Web Template Mechanisms in SOC Verification

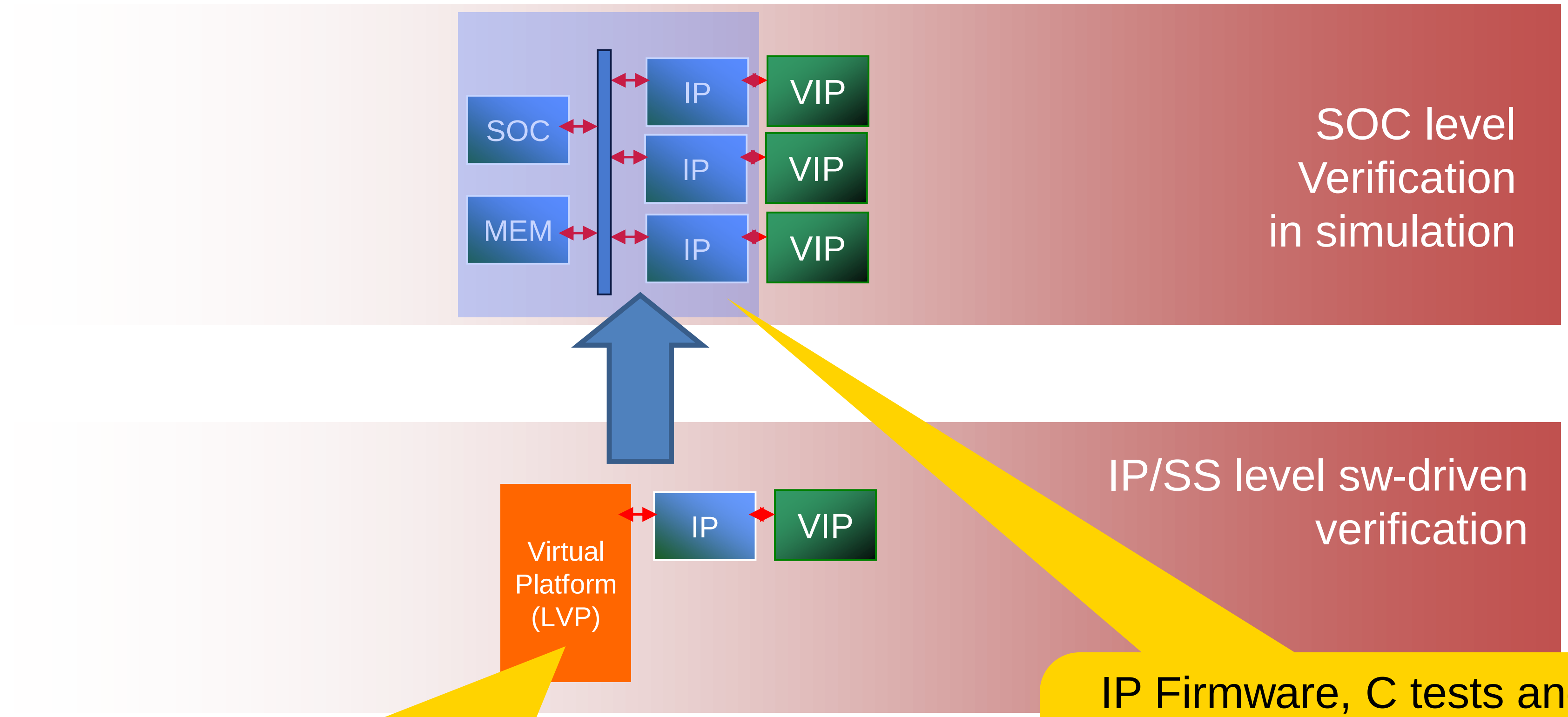
Rinaldo Franco, Alberto Allara

STMicroelectronics, Digital & Mixed Processes
Asic Division

Key SOC Methodologies

- IPs & SoC verification environments are based on UVM-methodology
  - Advanced verification capabilities
  - Robust class libraries
  - Open, Interoperable
  - CAD Multi-vendor compatibility
- Software Driven Verification for IPs & SoC
  - Development of SW tests running at bare metal without any OS
  - Low-level drivers to abstract hardware
    - Reusability during Top-Level verification
    - Reusability during silicon validation
  - Verification environment exposed on SW (VAL)
- Use of Virtual Platform for the verification
  - An LVP (Lightweight Virtual Platform) instantiated with Dut (IP or SUBS) used to develop test that will be ported at SoC level

The path to SOC verification


SOC
MEM
IP
IP
IP
VIP
VIP
VIP
SOC level Verification in simulation
Virtual Platform (LVP)
IP
VIP
IP/SS level sw-driven verification

LVP enables the development of integration tests in a simplified environment (abstraction of SoC)

IP Firmware, C tests and verification components are developed at LVP level and ported at SOC

Hide the differences

- Main assumption of the path from LVP to SOC:
  - The scenario developed at LVP must be reusable at SOC
- This implies that:
  - The differences in the SW layers and/or in the verification infrastructures are hidden to the test developer

Our Proposal

- Keep the information and relevant data to distinguish platforms (the “**model**”) separated from a layer representing SW and HVL implementation of functionalities (the “**view**”)
- In the Web application domain the technique is an architectural pattern known as **MTV** (Model-Template-View)
  - The data (“**Model**”) are separated from the way they are presented to the user (the “**View**” through “**Template**”)
- The Template language used is a Python package called “**Jinja2**”

What is a Template Language?

- The Template languages are tools used to simplify the dynamic generation of Web pages

- **Jinja2** is a modern and designer-friendly template language for Python
  - a Jinja2 template is a text file and can generate any text-based files as output
  - http://jinja.pocoo.org/docs/dev/
- A Jinja2 template contains variable and/or expressions, which get replaced with values coming from a context dictionary in Python during rendering

Example of Template mechanism

```
<!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>
```

Unrolls the content based on the information of “navigation” variable

Each item from navigation list include an href and a caption

The content of variable is represented with {{ }}

Supported tags are {% if %}, {%macro%}, {%filter%}, {% set %}, {%include%}, {% import %},..

Templates in the SOC context

- Our proposal is to apply the Jinja2 template mechanism in the context of a SoC verification

- The templates are used 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

Why JSON?

- JSON is a language independent open format using human-readable text.
- The choice of using JSON w.r.t. other formats more common in the SOC context (e.g. XML) is due to a list of benefits:
  - Python comes with a standard library to easily convert a JSON file into a dictionary
    - Jinja2 uses the dictionary to directly render a Template
  - JSON is extremely more compact than XML, aspect that simplifies the insertion and the manipulation of data
  - Typically IPXACT data targets register map and pin-level connectivity not addressed by the platform description

Template Engine

```
def generate_template(data,templ_tb,gen_tb):
    f=open(gen_tb,'w')
    template=env.get_template(templ_tb)
    sv=template.render(data)
    f.write(sv)
    f.close()
def main():

def converthex2dec(n,fmt=None):
        return int(n,16)

env.filters['converthex2dec']=converthex2dec

parser = argparse.ArgumentParser(description='Template generator')
parser.add_argument('--cfg',"-f", action="store", dest="cfg",
help="specify the configuration file in JSON",default="platform.json")
parser.add_argument('--otb',"-o", action="store", dest="otb",
help="define the file name of the generated file")
parser.add_argument('--itb',"-i", action="store", dest="itb",
help="define the file name of the template file")
parser.add_argument('--extval',"-e", action="store", dest="extval",
help="pass value to the template file", default="0")
args = parser.parse_args()
cfg_h = open(os.path.join(PATH,".",args.cfg),"r")
data = json.load(cfg_h)
generate_template(data,args.itb,args.otb)
```

2) Read the input template and create a template object

3) Render the template based on the content of the dictionary

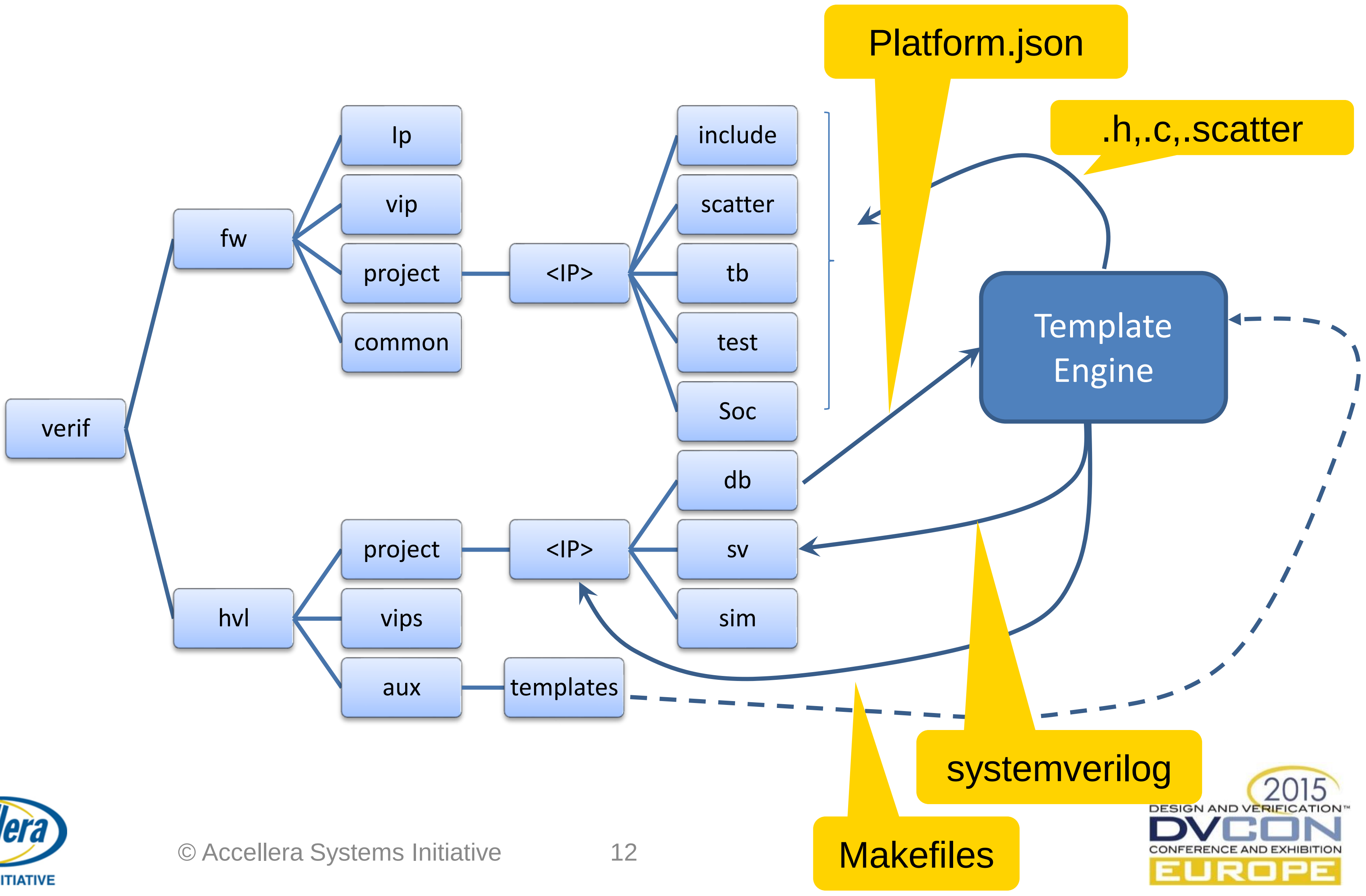
Example of user defined filter

1) Read and convert the JSON into a dictionary

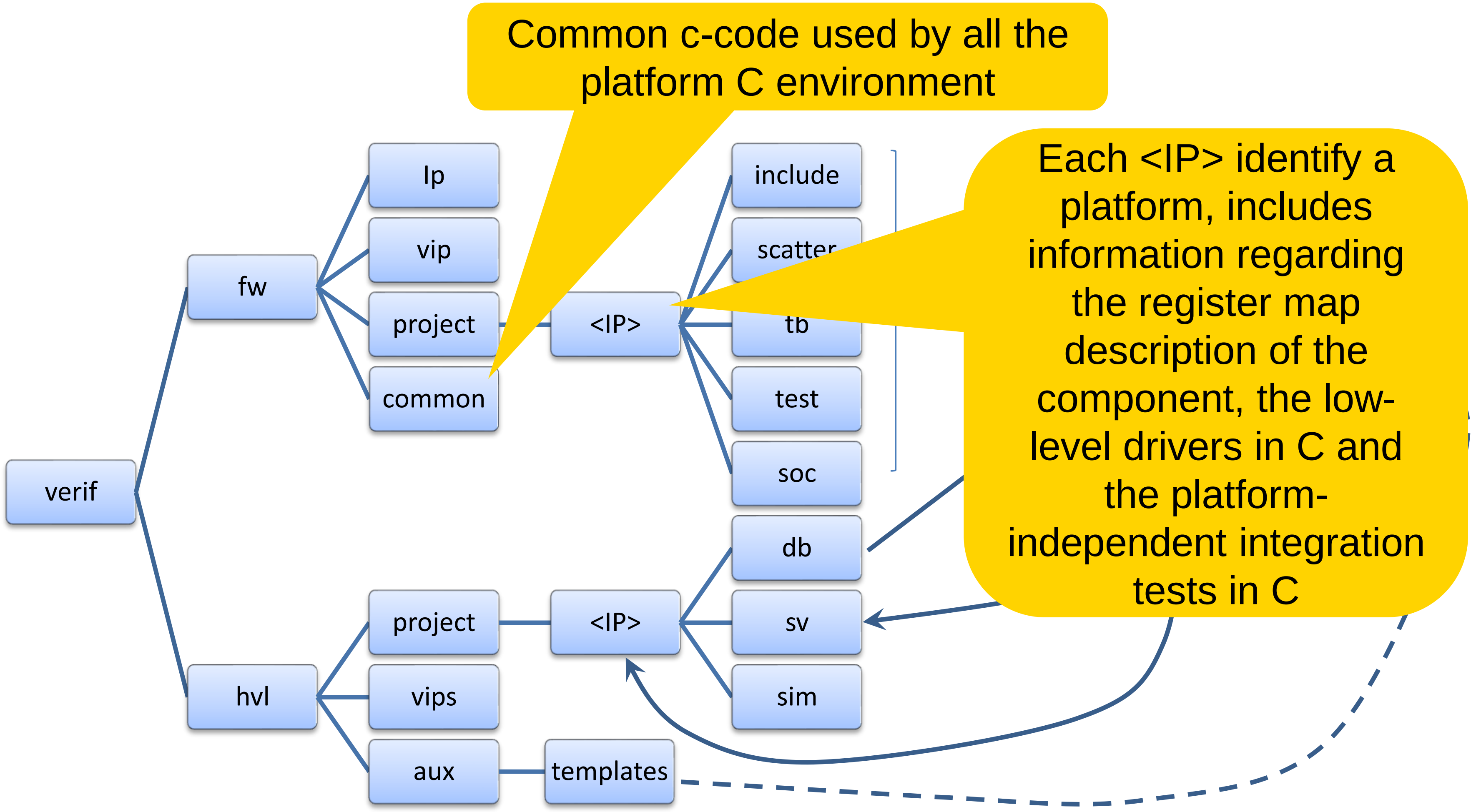
Example of Template file

```
#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_
```

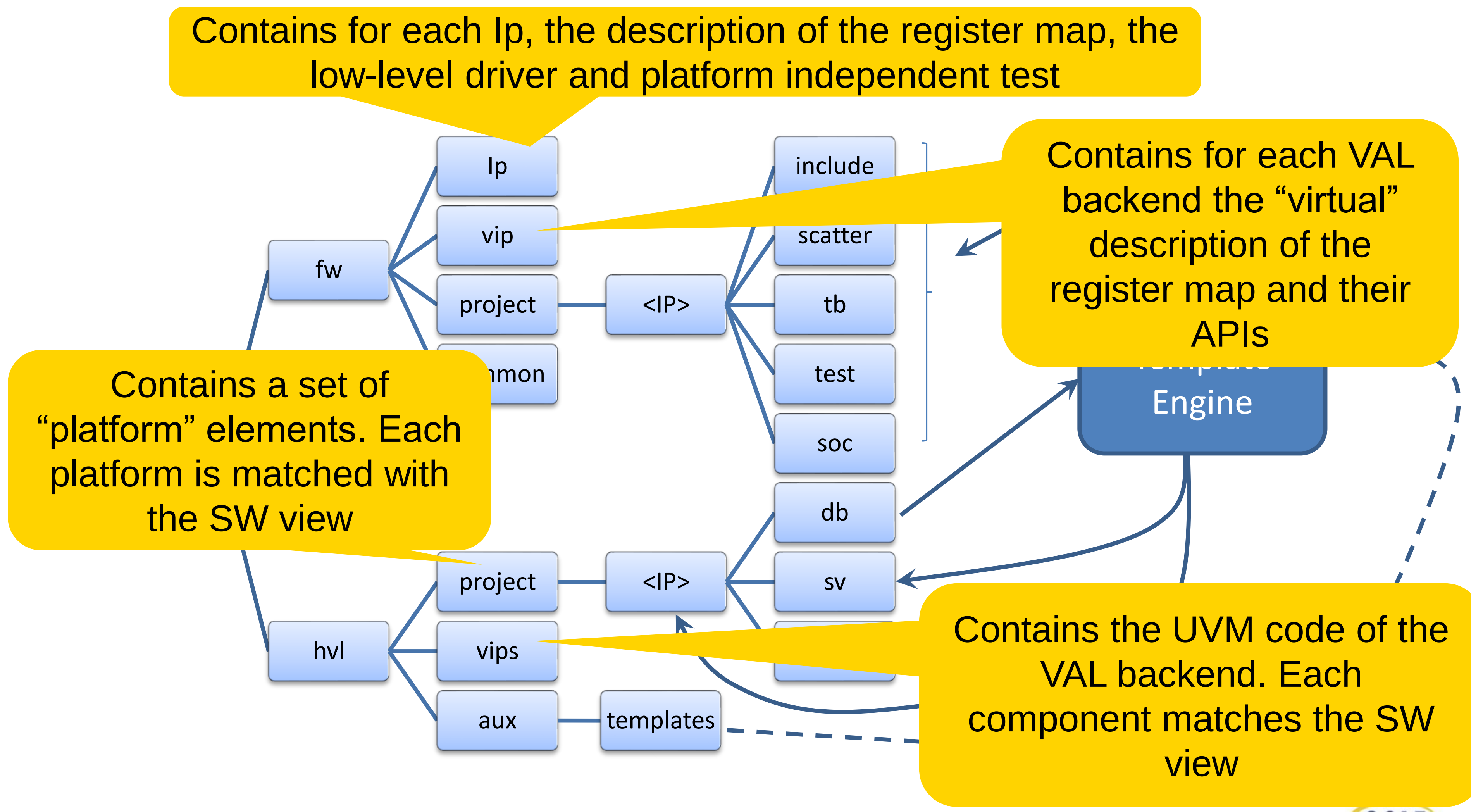
Template flow applied to a SOC



Template flow applied to a SOC (2)


Common c-code used by all the platform C environment
verif
fw
hvl
lp
vip
project
common
<IP>
include
scatter
tb
test
soc
db
sv
sim
project
vips
aux
<IP>
templates
Each <IP> identify a platform, includes information regarding the register map description of the component, the low-level drivers in C and the platform-independent integration tests in C

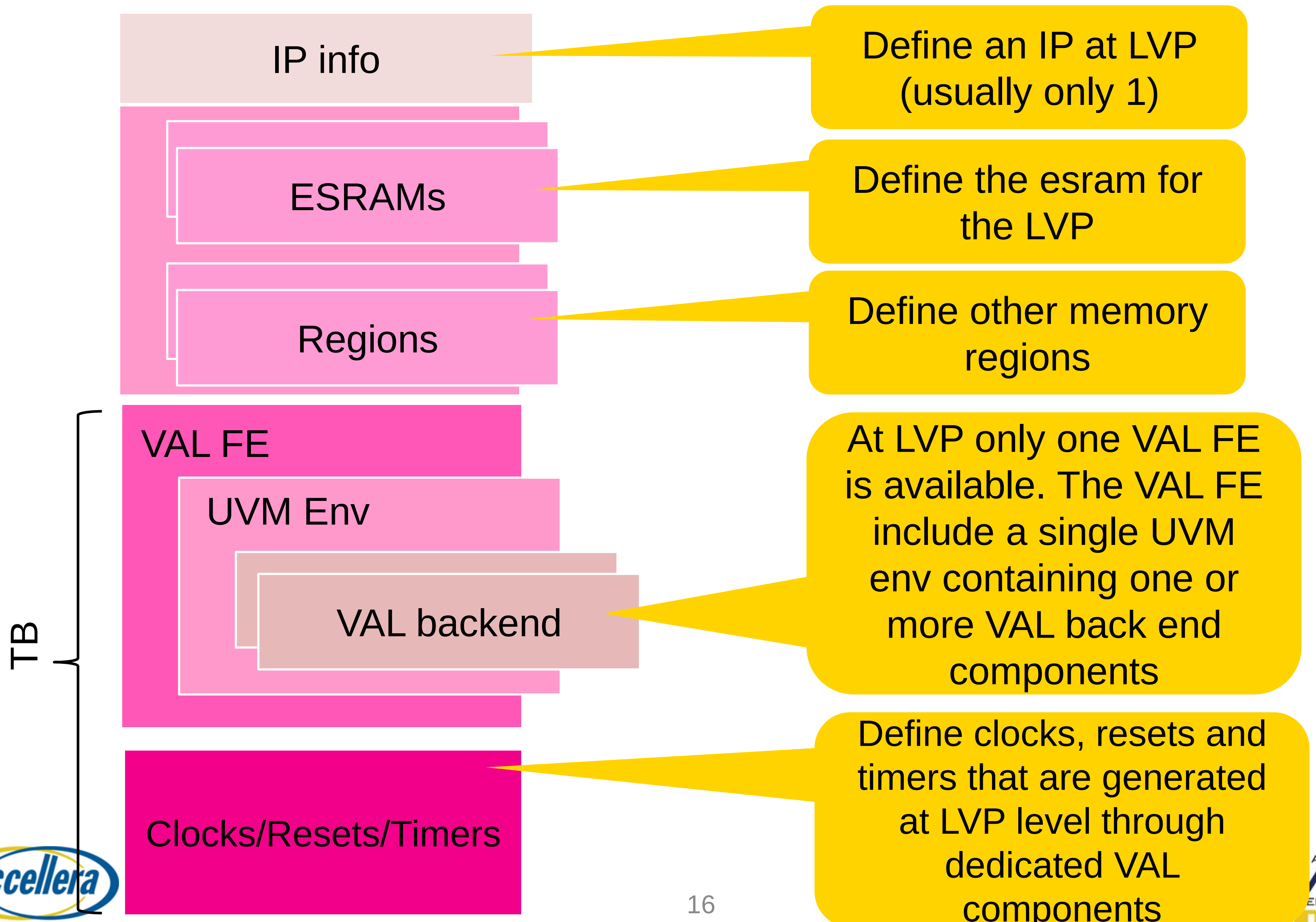
Template flow applied to a SOC (3)


Contains for each Ip, the description of the register map, the low-level driver and platform independent test
Contains for each VAL backend the “virtual” description of the register map and their APIs
Contains a set of “platform” elements. Each platform is matched with the SW view
Contains the UVM code of the VAL backend. Each component matches the SW view
fw
Ip
vip
project
mon
<IP>
include
scatter
tb
test
soc
db
sv
Engine
hvl
project
vips
aux
<IP>
templates

Platform Description File Format

- The platform description file is characterized by the following structure of information:
  - Details on IPs (e.g. base address, INT lines, DMA lines,..)
  - Static RAM and Memory regions
  - Testbench details with information regarding:
    - Clocks
    - Resets
    - Timers
    - VAL front-ends each them connecting a set of UVM env

Platform at LVP level



Define an IP at LVP (usually only 1)

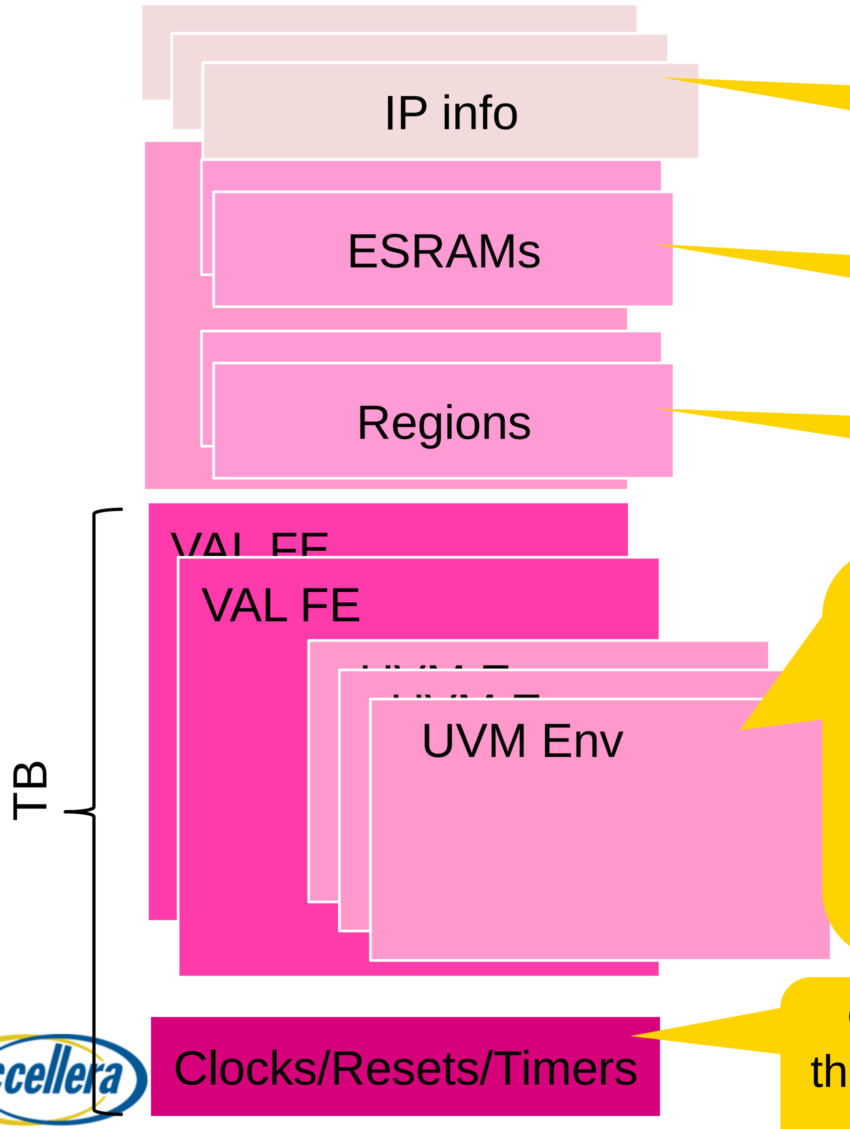
Define the esram for the LVP

Define other memory regions

At LVP only one VAL FE is available. The VAL FE include a single UVM env containing one or more VAL back end components

Define clocks, resets and timers that are generated at LVP level through dedicated VAL components

Platform at SOC level



At SOC level the platform is a set of IPs

Define the esram for the LVP

Define other memory regions

The platform contains a set of UVM env, one for each IP/SS. Each UVM env is directly connected to a VAL FE and internally it takes care of managing the VIPs needed for the verification of the associated IP. More than one VAL FE is possible

Clocks, resets and Timers are features of the SOC and not of the TB. Their information are used to pre-generate the related FW

Conclusions

- The Web Template Mechanism allow to separate the data (Platform configuration file) from the way they are used in the layers representing SW implementation of functionalities (c-code) and HDL verification infrastructures (System Verilog – UVM).

- The Platform configuration file contains high level descriptions of the scenario developed at LVP that can be reusable at SOC level, hiding differences to the test developer and reducing porting overhead.