2018 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

Increasing Efficiency and Reuse in Modeling SystemC/TLM IPs Targeting Virtual Prototypes for Software Development

David Spieker, Thomas Schuster, Rafael Zuralski, Christian Sauer

Cadence Design Systems, Munich, Germany

{*dspieker|thomschu|rafael|sauerc*}*@cadence.com*

***Abstract*— High-level hardware description languages like SystemC allow the development of IP models on a high abstraction level, rendering them useful for development of Virtual Prototypes. The Transaction Level Modeling (TLM) open standard offers a base protocol with defined interfaces for interoperability and compatibility with different IP. In our work we demonstrate how far simulation models for peripheral IPs targeting software development can be automatically generated. Moreover, we discuss abstraction methods for modeling IP-specific behavior. The presented techniques are detailed on the example of a USB3.1 host controller and device model. Measurements show that across the modelled IPs approximately 85% of the required code can be automatically generated and reused, resulting in a considerable modeling efficiency increase.**

Keywords*— *SystemC, TLM, IP, IP-XACT, Virtual Prototype, USB, xHCI

I. INTRODUCTION

Virtual Platforms (VP) have become a commodity in different application domains, shortening time-to-market, and improving quality of results, by enabling software and test development before first hardware becomes available. In the last decade a variety of tools supporting different aspects of Virtual Prototyping have been commercialized (e.g. [1], [2], [3]). These tools support architects in system partitioning (hw/sw co-design), system assembly, high-level synthesis, simulation, design space exploration, and more. Boosts for this technology were the standardization of the SystemC language [4] in 2005 (IEEE 1666-2005), and the publication of the TLM2.0 transaction level modeling library [5] in 2008. The conjunction of standardized language and interoperability layer resulted in wide industrial acceptance. However, up until now the application of Virtual Prototyping is often hampered by lack of appropriate simulation IP. The majority of design IPs are still only available on register-transfer-level (RTL). Development time for missing TLMs can easily engulf the 'early-available' advantage of VPs. Tool support for generating recurring structures like register banks and bus interfaces is available but does not unburden designers from having to abstract and implement functionality. VPs for software development shall provide just enough timing accuracy to allow an operating system to boot and load appropriate drivers. Abstraction from behavior and timing is necessary, because systems must simulate at least close to real-time. Therefore, TLM proposes the use of techniques like temporary decoupling and direct memory interface (DMI).

In our work we have investigated the required functional abstractions for providing fast simulation models for several industry-relevant design IPs like USB3.1, PCIe, CSI, DSI, DDR, and SD. To increase model creation efficiency, we widely automated the work flow. Moreover, we identified generic reusable/shared constructs in communication IP models, resulting in a partially-automated TLM creation framework. All models have been integrated in a library based on joint infrastructure. Verification was done using a combination of stand-alone unit tests and software driver tests on an ARM subsystem. We show-case our modeling approach, including abstraction decisions, and verification on the example of a state-of-the-art USB3.1 IP.

II. APPLICATION

In general, most systems exhibit considerable similarities regarding the use of peripheral IPs. Communication modules need interfaces to connect to both, the system side and the protocol-specific communication mechanism. This can be used as a base for developing a unified module structure: Based on the non-proprietary, open TLM

standard, all peripheral controller modules contain at least a single socket for register set access and sockets for system-side communication. In addition, they implement a module-specific set of standard sockets to support their application protocol. To increase compatibility with other models, protocol-specific communication is carried out via TLM generic payload objects. These contain data according to their respective protocol specifications in a bit-true way, giving the module's users the possibility to further analyze and process the standard-conformant data packages.

A. Shared infrastructure / automatic generation

The resulting generic module structure is depicted in Figure 1. It can be used as a template to automatically generate skeleton models of peripheral IPs from IP-XACT or SystemRDL descriptions. Along the actual SystemC module, such a generator (similar to Cadence tlmgen, part of VSP [1]) should also produce a software API, e.g. containing definitions for register field locations, and unit tests of core functionality, like register bank read/write.

Although the complete behavioral description cannot be created automatically, basic parts can be included to let the developer focus on the integral functionality. In general, C++ is enabling to create a modular and maintainable structure by providing the concepts of Multiple Inheritance and Polymorphism. Those build up the base for almost all automatically generatable code fragments presented here. A list of structures was identified that can be pulled out of the manual behavioral description.

1) Register set

A module's specification in form of the IP-XACT or SystemRDL already contains all information about the register set to be used. Especially a register's offset to a base location and the access rights are conveyed. This information is enough to create the whole register set automatically. The register set implementation is based on the `sc_register` SystemC model library provided by *Cadence* [6]. It is composed of several components:

- A register class (`sc_register`)
- A register field class (`sc_register_field`)
- A register bank class (`sc_register_bank`)

Thanks to an integrated callback mechanism and instrumentation interface, an IP block's registers can be debugged within a tool that supports this interface like *Cadence*'s *SimVision*. A register set that is auto-created like explained before, must be accessible from the outside. This is accomplished by the usage of a standard TLM target socket. Since all registers' locations are known, an automatic mapping between incoming addresses and register instance can be calculated. Therefore, the module's developer does not have to implement it by herself. Supporting the register set is done by simply inheriting from a base class which itself implements all register instantiation and default address decoding.

2) Lifecycle management

As all modules traverse certain common phases, this can be standardized for all modules of a library. These phases in their entirety are referred to as a lifecycle. The states DISABLED and ENABLED can be considered as the smallest common basis. By using pure virtual function declarations and providing a default behavior for enabling and disabling in an interface class, an IP's developer is enforced to overload these functions with application-specific behavior. Moreover, this lifecycle interface can be used to specify a common way to reset a module. As a further improvement, the same interface could define a whole module state machine that is customizable. Relevant design patterns are available for this purpose. By using the described interface, an implementor is given a narrow outline that largely determines further development work with regards to lifecycle management. This automation of common module behavior in combination with a fixed development frame in consequence reduces the susceptibility to errors.

3) Sub-components

When implementing complex IPs, the division of functionality into better maintainable subcomponents is often chosen. This leads to the problem how external events can be propagated down into the hierarchy. The reset of a

whole module can be taken as an example. When an external request to reset comes in, this request should be propagated to all subcomponents so that they also can reset their environment to a pre-defined state. A dedicated subcomponent interface similar to the lifecycle management interface can be defined to support automatic code generation. The process of resetting all subcomponents by hand can be reduced to a repeated invocation of resetting functions in a list of subcomponent interfaces. Since there is no need to know anything about the handcrafted code, this code fragments can be reused among all IPs. A subcomponent only must inherit from the common interface and be registered in a list of subcomponents. This list can be instantiated by default within the lifecycle management interface as a recommended location.

4) Signal interface

A module that is supposed to be integrated into a Virtual Prototype needs a way to communicate with the Instruction Set Simulator (ISS). The physical implementation serves as a reference here by using dedicated signal lines for interrupt handling and signaling a reset for example. In SystemC a simple `sc_signal` channel could be used to model this functionality. However, since our modelling abstraction focuses on the loosely-timed modeling style, there is no need for an additional delta-cycle between raising/lowering and registering a value change which is implied by the `sc_signal` semantics. To standardize a signaling interface, a draft standard was proposed introducing the `tlm_wire [7]`. In case of the implemented library, all modules have a single reset pin. This leads to a common resetting interface that defines a single `tlm_wire_in` port and a pure virtual function the developer must implement with the module's resetting behavior. Together with the mentioned lifecycle management that also defines a resetting functionality the developer either can choose to extend the resetting behavior when a reset is requested from the outside via the reset pin or leave it unchanged reusing the already implemented behavior.

5) Host-side memory interface

Protocol IPs need access to data stored in memory. Therefore, a TLM initiator socket can be used for data access. To enforce a common naming convention, this socket and all needed callback functions can be defined in another interface class. In addition, data access by an IP resembles the function of a DMA controller, although a distinction should be made. For a simple DMA, the controller will be configured to fetch data from a specific location and store it in another location. In general, these locations are somewhere in the system memory. When modeling a protocol IP, the IP itself often is the producer or consumer of data. Consequently, data management can be divided into a storing and a fetching interface. In combination, a DMA-like module can be assembled. Most implemented IPs need both, storing and fetching. A combined interface class can provide a common data access interface for all modules.

6) Protocol-side interfaces

Due to the large differences between communication protocols, the application-specific interfaces cannot be generalized. This also includes the number of sockets to be used. In some cases, a single initiator socket is sufficient for a protocol master. In others, even multiple target sockets must be used. However, polymorphism and inheritance can be used here as well, to define common communication data flow. Master and slave interface classes are generated that automatically instantiate the required number of sockets with the corresponding callback function stubs. A module's developer must override these.

Like mentioned before, the transactions carry data to be transmitted in a bit-true way. This implies that the data format used within a model can differ from the bit-true representation. This is the case for all IPs. A pre-defined serialization/deserialization interface can further unify a model's structure. While this does not lead to greatly reduced manually created code, a unified code structure is more maintainable. Therefore, the time required to become familiar with a module's functionality is greatly reduced. The serialization/deserialization interface can be integrated into the master and slave interface classes. In addition, a base class for data packets is provided to be shared among all IPs. It standardizes access to meta data like packet type information or the data length.

7) Analysis interface

With the TLM analysis port, the TLM standard provides a way to broadcast messages to several connected modules. This port can be integrated into a dedicated class that also defines some additional functions. These functions unify the transmitted message format, so that messages from different IPs are uniform containing

important information and statistics like the originating module's name and the simulation time stamp. The messages' uniformity is the basis for an automated processing of messages. Like for the processing of report messages, a message verbosity can be defined. The analysis interface can be made to cooperate with the customizable logging mechanism explained in II.A.8) below.

8) Logging

The logging implementation is realized as a base class to be inherited from. This enables to get a module's name from the SystemC hierarchical name and prepend it to every logging message. A class/module inheriting the logging base class has access to different functions that define distinct message verbosity levels. For example, Listing I shows how to send out a message with the DEBUG verbosity level and its corresponding output.

```
log_DEBUG()() <<
      "This is a message with DEBUG verbosity level";
```

```
Info:   0.000018s | usbssp_0: SSP driver loaded
```

Listing I Logging example

9) Parameterization interface / CCI

IPs often have a fixed set of functionalities that can be customized by setting different configurations. Therefore, developing a generic base that then can be specialized gives more flexibility in later use. To standardize the methodology to configure and inspect a model, the Accellera consortium has established a dedicated working group called Configuration, Control and Inspection (CCI) WG [8]. With a CCI draft standard available, tool providers have started to support it. Consequently, the proposed parameterization interface is based on this draft. Typically, a module's timing is dependent on the chosen implementation technology and the communication. A basic configuration may therefore contain at least a CCI parameter defining the register bus clock. With this information, the reading/writing delays can be calculated and modeled for simulation. However, other settings like operation modes or buffer sizes are also conceivable to be implemented using CCI parameters.

The common infrastructure, especially the analysis interface II.A.7) and the logging base class II.A.8) above also include several CCI parameters for verbosity level and output format customization.

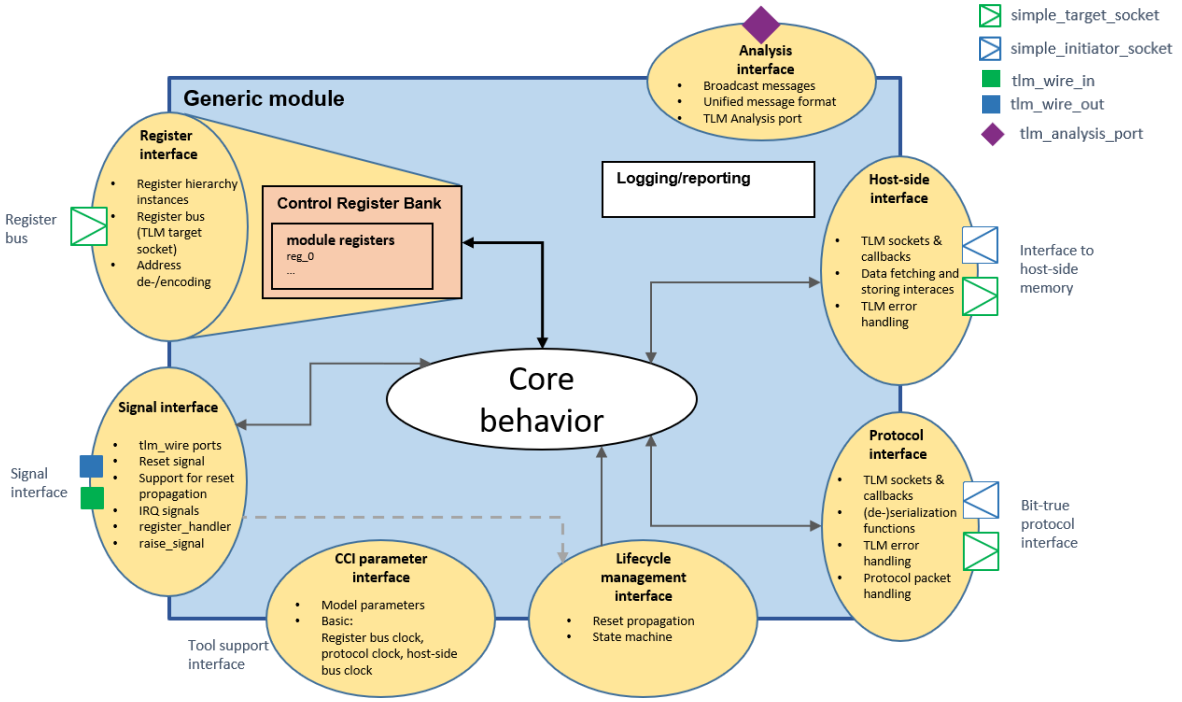

Figure 1 - Generic Module Structure

B. Abstraction of function and timing

Functionality and essential timing are specific to an IP and can therefore not be automatically implemented. Methods for generating behavior from RTL [9] or high-level specification/algorithms [10] exist, nevertheless a prevailing share of manual modeling work is required. We will use USB3.1 as an example for explaining required considerations.

With simulation performance in mind, all models targeting software development should be implemented using as few SystemC threads as possible. Since the USB protocol is mostly a polled protocol controlled by a host controller, the blocking interfaces suffice. However, deadlocks can occur when transactions are in flight and not responded to by a USB peripheral. The eXtensible Host Controller Interface for Universal Serial Bus (xHCI) specification [11] describes a command queue mechanism in which both participating modules must be biased with a transaction to maintain the conversation. This leads to the requirement that – with this model – the remote participant's firmware must answer a request in a timely manner or even bias its controller before the request arrives. Otherwise, blocking of the host controller as the master in the USB can occur. Based on this and other observations the behavior of the USB TLM is defined and hooked-up to callbacks and ports of the generic base module (Figure 2).

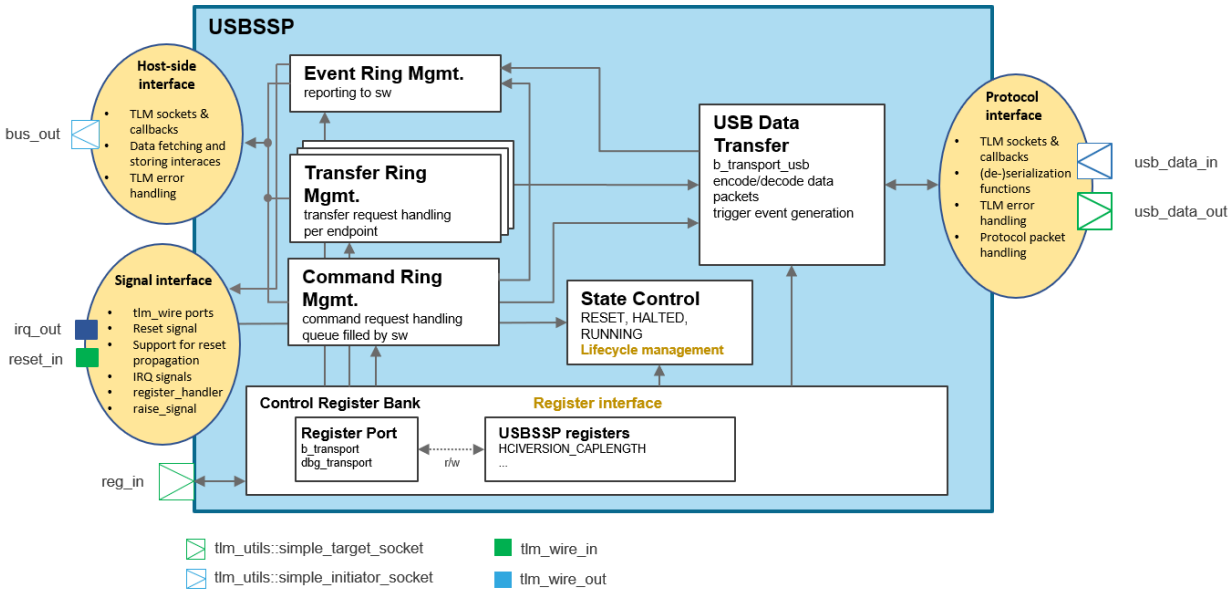

Figure 2 - Structure USB TLM

1) Ring management subcomponents

Software communicates with the controller via data structures placed in system memory and the register bank. The processor tells the controller where those data structures are placed by writing special configuration registers. A command to be executed by the IP must first be assembled by the processor after which the processor signals the controller to start processing the commands queued in memory. This signaling is called doorbell ringing where a doorbell register is written. While this communication scheme is shown on the example of the USB controller, the doorbell mechanism is common to other protocol IPs as well. It requires both, a register callback for reacting on a doorbell register write and a host-side memory interface fetching data from system memory. Both functionalities are contained in the common infrastructure explained in II.A above. The developer only needs to instruct the memory fetching interface to load the command data structures from a location specified in a configuration register. This is done by creating an additional thread that sequentially reads the command data structures and invoking the corresponding C++ functions. A thread is needed to keep the model responsive with regards to register writes.

While this demonstrates the normal use case, the controller can be turned off either by writing a status register or resetting it completely. The state machine implemented in a pre-defined base class explained in II.A can be used to prevent normal operation. The automatically generated reset signal propagation requires the developer to implement a single handler method per subcomponent. In the example of the USB controller, the different ring

management submodules store ring index pointers among other information. These must be reset to their default values to get to the initial state. This is an example for the tasks the developer must implement in a reset method.

While the Command Ring Management subcomponent is the consumer of data provided by the processor, the Event Ring Management submodule reports information to the processor. Therefore, it produces data to be stored in memory. In contrast to the data-fetching interface for consuming, the provided storing interface can be reused that is instructed to place e.g. command completion information in system memory. The location is again configurable in a module register.

Both, Command Ring Management and Event Ring Management subcomponents usually are instantiated only once. In contrast to that, since there can be multiple targetable data sinks or sources on a USB peripheral, there also can be multiple Transfer Ring Management subcomponent instances. They are very similar to the Command Ring Management sequentially looping through the transfer requests.

2) Data transfer component

To send and receive data over the protocol-specific connections, the pre-defined protocol-side interface can be reused. The USB IP is supposed to implement both, the host-side and peripheral-side controller. Therefore, it requires to have an initiator and target socket. The interface base classes need to be specialized to support the protocol-specific behavior. The conversion into a generic payload object used by TLM socket communication is done automatically as well as the TLM error handling. To support the USB packet types, the data packet base class mentioned in II.A.6) above is inherited from. As an example, the USB protocol defines data and acknowledgement packets. Both are derived from the base class, providing meta information. This can be used in the core code to react to packets. Access to the meta information is governed by the pre-defined member functions of the packets' base class. While this property is not exploited for automatic code generation yet, it can serve as a basis for future improvements.

3) Parameters

All developed IPs must communicate with the system side and the peripheral side. Every data interface contains a CCI parameter specifying its data clock period. Based on this, a module can calculate access times and set the TLM delay argument accordingly. As mentioned before, the USB IP can be set either in host or peripheral mode. This configuration also is implemented by using a CCI parameter. Shown as an example for the USB core, all configurations in the developed library are using CCI parameters. Especially verbosity control for reporting messages is done the same way for all modules, simplifying IP debugging.

III. RESULTS (PRELIMINARY)

A. Modeling effort

As discussed, big portions of the module infrastructure can be reused or automatically generated. In terms of 'lines of code' (LOC) generated infrastructure, predominantly register files, make up 77-95% of the considered IPs. The quota manually created accounts only for about 11.64% on average (TABLE I.).

TABLE I. LINES OF CODE COMPARISON

IP model	**Lines of Code**			
	complete	***generated***	***manual***	***manual quota***
USB	45977	43335	2642	5.75%
PCIe	79638	74773	4865	6.11%
MIPI CSI2RX	5225	4289	936	17.91%
MIPI CSI2TX	4428	3867	561	12.67%
MIPI DSI	7339	6590	749	10.21%
LPDDR4	20413	19214	1199	5.87%
SD4HC	6840	5268	1572	22.98%

IP model	Lines of Code			
	complete	*generated*	*manual*	*manual quota*
				11.64%

B. Simulation performance

The modules' simulation performance was evaluated in a test bench using the USB IP. Connected back-to-back and operating in either host mode and peripheral mode, data of different length up to 8 kB was transmitted and received. The peripheral side was configured to be used as an echoing device by software running on an ARM ISS. The test bench structure is shown in Figure 3. While the host system CPU needed 5.362773 seconds to finish the simulation, about 63% were needed for non-design-specific simulation infrastructure like the SystemC kernel and profiling routines, leaving 1.980072 seconds for processing the design. This again is divided into 89.9% for simulating the ARM processor ISS and 0.199987 seconds (10.1%) for the actual USB IP simulation. This is shown in Figure 4.

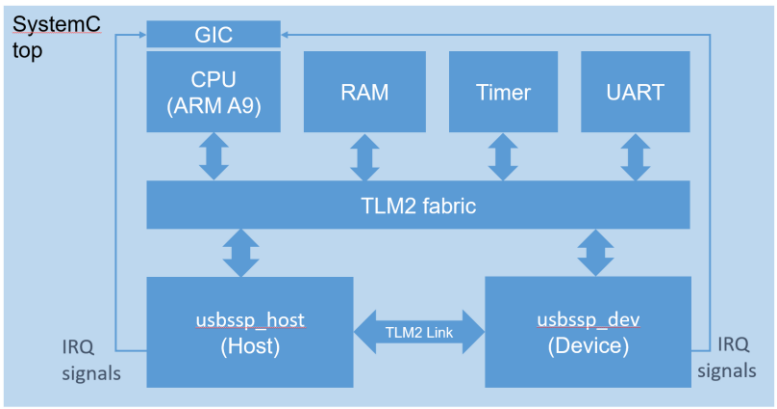

Figure 3 - Testbench layout

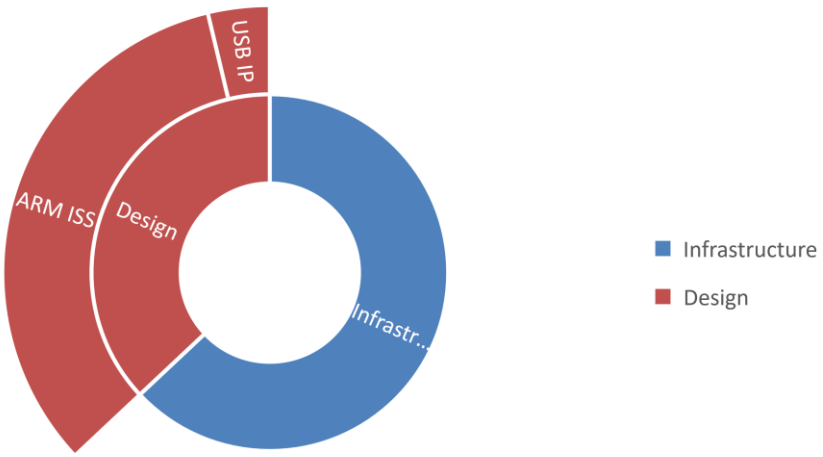

Figure 4 - Simulation performance split

IV. CONCLUSION

As we have shown, the high extent of similarities in communication IPs can be exploited to create a widely automated TLM IP generation flow. With a high extent of reusable code, the framework also facilitates the development of the protocol-specific behavioral part of the code. Using these common structural elements,

additional protocol IPs can be generated quickly for further system extensions. To be usable for software development, the models have to provide a high simulation performance. We have shown how IP behavior can be abstracted in a structured way and highlighted our approach on the example of a USB3.1 TLM.

V. REFERENCES

[1] Cadence Design Systems, "Virtual System Platform," Cadence Design Systems, 2018. [Online]. Available: https://www.cadence.com/content/cadence-www/global/en_US/home/tools/system-design-and-verification/software-driven-verification/virtual-system-platform.html. [Accessed 2018].

[2] C. Sauer and H. P. Loeb, "A lightweight infrastructure for the dynamic creation and configuration of virtual platforms," in *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, 2015.

[3] T. Schuster, R. Meyer, R. Buchty, L. Fossati and M. Berekovic, "SoCRocket - A virtual platform for the European Space Agency's SoC development," in *9th International Symposium on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, Montpellier, 2014.

[4] Open SystemC Initiative, *SystemC 2.0.1 Language Reference Manual*, San Jose, 2003.

[5] Open SystemC Initiative, *OSCI TLM-2.0 LANGUAGE REFERENCE MANUAL*, 2009.

[6] Cadence Design Systems, Inc., *Virtual System Platform Reference*, vol. Version 18.03, San Jose, 2018.

[7] S. Swan and J. Cornet, "Beyond TLM 2.0: New Virtual Platform Standards Proposals from ST and Cadence," in *NASCUG at DAC*, San Francisco, 2012.

[8] T. Wieman, B. Bhattacharya, T. Jeremiassen, C. Schroder and B. Vanthournout, "An Overview of Open SystemC Initiative Standards Development," *IEEE Design & Test of Computers*, vol. 29, no. 2, pp. 14-22, April 2012.

[9] ARM Inc., "User Manual Carbon Model Studio 8.1," ARM Inc., 2016. [Online]. Available: infocenter.arm.com. [Accessed 2018].

[10] The MathWorks, Inc., "Generate C Code from Simulink Model," [Online]. Available: https://www.mathworks.com/help/dsp/ug/generate-c-code-from-simulink-model.html. [Accessed 2018].

[11] Intel Corporation, *eXtensible Host Controller Interface for Universal Serial Bus*, 2017.