2016 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

An open and flexible SystemC to VHDL workflow for rapid prototyping

Bastian Farkas, E.I.S., TU Braunschweig, Braunschweig, Germany (farkas@c3e.cs.tu-bs.de)
Syed Abbas Ali Shah, E.I.S., TU Braunschweig, Braunschweig, Germany (shah@c3e.cs.tu-bs.de)
Jan Wagner, E.I.S., TU Braunschweig, Braunschweig, Germany (wagner@c3e.cs.tu-bs.de)
Rolf Meyer, E.I.S., TU Braunschweig, Braunschweig, Germany (meyer@c3e.cs.tu-bs.de)
Rainer Buchty, E.I.S., TU Braunschweig, Braunschweig, Germany (buchty@c3e.cs.tu-bs.de)
Mladen Berekovic, E.I.S., TU Braunschweig, Braunschweig, Germany (berekovic@c3e.cs.tu-bs.de)

***Abstract*—We present a workflow for HDL code generation from virtual platform configurations. For this, we utilize the scripting language Python and a template engine called Jinja2, which is popular among web developers. As virtual platform we use SoCRocket, which is based on Cobham Gaislers GRLIB. Therefore we can access a large library of VHDL models from which we can stick together various system-on-chip configurations. These configurations can be quickly brought up on an FPGA board for prototyping purposes.**

Keywords— SystemC; Python; VHDL; rapid prototyping; templates; SoCRocket

I. INTRODUCTION

As the complexity of high-end SoCs increases steadily, so does the design process itself. Between 2003 and 2013 development complexity has increased threefold [1]. To counter this increase in complexity, more personnel is hired, the duration of projects increases, and more IP cores and designs are reused. It has been shown that increasing team sizes reduced the overall efficiency dramatically [1]. To restore design efficiency while dealing with increased design complexity, another approach is to focus on developing new design methodologies and tools.

Using high-level modeling languages like SystemC in early design phases has become mainstream in most application areas today. In the fast-paced consumer electronics field, it dramatically reduces time needed for production cycles. In other areas where testing and verification are more prevalent, the adoption has been hesitant.

With the introduction of UVM, also more tradionally conservative industries like automotive are adopting these new methodologies, whereas in the aerospace domain it is still not used much – although also here engineers suffer from the design gap. Especially in the aerospace domain, there are strict requirements for reliability and power efficiency. To fulfill these, extensive testing and validation is needed which could be greatly accelerated with the use of a high-level simulation platform like SoCRocket [11].

Together with ESA we have developed a state-of-the art SystemC framework [11] with the space domain in mind. For this, we have modeled Cobham Gaisler's GRLIB models that are available under GPL. Our own framework is available under AGPL. During initial development we have verified our SystemC models against Gaisler's VHDL models, so the focus of the work presented in this paper now lies on rapid prototyping and validation. Never the less, also verification could very well be done in an automated manner with our platform and scripting interface.

Prior to this work, we have extended the inital platform with an introspection and reflection interface to aid debugging [2] as well as a universal scripting interface [3]. This scripting interface enabled a new design workflow for quick and easy validation of configurations during design-space exploration and rapid prototyping. With all these features in place, we were able to redesign our design-space exploration methodology. With the help of a flexible template engine, we are able to generate VHDL code from our SystemC platform configurations. Tradionally, a

template engine like this would be used in web development or text processing, but it is flexible enough for code generation as well as we will show later.

The rest of the paper is organised as follows: In the subsequent section we will give an overview of similar approaches and show how our approach is unique. In Section III we explain our implementation in more detail. Then, we will discuss our findings in Section IV and finally conclude the paper in Section V.

II. Related Work

In recent years, a common way to go from an abstract SystemC model down towards the RTL level was through IP-XACT [4]. IP-XACT contains metadata information for models in a XML format that has been standardized. Although since it is XML it can be easily extended and repurposed. Cho et. al. explain in [5] their design methodology which uses IP-XACT metadata for platform integration and verification after the architecture exploration. Kamppi et. al. follow a similar path with their Kactus 2 project, which provides a GUI for putting together a SoC at the system level. Given that you already have your IP-XACT descriptions for your IP core library this could be a comfortable approach. Unfortunately all these modifications make their software and their IP-XACT libraries incompatible to other tools and the latest standard. But it seems that Pekkarinen et. al. have updated the tool to the latest standards [6] to achieve interoperability again.

Somehow the IP-XACT metadata needs to be written, and surely this is not done by hand (anymore). Tallec et. al. describe in [7] how it is possible to extract IP-XACT metadata from existing SystemC/TLM libraries with their SCIPX tool. It is an interesting approach, although it has its limitations and they tested it only on simple designs. Through our universal scripting interface, we have access to all required metadata information and could use a simple Python script to extract it.

The focus of the previously mentioned works was mainly in extracting and using metadata and not directly generating HDL code. A preprocessor from any current compiler suite could also be used for code generation in languages the suite was not originally designed for, but then it would need to be integrated in the tool workflow through a scripting interface or API. In [8] Fischer et. al. describe how they use a C-based template engine to be able to generate prototypes quickly from a large configuration database. With the help of their templating engine, they can generate a working platform from a collection of VHDL IP cores. They however do not mention whether they use IP-XACT metadata in their configuration database or of they have their own format. Another approach was presented by Ecker et. al in [14]. They developed a metamodeling and code generation framework based on Python and a template engine, which creates meta-models from specification to then generate the desired design view and accompanying design tools.

The closest to our own approach is probably [9] by Franco and Allara. They used the Jinja2 [10] template engine to create UVM verification testbenches, which is a popular choice in the web development community. They save their platform configurations in a JSON (JavaScript Object Notation) file and process it with their template engine written in Python.

III. Implementation

Before we dive into the details of our template engine description, we have to give a short overview of our SoCRocket Virtual Platform. It has been introduced in [11]. The current source code is available at [12] and the documentation at [13].

For this work, we used a small platform configuration as seen in Figure 1. The AHB/APB buses are used to connect several peripherals and a Leon3 CPU. All these components and their interconnections are described in the *sc_main.cpp* file. Our goal was to be able to quickly synthesize and run a configuration like that on our target platform (Avnet Zedboard).

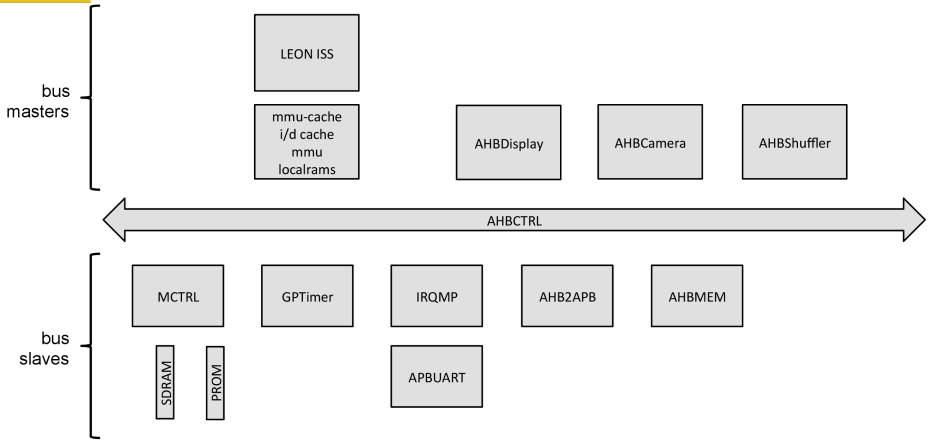

Figure 1 The SoCRocket platform configuration used in our lab

The components that comprise the platform shown in Figure 1 almost all come from the Cobham Gaisler GRLIB. The components AHBDisplay, AHBCamera and AHBShuffler have been developed by us and exist as SystemC as well as VHDL descriptions. The GRLIB contains several VHDL IP cores for the development of SoCs and is published as free software under the GNU General Public License (GPL). The centrepiece of the GRLIB is the LEON3 Processor IP core, a 32-bit processor using the SPARC V8 architecture. Nearly all other IP cores are connected to the LEON3 via AMBA 2.0 AHB/APB bus. In addition to the raw IP cores for processor, buses etc., GRLIB also contains template designs for a variety of evaluation boards as well as general templates. With these templates it is possible to quickly implement a LEON3 system on a Field Programmable Gate Array (FPGA), provided all the necessary tools are installed. Every design contains a top-level VHDL file *leon3mp.vhd*, a configuration file *config.vhd* that contains the GENERICS configuration and is automatically generated by a graphical configuration utility "xconfig", a sample testbench *testbench.vhd* which simply executes the code given in *systest.c* containing a small test program suitable for the design, lastly the *Makefile* allows compilation, synthesis, and even programming of the FPGA for which the specific design was created.

In our SystemC platform the equivalent to GRLIB's *config.vhd* is a JSON configuration file which can be optionally supplied to a simulation.

JSON stands for "JavaScript Object Notification". Contrary to its name it is a language-independent data format. Within SoCRocket, JSON can be used as data format to extract a SoCRocket configuration as well as to reconfigure some SoCRocket IP core parameters without recompiling the C++ and SystemC files.

As template engine, we opted for Jinja2 [10] since it is very flexible and works well with Python. Jinja2 is mainly used in web development scenarios to generate HTML/CSS code, but it can be very well used to create any other text-based language like VHDL. The Jinja2 template engine works in a similar way as many other template engines: the designer creates a template file with placeholders and then let a script fills these placeholders with provided data and "renders" the template.

The first thing to do when adopting a template design from the GRLIB for the use with SoCRocket is to recognize the similarities and differences between the designs. It has to be noted which units are used in the VHDL design. The default SoCRocket SystemC design is using the following modules: SDRAM model (sdram), Memory Controller (mctrl), General Purpose Timer Unit (gptimer), Interrupt Controller (irqmp), AHB Controller (ahbctrl), AHB Memory (ahbmem), APB Controller (apbctrl), APB UART (apbuart1), MMU Cache, Leon3 Processor.

The *leon3mp.vhd* in connection with the *config.vhd* of the *leon3-digilent-xc7z020* design states, which units are used in the ZedBoard design: one LEON3 CPU, Debug Support Unit, AHB Controller, AHB JTAG, APB Controller, Interrrupt Controller, General Purpose Timer Unit, General Purpose IO Ports, APB UART, AHB Status Register, and AHB ROM.

The ZedBoard design reads the initial setup from the AHB ROM, otherwise the onboard DDR RAM is used as memory. Therefore, a memory controller unit is not present like in other designs and it is neither necessary nor possible to carry over the settings for the `mctrl` unit from the SystemC design. The settings for the LEON3 CPU, the APB Controller, AHB Controller, GPTimer, APB UART, Interrupt Controller and APB UART shall be carried over from the SystemC design. Other units should be disabled, if the implementation allows for it. The AHB ROM module is needed, for example, as it loads the initial setup. Settings from the AHB MEM module can also be used for the AHB ROM VHDL module, as testing has shown.

We could not directly work with the existing Zedboard example design supplied with the GRLIB. Hence we modified it in a way, that just the units are active, which are also present in the SoCRocket design. We had to remove/disable the following components: Debug Support Unit, AHB JTAG, GPIO, AHB Status Register. We tested our modified design with Questasim and it worked. After that we synthesized it with the Xilinx tools and tested it on the Zedboard. This also worked. Now we had a base for our template.

We created templates for all the design files listed in the previous section. Our Python script contains two important function **set_var** and **create_generic_map**. They will be explained momentarily. The **set_var** function, depicted in Listing 1, matches SystemC variables to the ones inside *config.vhd* and works like this:

1. The given variable is searched in the SystemC model.
2. If it is a hexadecimal or boolean value, it is converted into a VHDL conform notation of the particular numeral system. Otherwise the value is left untouched.
3. Finally, the values is returned to the function and ultimately written into the created *config.vhd*.

The **create_generic_map** function is more complicated: Listing 3 shows the original generic map code of the *ahbctrl.vhd*. In Listing 4 the modified part is seen showing the use of the **create_generic_map** function, which itself is depicted in Listing 2.

```
1 def set_var(variable):
2     if 'base' in variable:
3         if variable['base'] == 'hex':
4             return '16#'+'%03X'%variable['value']+'#'
5     elif isinstance(variable['value'], bool):
6         return str(int(variable['value']))
7     else:
8         return variable['value']
```

Listing 1 The **set_var** function

```
def create_generic_map(values):
    var_string = ''
    if isinstance(values, dict):
        key = values.keys()
        for i in key:
            if not isinstance(values[i], list):
                if 'vhdl_name' in  values[i]:
                    if isinstance(values[i]['value'], bool):
                        var_string += values[i]['vhdl_name']+'=>'\
                                +str(int(values[i]['value']))+',\n'
                    elif 'base' in values[i]:
                        if values[i]['base'] == 'hex':
                            var_string += values[i]['vhdl_name']+'=>'+\
                                    '16#'+'%03X'%values[i]['value']+\
                                    '#'+',\n'
                    else:
                        var_string += values[i]['vhdl_name']+'=>'+\
                                str(values[i]['value'])+',\n'

                #parameters from tpp files
                elif i == 'hindex' or i == 'pindex' or i == 'pirq' \
                        or i == 'hirq':
                    if not values[i]['value'] == 0 or not (i=='pirq' \
                            or i=='hirq'):
                        var_string +=   i +'=>'+\
                                format(values[i]['value'])+',\n'
                elif i == 'hmask' or i == 'pmask' or i == 'paddr'\
                        or i == 'haddr':
                    var_string += i +'=>'+'16#'+\
                            '%03X'%values[i]['value']+'#'+',\n'
        #delete the comma
        var_string = var_string[:-2]
    return var_string
```

Listing 2 The **create_generic_map** function

In this particular example, the function **create_generic_map** is creating a generic map in VHDL based on the generic settings of the AHBctrl unit. The generic parameters of the AHBctrl unit are searched. Every generic parameter that has the **vhdl_name** property is added to the genericmap, as well as all **hindex**, **pindex**, **pirq**, **hirq**, **hmask**, **pmask**, **paddr,** and **haddr** values. The **vhdl_name** property needs to be added to be attached to the SystemC models. This is required because the SystemC names of some models differ inside the simulation (e.g. uart0) from the component name inside GRLIB. This property can be added in several ways: inside **sc_main** as **sr_param**, in the constructor of the module, inside a JSON platform configuration, or even inside a Python script at the start of simulation. The easiest option to ensure consistency between VHDL and SystemC would be a JSON platform configuration which is maintained together with the models.

From our experiences with this particular example design, we formulated a list of steps that need to be taken to adapt other designs:

1. Comparison of modules used in the GRLIB VHDL template to the modules used in SoCRocket

```
1 ahb0 : ahbctrl -- AHB arbiter / multiplexer
2 generic map (defmast => CFG_DEFMST, split => CFG_SPLIT,
3   rrobin => CFG_RROBIN, ioaddr => CFG_AHBIO,
4   fpnpen => CFG_FPNPEN, nahbm => maxahbm , nahbs => maxahbs )
5 port map (rstn, clkm, ahbmi, ahbmo, ahbsi, ahbso);
```

Listing 4 Original AHB control unit in *leon3mp.vhd*

```
1 ahb0 : ahbctrl -- AHB arbiter / multiplexer
2 generic map (
3   {{ ahbctrl.generics|create_generic_map }} )
4 port map (rstn, clkm, ahbmi, ahbmo, ahbsi, ahbso);
```

Listing 3 Modified AHB control unit in *templ_leon3mp.vhd*

2. Deactivation of all modules in the VHDL model, that are not used in SoCRocket and verification of the function of this model. If there are any errors, specific modules might need to be reactivated.
3. If it is desired to synthesize the VHDL model, synthesis must be tested at this point.
4. Creation of a new template folder and a new Python script based on a previous one.
5. Change of all modules and values inside *templ_leon3mp.vhd* and *templ_config.vhd,* that can be taken from SoCRocket using the **set_var** and **create_generic_map** functions.
6. Execution of script, verification of behavior — simulation and/or synthesis as needed.

These steps can also be generalized and used for other virtual platforms. They are not unique to SoCRocket. Figure 2 shows the current workflow for rapid prototyping and RTL validation. The dashed lines symbolize information feedback into the configuration process. This could be useful in an automated design-space exploration scenario.

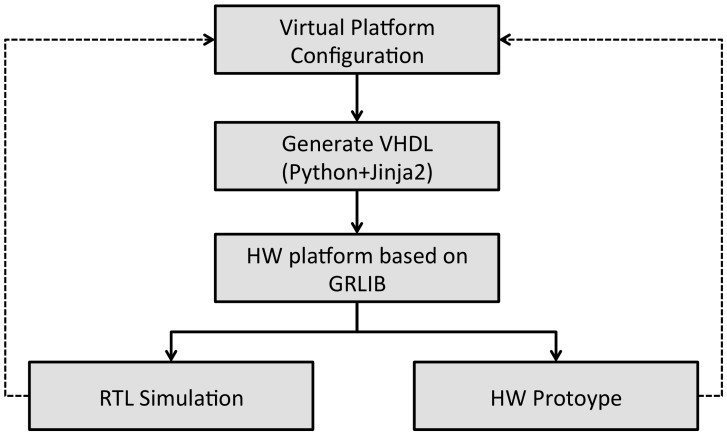

Figure 2 The workflow with included template engine VHDL generation

IV. DISCUSSION

Apart from being able to quickly create working hardware prototypes from a virtual platform configuration there are more applications for our presented workflow. Instead of opting for the hardware prototype, one could run RTL simulations with the generated HDL code in various external tools. You could create, for example, power or area estimations. This would then require to be able to process the reports generated by such external tool simulations.

V. CONCLUSION

We have presented an open and flexible workflow for VHDL generation from a SystemC virtual platform. The presented template engine and Python functions could easily be adapted to other virtual platform frameworks. The application is not just limited to rapid prototyping. The workflow could also be integrated into an automated design-space exploration workflow. Here you could generate power estimation values for a configuration from RTL simulations and feed back those values after normalization into the SystemC simulation and find an optimal low power configuration. Another application could be integration of universal verification methodology to verifiy whole system configurations and not just single components. Here the templates and Python scripts would need to be heavily modified. If you have ARM models and plan to run Linux on your system you could also automatically create device trees from your system configurations with a simple template and a Python script.

ACKNOWLEDGMENT

The authors would like to thank Tobias Rust and Henrik Lange for their support in the implementation and extensive testing of the workflow.

REFERENCES

[1] R. Collett and D. Pyle, “What happens when chip-design complexity outpaces development productivity?” McKinsey on Semiconductors, vol. 3, pp. 24–33, 2013.

[2] J. Wagner, R. Meyer, R. Buchty, and M. Berekovic, “A scriptable, standards-compliant reporting and logging extension for systemc,” in Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), 2015 International Conference on, July 2015, pp. 366–371.

[3] R. Meyer, J. Wagner, R. Buchty, and M. Berekovic, “Universal scripting interface for systemc,” in DVCon Europe Conference Proceedings 2015, Nov 2015. [Online]. Available: https://dvcon-europe.org/sites/dvcon-europe.org/files/archive/2015/proceedings/DVCon_Europe_2015_TA3_1_Paper.pdf

[4] IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tool Flows, Ieee std 1685–2014 (revision of ieee std 1685–2009) ed., IEEE, 2014.

[5] K. Cho, J. Kim, E. Jung, S. Kim, Z. Li, Y.-R. Cho, B. Min, and K.-M. Choi, “Reusable platform design methodology for soc integration and verification,” in SoC Design Conference, 2008. ISOCC ’08. International, vol. 01, Nov 2008, pp. I–78–I–81.

[6] E. Pekkarinen, M. Teuho, E. Salminen, and T. D. Hämäläinen, “Resolving parameter reference management in ip-xact using kactus2,” in Industrial Electronics Society, IECON 2015 - 41st Annual Conference of the IEEE, Nov 2015, pp. 002 765–002 770.

[7] J. F. L. Tallec and R. de Simone, “Scipx: A systemc to ip-xact extraction tool,” in Electronic System Level Synthesis Conference (ESLsyn), 2011, June 2011, pp. 1–6.

[8] T. Fischer, C. Kollner, M. Hardle, and K. D. Muller-Glaser, “Product line development for modular fpga-based embedded systems,” in Rapid System Prototyping (RSP), 2014 25th IEEE International Symposium on, Oct 2014, pp. 9–15.

[9] R. Franco and A. Allara, “Web template mechanisms in soc verification,” in DVCon Europe Conference Proceedings 2015, Nov 2015. [Online]. Available: https://dvcon-europe.org/sites/dvcon-europe.org/files/archive/2015/proceedings/DVCon_Europe_2015_TA3_3_Paper.pdf

[10] A. Ronacher, “The jinja2 templating engine,” http://jinja.pocoo.org/, 2016.

[11] T. Schuster, R. Meyer, R. Buchty, L. Fossati, and M. Berekovic, “SoCRocket – A virtual platform for the European Space Agency’s SoC development,” in Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), 2014 9th International Symposium on, May 2014, pp. 1–7.

[12] SoCRocket, “SoCRocket sources,” https://github.com/socrocket, 2016.

[13] R. Meyer, T. Schuster, B. Farkas, and J. Wagner, SoCRocket Documentation, http://socrocket.github.io/usermanual.html, TU Braunschweig, 2015.

[14] W. Ecker, M. Velten, L. Zafari and A. Goyal, "The metamodeling approach to system level synthesis," *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Dresden, 2014, pp. 1-2.