ahb_conv.send_item.connect(uartenv.uart_env_export);
     endfunction: connect_phase

   endclass : top_sve
```

```
`endif
```

Figure 8: top_sve.sv file

VII. CONCLUSIONS AND FUTURE WORKS

In this paper we have presented a novel technique borrowed from the Web engineering to use a Template mechanism and a platform description in JSON for the automatic generation both of the firmware and the verification environment of a given SoC. The approach allows to easily move in a consistent way from a LPV environment to a full SoC and from one SoC to another keeping the differences among platforms as hidden as possible.

We believe we have just scratched the surface for the application of this technique in the context of the SoC verification. In particular we are starting to explore the possibility to describe in JSON the IO connectivity of a SoC and use the Template mechanism to generate automatically the related code. It will be possible to generate the low-level driver of the IO control for the firmware side and to generate the SystemVerilog code implementing the testbench environment. The testbench code will include a de-muxing mechanism to connect the internal pins of SoC IPs to the external verification components through SoC pads.

Another possible application of the Templates is for documenting the features of a platform such as the memory mapping layout, the list of supported interrupt and so on.

ACKNOWLEDGMENTS

We would like to extend our thanks to all the members of our SoC verification team in DMA division. In particular, we would like to thank Matteo Barbati, Matteo Ferranti, Andrea Masi and Massimo Vincenzi.

REFERENCES

[1] A. Allara, F. Brognara, “Bringing constrained random into SoC SW-driven verification”, DVCon SanJose 2013

[2] Jinja2 Home Page: http://jinja.pocoo.org/docs/dev/

[3] P. Kumar,D. P. Singh, A. Jain,”A methodology for vertical Reuse of functional verification from subsystem to SoC level with seamless soC emulation”, DVCon Europe 2014

[4] JSON format page: http://www.json.org/