Automated Integration of MathWorks® Simulink® Signal Flow Graph Models into Synopsys® Virtualizer™-based Virtual Prototypes

Andreas Mauderer*, Alexander Schreiber+, Jerome Chevalier‡, Jan-Hendrik Oetjens*

*Robert Bosch GmbH
AE/EID2, AE/EIM
Reutlingen, Germany

+The MathWorks GmbH
Application Engineering
Ismaning, Germany

‡The MathWorks, Inc.
SPC / HDL Verification
Natick, MA, USA

ABSTRACT

The rapid progress of Advanced Driver Assistance Systems (ADAS) increases the importance of automotive sensors, leading to stringent requirements and reduced time-to-market. MEMS sensors are usually integrated as System-in-Package (SiP), with a signal processing System-on-Chip (SoC) ASIC (Application-Specific Integrated Circuit). Virtual Prototyping is an established approach for early validation of complex SiP/SoC concepts and for early software development. In this paper, we present an improvement to our IEEE 1685™: IP-XACT-centric workflow for Virtual Prototype (VP) creation [1]. *This approach was developed in close collaboration between Bosch and MathWorks®. In the approach, extended IP-XACT register descriptions provide input to MathWorks code generation tools to automatically produce C/C++ code for e.g. signal processing functionality and SystemC® TLM 2.0 wrappers with Synopsys® SystemC Modeling Library (SCML) registers. These components, designed and verified using Model-Based Design (MBD) and MathWorks MATLAB® and Simulink® software, can directly be integrated into a Synopsys® Virtualizer™ architecture. We show how the proposed approach accelerates VP construction time and ensures functional consistency by applying it to an industrial-grade ASIC design.*

Table of Contents

Table of Figures

Trademark Attribution

Accelera, SystemC and IP-XACT are trademarks of Accellera Systems Initiative Inc.

BOSCH is a trademark of Robert Bosch GmbH.

IEEE and IEEE 1685: IP-XACT are registered trademarks or trademarks of IEEE.

Magillem is a registered trademark of Magillem S.A.

MathWorks, MATLAB, Simulink, Stateflow, DSP System Toolbox, Simulink Coder, Embedded Coder, HDL Coder and HDL Verifier are trademarks or registered trademarks of The MathWorks Inc.

Oracle and Java are registered trademarks of Oracle and/or its affiliates.

Synopsys, Virtualizer, Platform Architect and VCS are trademarks or registered trademarks of Synopsys Inc.

Other product or brand names may be trademarks or registered trademarks of their respective holders.

1. Introduction

Automotive sensors are an integral part of arising new technologies like Advanced Driver Assistance Systems. These sensors provide information about the physical environment for enabling the car to perform the appropriate actions depending on the current situation. Thereby, signal processing ASICs are used for preprocessing sensor signals and for providing the sensor data to the microcontrollers on Electronic Control Units (ECUs). These signal processing ASICs are heterogeneous mixed-signal SoCs containing different kinds of on-chip components like hardwired digital components, processors, memories, bus architectures as well as one or more signal path components. A signal path component usually performs the main task of processing sensor signals. It consists commonly of A/D converters, digital hardwired logic and digital signal processors.

Figure 1 shows an example on-chip architecture of a sensor signal processing ASIC. The architecture contains a General Purpose Processor (GPP) with its associated RAM and ROM which performs tasks like bootloading or safety monitoring. There is also a signal path component performing the sensor signal processing. Thereby, the signal path has got an external port to the sensor. Besides, the ASIC contains a hardware safety controller as well as external interfaces such as CAN, SPI and PSI. All of these components are interconnected by two on-chip buses and a bridge.

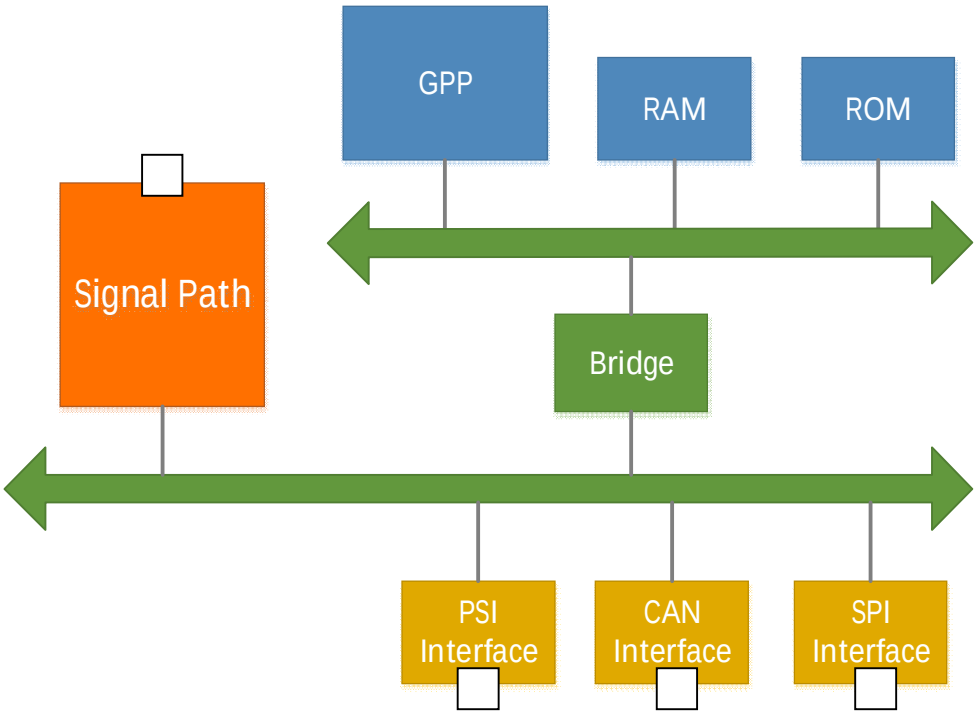


Figure 1: Example On-Chip Architecture of a Signal Processing ASIC.

With the ongoing advancement of Advanced Driver Assistance Systems, the requirements for automotive sensors are becoming more and more stringent and also the time-to-market of these sensors is reduced. This leads to a permanently increasing complexity of sensor signal processing ASICs as well as an increasing pressure of time for the development of the ASICs. An established approach to address these challenges is Virtual Prototyping which enables for example an early validation of system concepts or an early hardware/software co-design. In order to provide VPs in early project phases, methods for the acceleration of the construction of VPs are needed. This work focuses on extending our existing development flow in order to even further accelerate the construction of VPs.

VPs are fully functional software models of physical hardware systems. VPs of Systems-on-Chip are usually described in SystemC and serve at early design stages for example as executable specification or are used for concept validation and architectural exploration. A further use case for VPs is the early development of on-chip software which enables the software developers to develop software in parallel to the RTL (register-transfer level) hardware development process.

VPs are focusing on the architectural system perspective abstracting communication interfaces using e.g. Transaction Level Modelling (TLM) techniques [2].However, signal path components which are a crucial part of signal processing ASICs are usually developed by using the Model-Based Design (MBD) methodology [3]. Thereby, Model-Based Design focuses on the algorithmic system perspective and the associated signal flow.

The components of the signal processing path are modelled on different levels of abstraction. The signal processing algorithm which will be implemented on the ASIC, is embedded in its environment. The models of the stimulating components allow to assess different verification and fault injection scenarios. Output signals of the signal processing algorithm can also be fed back to the stimulating component forming a closed-loop control system. The functionality and robustness of the algorithm can therefore be verified early through simulation and analysis in the time and frequency domain.

The signal path shown in Figure 1 can be detailed like follows (see Figure 2):

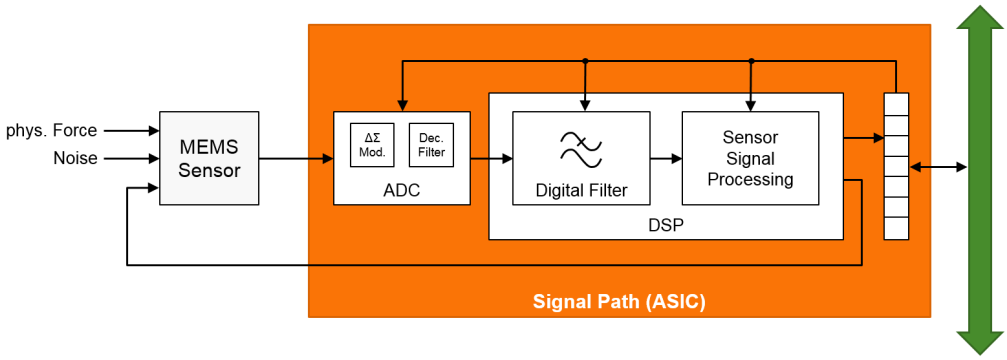


Figure 2: Detailed signal path within its environment

- MEMS (Micro-Electro-Mechanical System) sensor
  The MEMS sensor transforms physical forces (e.g. pressure, acceleration) into electrical signals. Its behavior is modelled by mathematical formulas using MATLAB. Besides the primary input of the physical force value it is also possible to inject fault signals, e.g. noise caused by additional mechanical vibrations.

- Analog-to-Digital Converter (ADC)
  The analog, i.e. time- and value-continuous signal delivered by the MEMS sensor needs to be sampled and quantized. This is done by e.g. a Delta-Sigma Analog-to-Digital Converter (ADC). Such an ADC represents a mixed-signal and multi-rate component in itself and consists of the delta-sigma modulator and a digital decimation filter. The behavior of the ADC is modelled by its differential (time-continuous part) and difference (time-discrete, i.e. sampled part) equations using basic Simulink blocks.

- Digital Filter
  The output signal of the MEMS sensor can contain additional signal components like noise which is caused by the environment the MEMS sensor is operated in or by the sensor itself. These signal components need to be filtered out using appropriate digital filters. The filter

architecture, its coefficients and the fixed-point quantization are designed and verified using MATLAB and exported as a filter block to Simulink.

- Sensor Signal Processing
  The filtered signal is the input for the functional processing of the MEMS sensor signal.

During these first steps of the algorithm development and verification the physical boundaries of the different components of the overall signal processing path are described on a high level of abstraction. The components are exchanging their information through either time-continuous or time-discrete signals using either value-continuous or value-discrete data types. Within this stage it is not determined yet if the algorithms are implemented as software running on a processor or in hardware.

For accelerating the construction of VPs, the fact that we develop and verify signal path components using the MBD methodology can be exploited for the construction of VPs. In this paper, we present an approach of the automated integration of existing Simulink models representing signal path components into VPs using TLM based interfaces.

2. State of the Art

For our ASIC development, we use an IEEE 1685™: IP-XACT-centric workflow for generating multiple kinds of descriptions from IP-XACT [4]. Here, we generate HTML documentation, RTL descriptions in VHDL, header files for software development and test descriptions (see Figure 3). This flow ensures the consistency between different ASIC descriptions and also reduces the manual effort of creating different descriptions containing similar information.

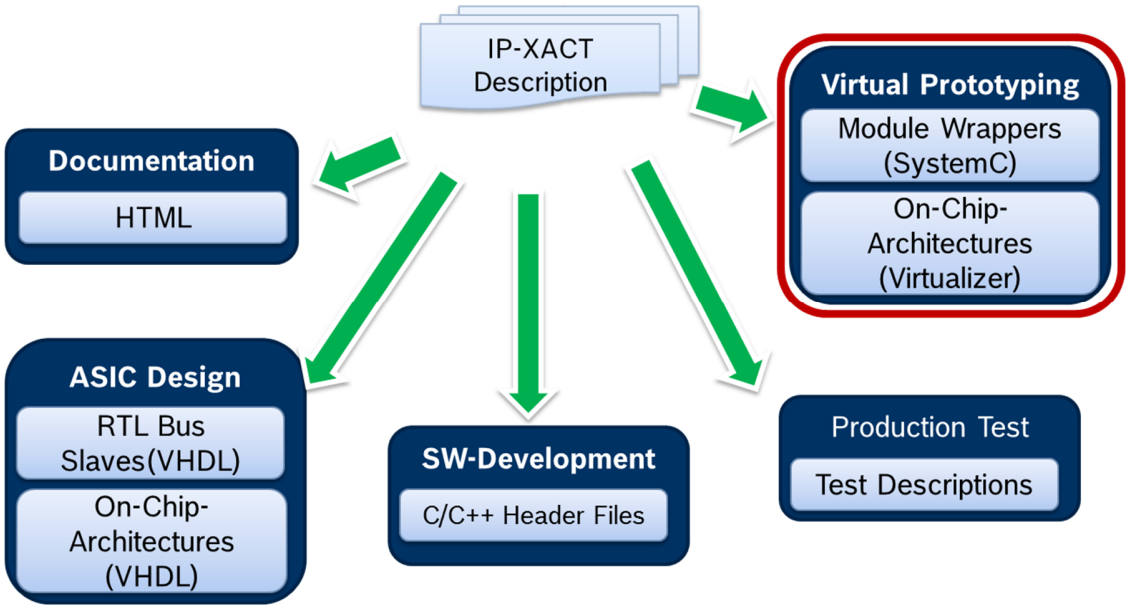


Figure 3. IP-XACT-centric workflow for generating different ASIC descriptions [1]

In order to accelerate the construction of VPs as well, we extended the workflow with the generation of VP components [1]. This workflow is depicted in Figure 4 and combines two approaches: On the one hand, we generate Virtualizer Tcl Scripts which create complete Synopsys Virtualizer architectures. These Tcl Scripts are generated using Java® Emitter Templates (JET) [5] based on

IP-XACT design descriptions. On the other hand, we use Synopsys TLM Creator for generating SystemC TLM2.0 [6] [7] module wrappers containing SCML [8] register interfaces. These wrappers represent on-chip components based on IP-XACT register interface descriptions. The functionality of these components can be described manually in SystemC and can then be integrated into the generated module wrappers.

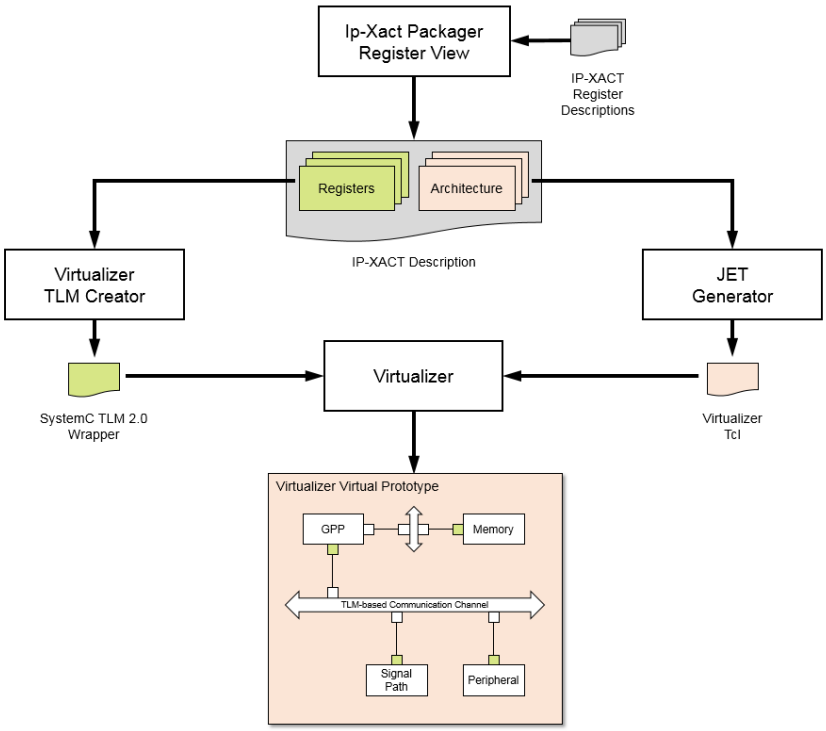


Figure 4: IP-XACT-centric workflow for generating VPs described in [1]

By using this workflow, VPs containing on-chip communication architectures as well as module wrappers with register interfaces can be generated from IP-XACT descriptions. But the approach does not comprise a solution for generating component functionality such as signal processing functionality. Therefore, the functionality of on-chip components has to be modelled manually.

On the other hand, the functionality of the signal processing components is already available in the signal flow representation using MATLAB [9] and Simulink [10] within the step “Algorithm Design” of MBD (see Figure 5) [3]. The signal processing algorithms have been embedded within their environment of surrounding components. Their functionality and robustness have been verified against their requirements through early simulation. Different algorithmic architecture alternatives can be evaluated which are e.g. already optimized for either a software or hardware based implementation. The effects of fixed-point quantization on the overall algorithm performance can also already be assessed and optimized through bit-true simulation.

These refined algorithm descriptions in MATLAB and Simulink are the basis of the next step within MBD, the “Algorithm Implementation” through automatic code generation targeting either C/C++ using Simulink® Coder™ [11], Embedded Coder® [12] or Hardware Description Languages (HDL) like VHDL or Verilog using HDL Coder™ [13]. The automatic code generation guarantees bit-true and cycle-accurate functional equivalence between the signal flow representation within the MATLAB and Simulink simulation environment and the implementation on the physical target platform. Therefore it avoids the introduction of functional differences which is common for manual coding of

the algorithms. Additionally, changes of the requirements can also be implemented more quickly based on the modified signal flow description which serves as reference for the subsequent steps within the MBD methodology.

The primary intention of this automatically generated source code is the implementation of the algorithm on the respective hardware or software target platform. HDL Coder generates generic, i.e. technology independent, synthesizable Register-Transfer-Level (RTL) HDL code based on the quantized, time-discrete algorithm description in MATLAB and Simulink. Event-triggered control logic can also be described using Stateflow® [14]. When using a value-continuous, time-discrete model instead, HDL Coder generates a RTL model which cannot be synthesized as it uses the "real" data type but which can be used as behavioral HDL model. This approach has been used in the past [15] to generate time-discrete functional models of signal processing components.

The Embedded Coder generates by default generic ANSI C or C++ code which can be used either to target a variety of different embedded processor devices without the need for code regeneration or for pure functional modeling purposes, e.g. the functional core of a VP component. In the latter case, the TLM compatible interface and the coupling between the interface and the functional core needs to be done manually or by using the Virtualizer TLM Creator (see Figure 4).

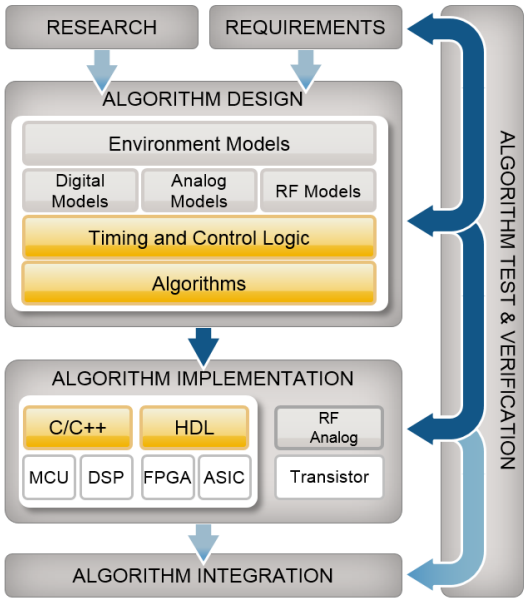


Figure 5: Overview Model-Based Design (MBD) [3]

3. Generating Sensor Signal Processing Components from Simulink

In chapter 2. , we introduced our IP-XACT-centric workflow for the generation of Virtualizer architectures including SystemC module wrappers based on IP-XACT design and component descriptions. Concerning the architecture and register description creation, the starting point for a signal processing ASIC design is the definition of the on-chip architecture in our workflow. This architecture is developed by defining on-chip components and by identifying an appropriate communication architecture which fulfills the given throughput requirements. Such a communication architecture can be determined for example by using Synopsys Platform Architect™ which provides processor models, abstract data sink and source modules as well as bus IP for design

exploration using traffic generation and exhaustive performance analysis. An IP-XACT architecture description can then be created graphically in Magillem®Ip-Xact Packager (MIP) [16], [17]. Furthermore, register interfaces of the on-chip slave modules have to be defined based on module functionality that has to be accessible from the on-chip bus. These register interface descriptions can be created in Magillem Register View (MRV) [18] and are saved as IP-XACT component descriptions. From these IP-XACT descriptions, we can already generate Synopsys Virtualizer architectures as well as SystemC module wrappers as described in Chapter 2. In this chapter, we will describe a further advanced method in order to generate signal path models based on Simulink models and IP-XACT register descriptions. Figure 6 shows our extended VP generation workflow. The left branch in Figure 6 shows the workflow for the generation of Synopsys Virtualizer architectures. The workflow for the module wrapper generation can be found in the middle two branches and the right branch depicts the new method for generating signal path models based on the existing algorithm design using MBD.

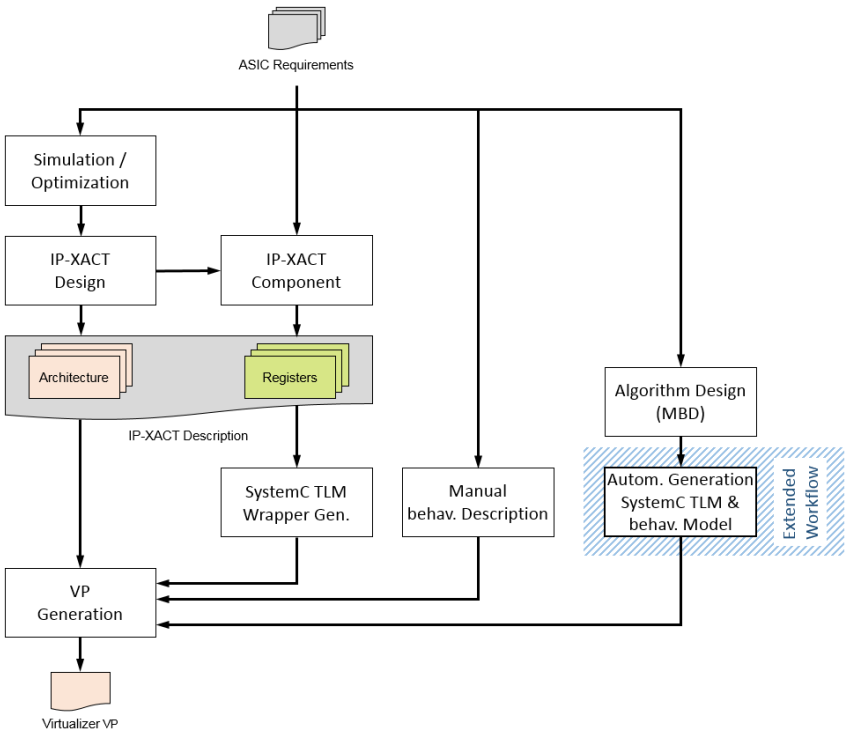


Figure 6: Extended VP generation flow

We also introduced the classical MBD workflow in chapter 2. It starts with the representation of the requirements on the algorithm under development as executable specification. The algorithm is embedded in an as realistic as possible environmental model serving as test bench for early and continuous verification and as reference for further architectural and algorithmic design exploration. The goal is to identify the optimal solution for the given specification and to assess the impacts of imperfect design effects caused by e.g. architectural and algorithmic choices for a more efficient implementation on the target platform (hardware or software) or the reduced precision of fixed-point data types. The refined and optimized algorithm is implemented through automatic code generation (C/C++, HDL) which is then verified through different kinds of equivalence simulation like Software-in-the-Loop (SIL), Processor-in-the-Loop (PIL) or External Mode simulation in case of C/C++ or like HDL Co-simulation and FPGA-in-the-Loop (FIL) simulation in case of HDL, either kind reusing the existing reference test bench of the “Algorithm Design” stage.

As already identified in chapter 2. , this classical MBD workflow is mainly focusing on the algorithm implementation rather than providing a tight integration to verification workflows for SoC devices using VPs. While a functional description of the algorithm can be generated automatically, it still needs to be integrated manually in e.g. SystemC TLM compliant wrappers which are either manually written or automatically generated by e.g. Virtualizer TLM Creator.

In order to further automate the generation of functional models of algorithms targeting SoC-level VPs within the MBD workflow, the classical workflow has been extended by a “Model Generation” step and additional C/C++ code generation targets (see Figure 7) for Simulink® Coder™ or Embedded Coder® in combination with HDL Verifier™ [19]. Figure 7 shows this model generation step embedded in the model-based design workflow, which corresponds to the right branch in Figure 6. The functional behavior of the component’s algorithmic core is delivered by the classical automatic C++ code generation workflow and is automatically integrated in the also automatically generated SystemC TLM 2.0 compliant wrapper. Additional information like register address mapping can also be taken into account as input for the automatic code generation as e.g. IP-XACT register description files or be provided as an output to facilitate the component’s integration into an existing IP-XACT-based framework.

Besides the SystemC TLM 2.0 compliant models targeting VPs also IEEE 1800™-2012: SystemVerilog [20] Direct Programming Interface (DPI) models can be automatically generated on the basis of Simulink models. These can be used e.g. for SystemVerilog-based verification based on UVM [21] [22] and in mixed-signal simulation environments like Synopsys VCS AMS [23].

In the following chapters we will demonstrate the different steps of the whole workflow using an example based on a signal processing ASIC for an acceleration sensor.

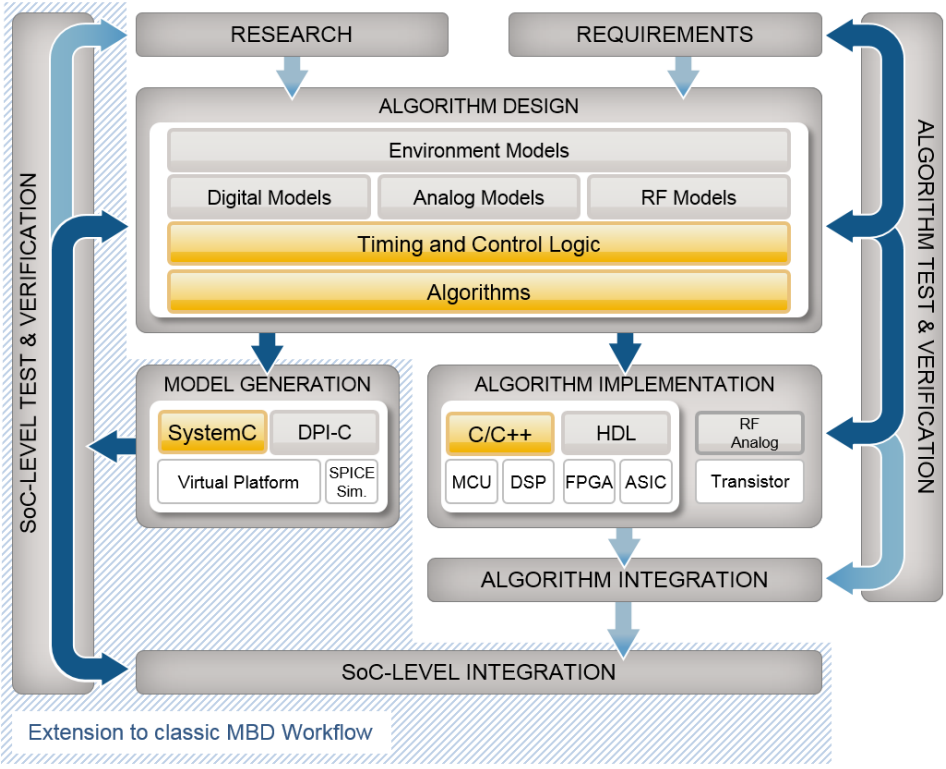


Figure 7: Overview extended Model-Based Design (MBD)

3.1 Architecture and Register Interface Design

In the following, we will describe the architecture and register description design flow using the example of an acceleration sensor signal ASIC whose architecture was already presented in Figure 1. Magillem Ip-Xact Packager [16], [17] provides a graphical interface to create an on-chip architecture that leads to an IP-XACT design description. Figure 8 shows the on-chip architecture of the ASIC which fulfills the throughput requirements of the design in Magillem Ip-Xact Packager. This architecture corresponds to the architecture already described in Chapter 2. The architecture can be developed by exploring different on-chip architectures and by analyzing the satisfaction of throughput requirements. After that, the register interfaces of the on-chip components have to be defined. This can be achieved in Magillem Register View [18] or by manually writing the IP-XACT component descriptions.

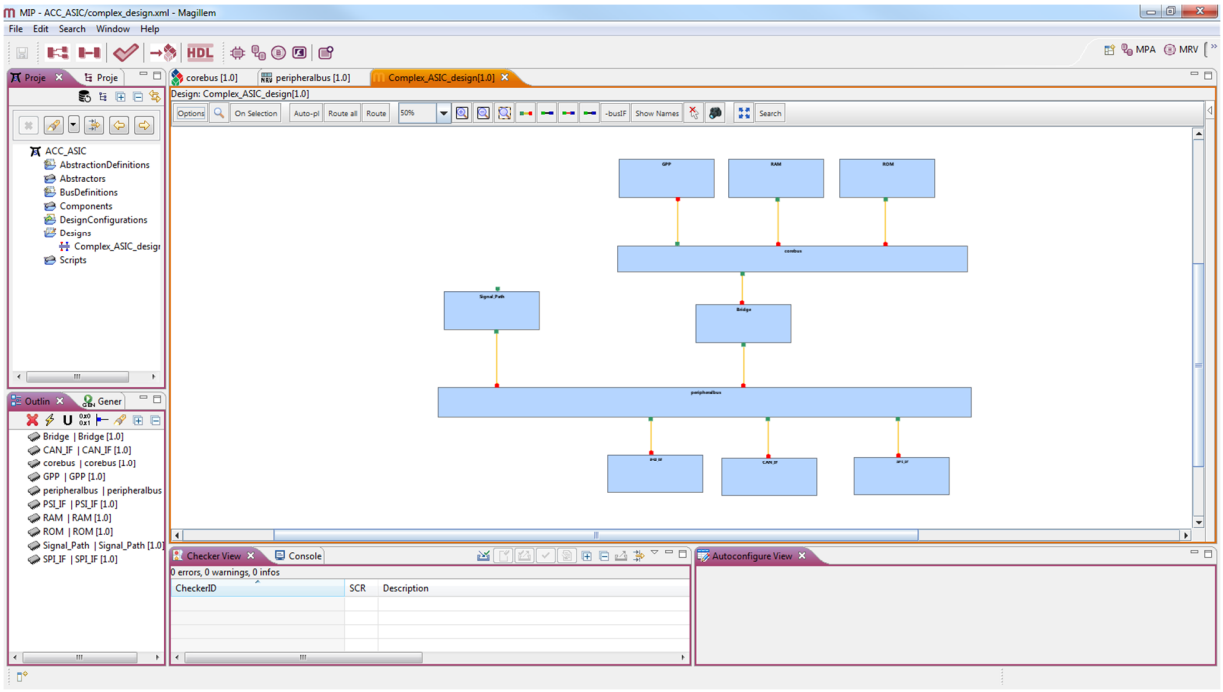


Figure 8: On-chip architecture in Magillem Ip-Xact Packager

3.2 Algorithm Design

In the “Algorithm Design” step we will design and verify a digital FIR lowpass filter using the “filterbuilder” tool [24] of MathWorks DSP System Toolbox™ [25]. This FIR filter serves as an abstract example of a signal path of a sensor signal processing ASIC. The following specification will use the normalized Nyquist frequency as reference for the filter characteristics:

- Lowpass filter
  - FIR filter response using the equiripple design method
  - 6dB-cutoff frequency: 0.25
  - Stopband attenuation: 60 dB
  - Passband ripple: 1 dB

3.2.1 Digital Filter Design and Analysis

The “filterbuilder” tool provides a Graphical User Interface (GUI) to the underlying MATLAB functions to conveniently specify and analyze the digital filter characteristics. The filter specification

is entered in the appropriate input fields as shown in Figure 9 for the lowpass filter and the tool calculates the corresponding filter coefficients (Figure 10b).

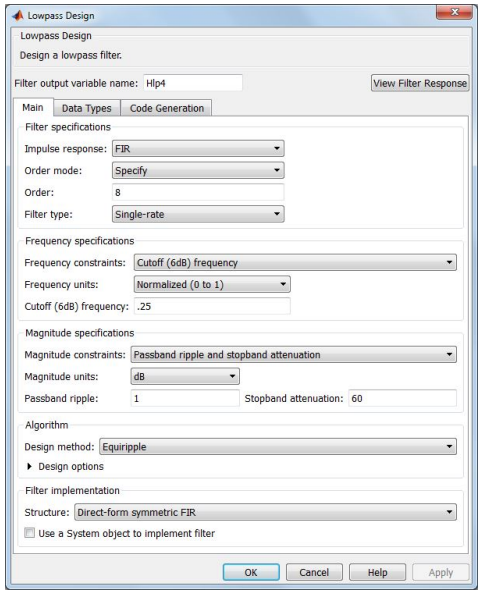
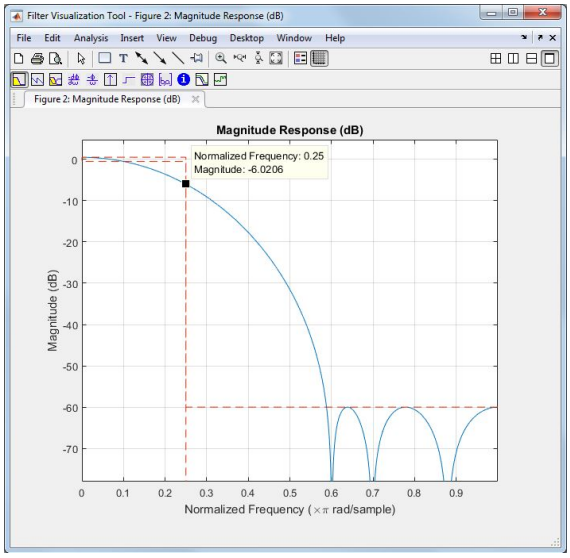
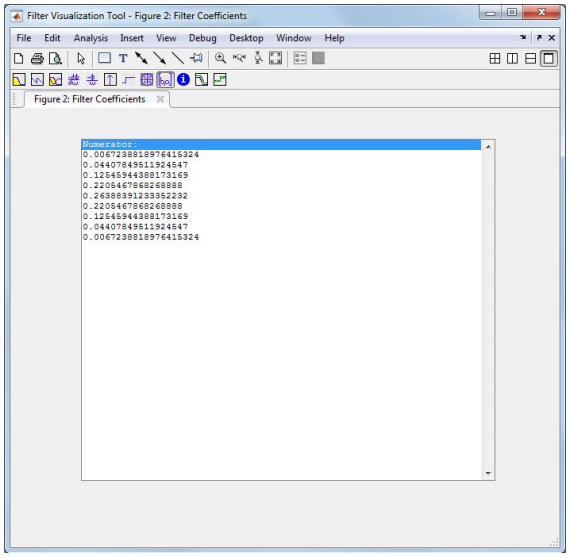


Figure 9: Definition of the lowpass filter characteristics in "filterbuilder"

The filter parameters like the 6dB-cutoff frequency, the passband ripple, stopband attenuation and the filter order – the latter one was chosen to 8 for a small implementation footprint – have a high influence on the feasibility and the actual achievable filter characteristics. Therefore it is important to further verify the filter performance with regard to the specification. As a first verification step, the filter magnitude (Figure 10a) and phase responses and other characteristics can be analyzed graphically as shown below. The 6 dB attenuation is achieved at 0.25 times the normalized Nyquist frequency and the overall attenuation in the stop band region is 60 dB. The filter order is eight and the coefficients are symmetrical as specified.





a) Magnitude Response of lowpass filter **b) Symmetrical filter coefficients of lowpass filter**

Figure 10: Analysis of the lowpass filter characteristics

3.2.2 Verification through Signal Flow Graph based Simulation

The magnitude response of the lowpass filter (see Figure 10a) respects the specification given at the beginning of this chapter. Especially because of the non-flat characteristic within the passband region which can cause distortion in the overall signal processing path it might make sense to further analyze the filter's performance by integrating it into a time-based simulation. The filter can be exported from the MATLAB-based context of the "filterbuilder" tool to a Simulink model using either specific filter blocks or basic Simulink blocks. The Simulink model can be rather simple like a unit-level test bench as shown in Figure 11 or a complex system e.g. containing the complete signal processing path of the ASIC similar to the one already shown in Figure 2.

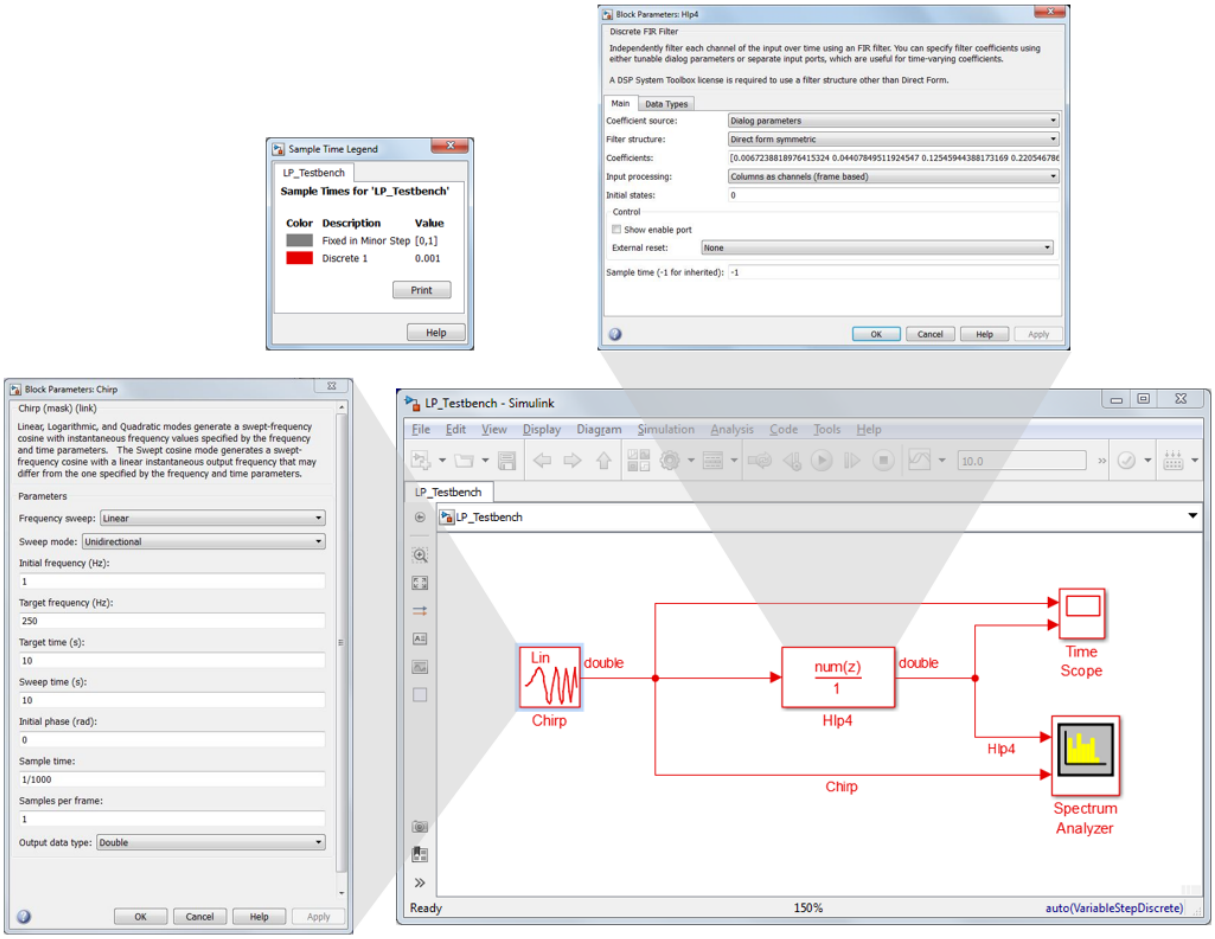


Figure 11: Simulink unit-level test bench for the lowpass filter

The lowpass filter represented by the digital FIR filter block "Hlp4" is stimulated by a linear chirp signal with an amplitude of one, a starting frequency of 1 Hz and a target frequency of 250 Hz. The sinusoidal input signal with steadily increasing frequency is sampled with a sample frequency of 1 kHz. Given these values and the filter specification (see Figure 9), the 6dB-cutoff frequency is expected to be located at 125 Hz.

As a first verification step we will have a look on the input and filtered output signals in the time domain (see Figure 12). The Time Scope shows the superposed amplitudes of the stimulating input signal (yellow in the background) and the output signal (blue in the foreground) of the digital filter in the course of time. The 6dB-cutoff frequency defines the frequency when the amplitude of the

output signal is attenuated to half of the input signal's amplitude. The chirp signal at the input starts with a frequency of 1 Hz and reaches 250 Hz after 10 seconds simulation time (see Figure 11, "Block Parameters: Chirp"). The 6dB-cutoff frequency is therefore reached after 5 seconds simulation time. The amplitude of the filtered output signal is half of the amplitude of the input signal at that moment in time as expected and as shown at measurement marker 2 in Figure 12.

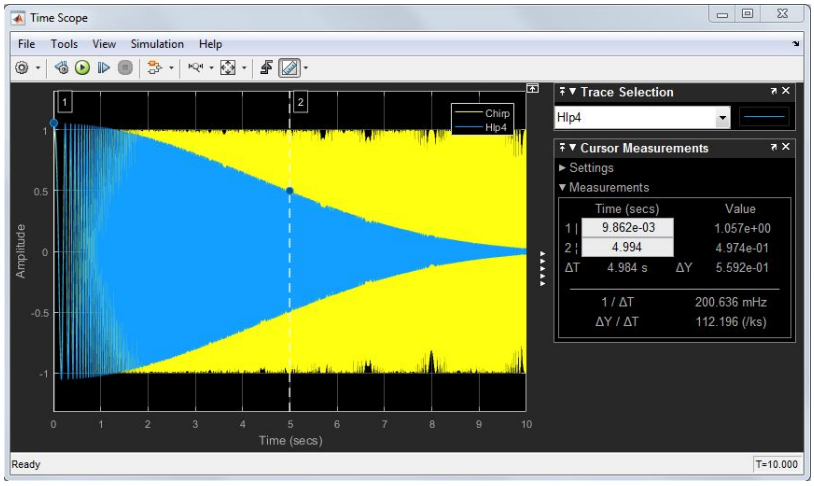
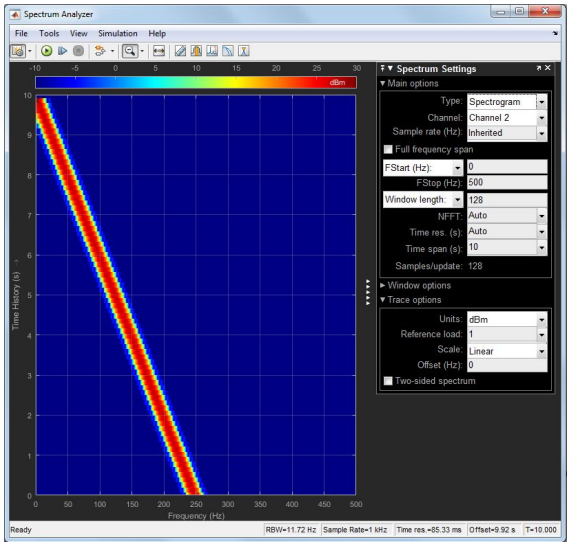
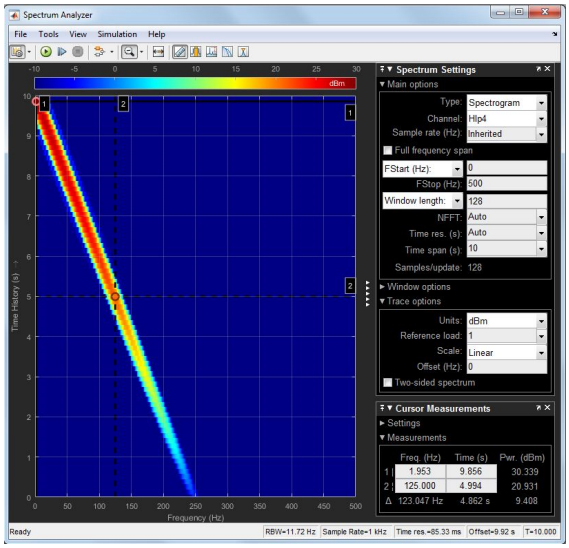


Figure 12: Time domain analysis of the lowpass filter

Additionally, we also analyze the signals in the frequency domain using the Spectrum Analyzer block. The frequency spectrum of the input chirp signal and the filtered output signal over the course of time are shown Figure 13 using a spectrogram. While the spectral power of the input chirp signal stays constant for all frequencies (Figure 13a), it decreases at the output of the lowpass filter as expected (Figure 13b) following its magnitude response for higher frequencies (see Figure 10a).



a) **Spectrogram of the input chirp signal**



b) **Spectrogram of the filtered output signal**

Figure 13: Frequency domain analysis of the lowpass filter

As a result of this design step we got the signal processing algorithm which meets the functional requirements and which is verified in time and frequency domain using a signal flow graph based simulation.

3.3 Model Generation

After the algorithm was designed and verified within the signal flow graph based Simulink simulation environment, the next step within the extended MBD workflow is the "Model Generation" (see Figure 7). In order to avoid the manual description of the component's behavior and its integration into an existing SystemC TLM 2.0 wrapper (see chapter 2. and therefore to automate these tasks, Simulink Coder and Embedded Coder also directly support the generation of all of these three parts. The automatic generation of this complete SystemC TLM 2.0 compliant models containing not only the SystemC TLM compliant wrapper but also its functional behavior is shown in Figure 14 in the context of our overall design workflow [26].

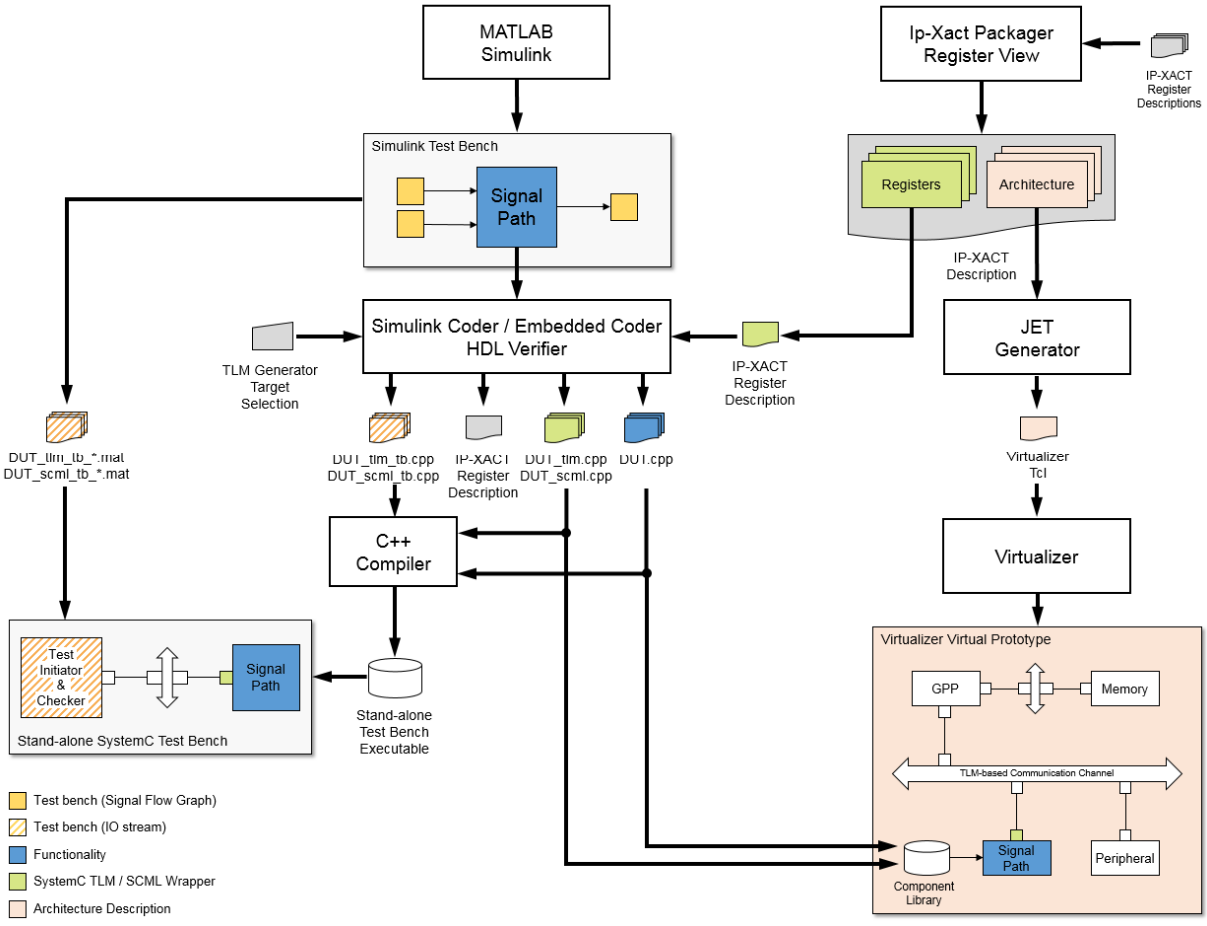


Figure 14: SystemC TLM 2.0 Model Generation Workflow

In the following of this chapter we will illustrate this workflow in detail reusing the lowpass filter example of the previous step "Algorithm Design" and integrating it into the following Simulink model instantiating two symmetrical FIR lowpass filters with external parameters (Figure 15).

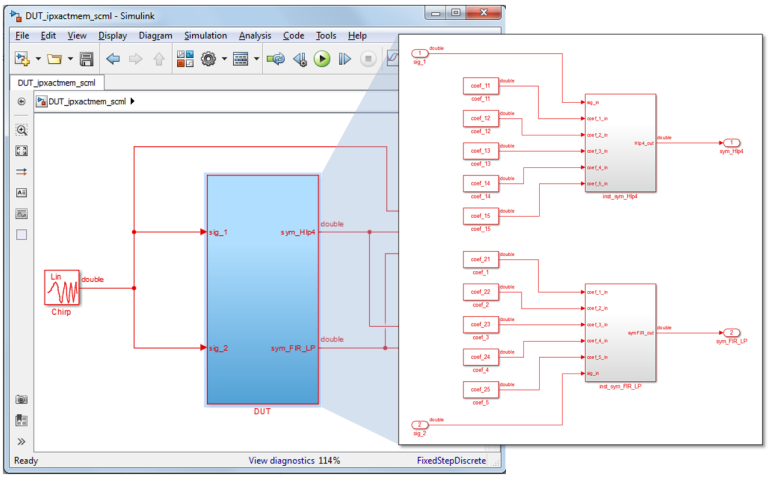


Figure 15: Simulink reference model with two symmetrical, parametrizable FIR lowpass filters

3.3.1 Automatic Generation of the SystemC TLM 2.0 compliant Component Model

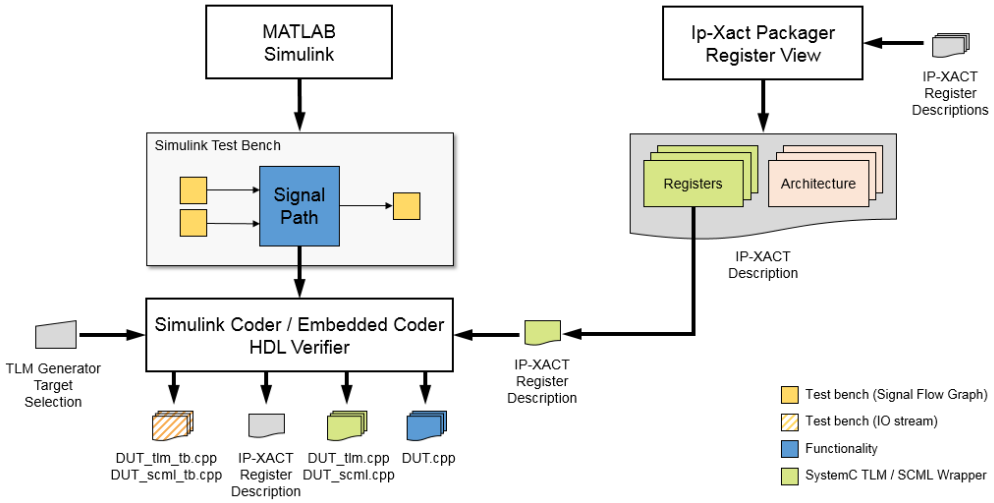


Figure 16: Automatic SystemC TLM 2.0 code generation

The Target Language Compiler (TLC) functionality of Simulink Coder and Embedded Coder provides a great deal of freedom and flexibility to customize the automatically generated ANSI C or C++ code to specific requirements [27]. The customized C or C++ code is primarily intended to e.g. target a particular embedded processor device taking advantage of their extended command sets to get access on peripheral units. Besides the target device specific customization, the TLC functionality also provides the possibility to automatically generate additional C/C++ code to enable and facilitate the integration of the functional C/C++ code in a specific environment.

Setup

HDL Verifier defines such specific target support packages for SystemC TLM 2.0 [26] and SystemVerilog DPI-C [28] targeting VPs. They are selected in the “Configuration Parameters” of the Simulink model (see Figure 17).

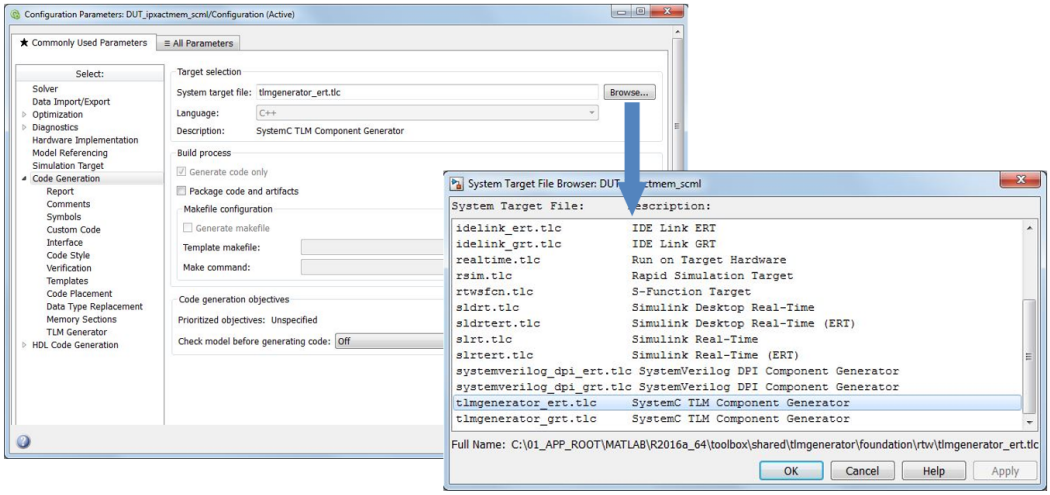


Figure 17: SystemC TLM Target Support Package Selection

The Simulink subsystem for which the SystemC TLM 2.0 component should be generated contains two representations of lowpass filters sharing the same signal processing characteristics. The filter parameters are provided externally to the filter using constant blocks. The numeric values of the constant blocks are defined in the MATLAB workspace.

When these parameters are used in the block parameters of e.g. the constant block, their respective numerical values will by default be “inlined” in the automatically generated C++ code, i.e. these values are hardcoded in the C++ code and cannot be changed anymore during runtime. In order to get the possibility to change these values during runtime of the Simulink simulation and in the C++ code it is possible to declare them as “tunable” in the model’s “Configuration Parameters” as shown in Figure 18 [29]. In this example we only declare the coefficients “coef_11”, “coef_12”, “coef_13”, “coef_14” and “coef_11” of the first symmetrical FIR lowpass filter as “tunable” to highlight the differences in the automatically generated C++ code.

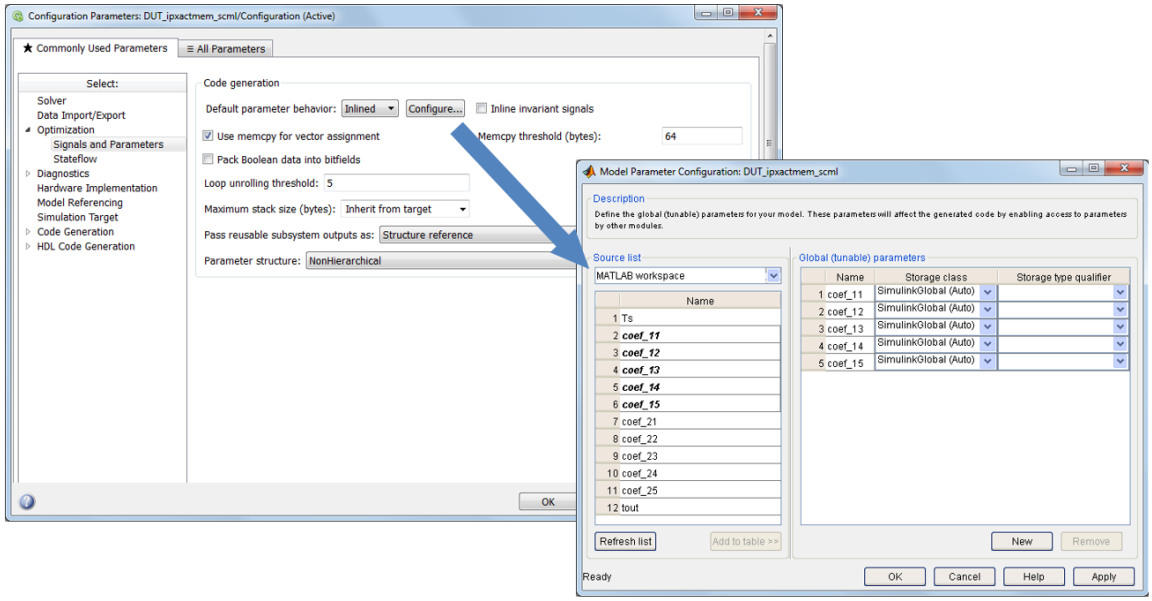


Figure 18: Definition of tunable parameters

As already introduced in Figure 3 and Figure 4 we are using an IP-XACT centric workflow. The IP-XACT description serves as reference for several derived documents, automatically generated files, e.g. SystemC TLM wrappers through Virtualizer TLM Creator, and JET-based scripts for the generation of the Virtualizer architecture. Besides the architecture description, it contains the definition of the memory maps of the component, which contain the input, output and parameter registers. Thereby, the IP-XACT description can contain several memory maps. For every memory map, a separate TLM target socket will be generated. The mapping between inputs, outputs and parameters of the Simulink model and registers is defined by the “MWMapInput”, “MWMapOutput” and “MWMapParam” parameter elements. Figure 19 and Figure 20 show the input registers of the signal path’s sensor interface in Magillem m Register View (MRV) [18] and the corresponding input register definition in IP-XACT in its base address space. Figure 21 and Figure 22 show the IP-XACT description of the output and parameter registers of the bus interface.

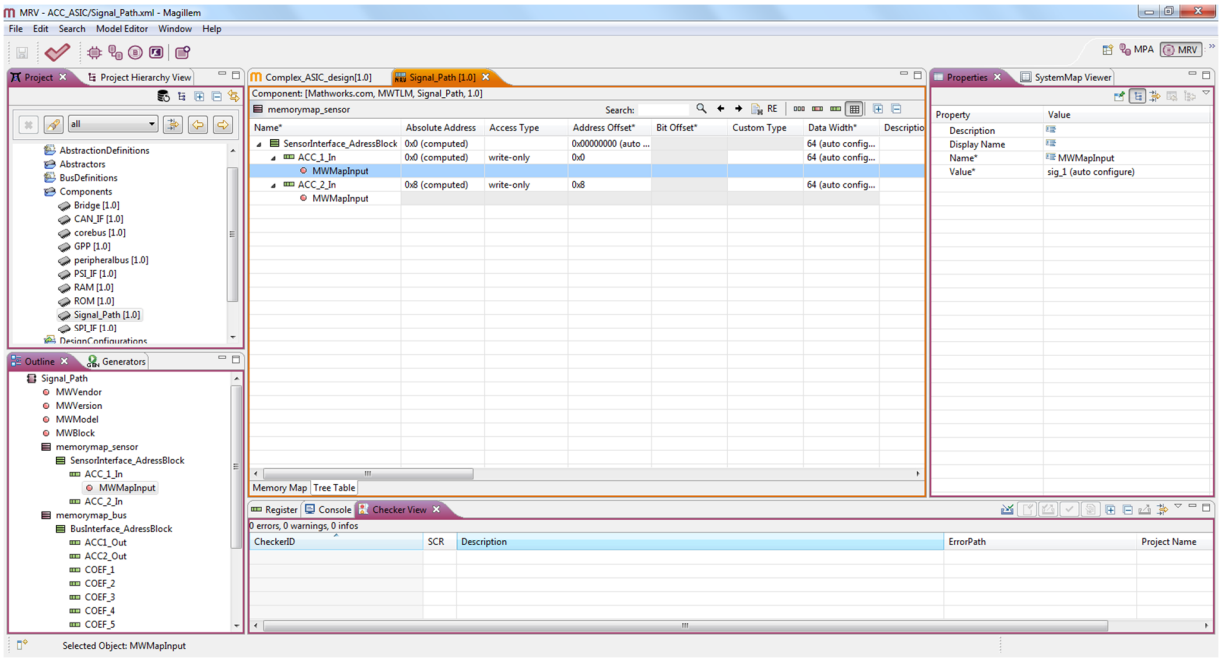


Figure 19: Register interface of signal path component in Magillem Register View (MRV)

```
46      <spirit:memoryMaps>
47          <spirit:memoryMap>
48              <spirit:name>memorymap_sensor</spirit:name>
49              <spirit:addressBlock>
50                  <spirit:name>SensorInterface_AdressBlock</spirit:name>
51                  <spirit:baseAddress spirit:id="base_address_input" spirit:resolve="user">0x00000000</spirit:baseAddress>
52                  <spirit:range>16</spirit:range>
53                  <spirit:width>64</spirit:width>
54                  <spirit:usage>register</spirit:usage>
55                  <spirit:register>
56                      <spirit:name>ACC_1_In</spirit:name>
57                      <spirit:addressOffset>0x00</spirit:addressOffset>
58                      <spirit:size>64</spirit:size>
59                      <spirit:access>write-only</spirit:access>
60                      <spirit:reset>
61                          <spirit:value>0x00</spirit:value>
62                      </spirit:reset>
63                      <spirit:parameters>
64                          <spirit:parameter>
65                              <spirit:name>MWMapInput</spirit:name>
66                              <spirit:value>sig_1</spirit:value>
67                          </spirit:parameter>
68                      </spirit:parameters>
69                  </spirit:register>
```

Figure 20: IP-XACT memory map definition "memorymap_sensor" (ex.: input register ACC_1_in mapping to Simulink input signal "sig_1")

```
87          <spirit:memoryMap>
88              <spirit:name>memorymap_bus</spirit:name>
89              <spirit:addressBlock>
90                  <spirit:name>BusInterface_AdressBlock</spirit:name>
91                  <spirit:baseAddress spirit:id="subsystem_tlm_base_address_output" spirit:resolve="user">0x00010000</spirit:baseAddress>
92                  <spirit:range spirit:resolve="immediate">0x100</spirit:range>
93                  <spirit:width>64</spirit:width>
94                  <spirit:usage>register</spirit:usage>
95                  <spirit:register>
96                      <spirit:name>ACC1_Out</spirit:name>
97                      <spirit:addressOffset>0x00</spirit:addressOffset>
98                      <spirit:size>64</spirit:size>
99                      <spirit:access>read-only</spirit:access>
100                     <spirit:reset>
101                         <spirit:value>0x00</spirit:value>
102                     </spirit:reset>
103                     <spirit:parameters>
104                         <spirit:parameter>
105                             <spirit:name>MWMapOutput</spirit:name>
106                             <spirit:value>sym_Hlp4</spirit:value>
107                         </spirit:parameter>
108                     </spirit:parameters>
109                 </spirit:register>
```

Figure 21: IP-XACT memory map definition "memorymap_bus" (ex.: output register ACC1_Out mapping to Simulink output signal "sym_Hlp4")

```
125                 <spirit:register>
126                     <spirit:name>COEF_1</spirit:name>
127                     <spirit:addressOffset>0x10</spirit:addressOffset>
128                     <spirit:size>64</spirit:size>
129                     <spirit:access>read-write</spirit:access>
130                     <spirit:reset>
131                         <spirit:value>0x00</spirit:value>
132                     </spirit:reset>
133                     <spirit:parameters>
134                         <spirit:parameter>
135                             <spirit:name>MWMapParam</spirit:name>
136                             <spirit:value>coef_11</spirit:value>
137                         </spirit:parameter>
138                     </spirit:parameters>
139                 </spirit:register>
```

Figure 22: IP-XACT memory map definition "memorymap_bus" (ex.: parameter register COEF_1 mapping to Simulink parameter "coef_11")

The SystemC TLM 2.0 target support package of the Simulink Coder or Embedded Coder accepts this IP-XACT memory map definition as input. The path to the IP-XACT file is also defined in the model's "Configuration Parameters" (see Figure 23). The specification to implement the interface using SCML

compliant registers is done in the same menu.

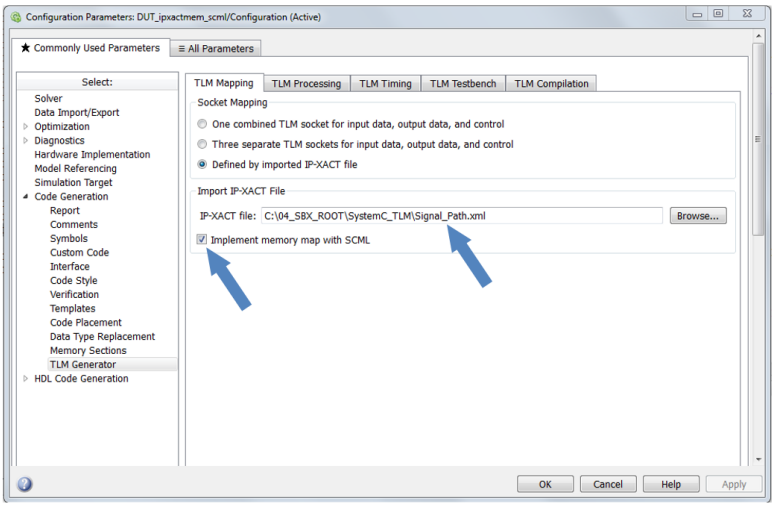


Figure 23: Definition of path to the IP-XACT description file

Component Generation through automatic Code Generation

The automatic SystemC TLM 2.0 component generation follows the classic automatic C/C++ code generation workflow. It is started by a right-mouse-click on the subsystem for which the component should be generated (here: “DUT”) and by selecting “C/C++ Code” and “Build This Subsystem” (see Figure 24). The tunable parameters of the subsystem can be reviewed and changed if necessary. As already specified in the model’s configuration parameters, only the coefficients of the upper symmetrical FIR lowpass filter are declared as “tunable” using “Simulink Global” as interface. The coefficients of the second symmetrical FIR lowpass filter are “inlined”, i.e. their values will be hardcoded in the functional C++ code of the component. The code generation process is finally started by selecting the “Build” button in the opened GUI and its status can be followed in the Diagnostics Viewer.

In the previous workflow (see chapter 2.), the SystemC TLM 2.0 wrapper was also automatically generated by the Virtualizer TLM Creator based on the IP-XACT component description. But both, the description of the component’s functionality and its integration into the SystemC TLM 2.0 wrapper had to be done manually. The new workflow automates now all three parts through automatic code generation based on a Simulink model:

- Functional description of the component
- SystemC TLM 2.0 interface wrapper using SCML registers
- Link between the SystemC wrapper and the functional description

In the following we will have a closer look on the automatically generated files.

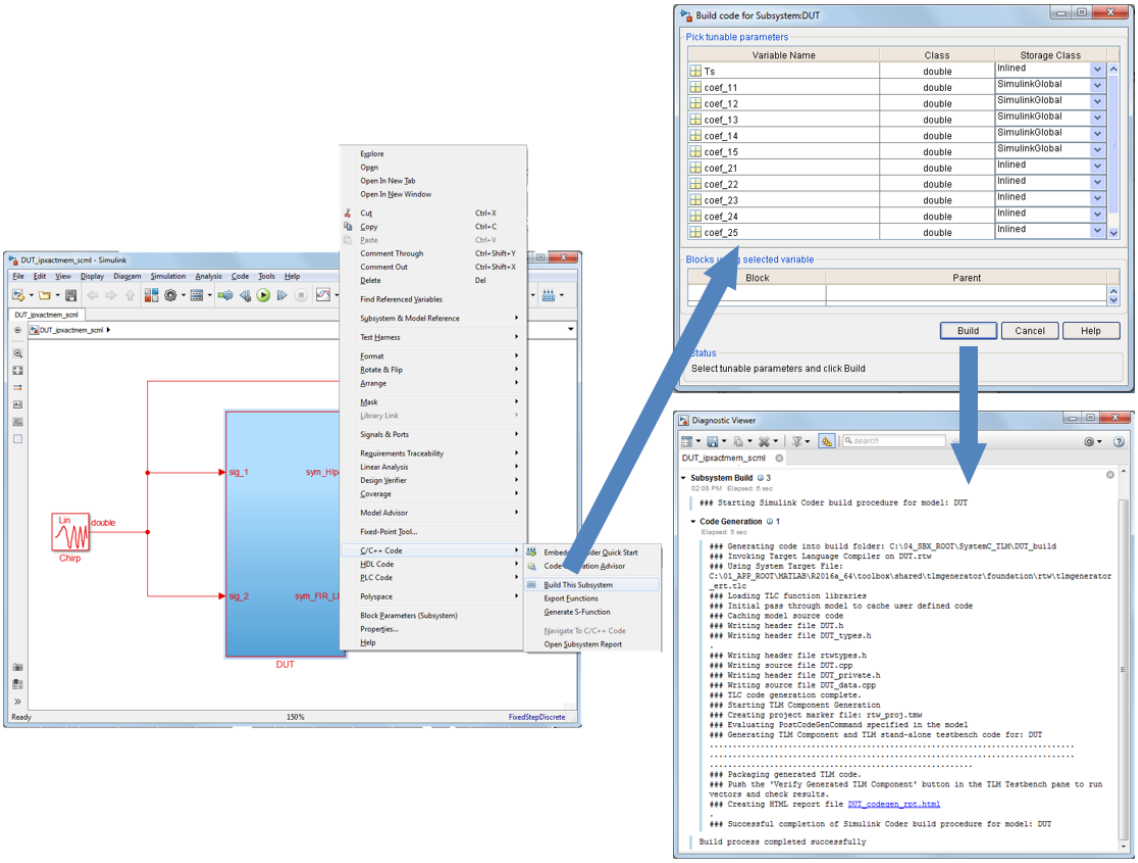


Figure 24: Launching the automatic SystemC TLM 2.0 component generation

Functional Description of the Component

The component's functional core is described using generic C++. It is the same as for the "Algorithm Implementation" path in the classical MBD workflow (see Figure 7). Three different functions are generated:

- Initialization function (e.g. "DUT_initialize")
  It sets all internal states of the functional description to the corresponding reset value. It is only called once at the very beginning of the simulation.
- Step function (e.g. "DUT_step")
  This function contains the functional description of the algorithm. It is repeatedly called to calculate the algorithm's outputs (see Figure 25).
- Termination function (e.g. "DUT_terminate")
  This function frees up all internal state variables at the end of the simulation. Also this function is only called once at the very end of the simulation.

```
30   // DiscreteFir: '<S3>/Hlp4_1' incorporates:
31   //   Constant: '<S1>/coef_11'
32   //   Constant: '<S1>/coef_12'
33   //   Constant: '<S1>/coef_13'
34   //   Constant: '<S1>/coef_14'
35   //   Constant: '<S1>/coef_15'
36   //   Inport: '<Root>/sig_1'
37   //   SignalConversion: '<S3>/ConcatBufferAtVector ConcatenateIn1'
38   //   SignalConversion: '<S3>/ConcatBufferAtVector ConcatenateIn2'
39   //   SignalConversion: '<S3>/ConcatBufferAtVector ConcatenateIn3'
40   //   SignalConversion: '<S3>/ConcatBufferAtVector ConcatenateIn4'
41
42   dataIdx = DUT_DW->Hlp4_1_circBuf;
43   if (DUT_DW->Hlp4_1_circBuf + 7 >= 8) {
44     dataIdx = DUT_DW->Hlp4_1_circBuf - 8;
45   }
46
47   rtb_UnitDelay1 = (DUT_DW->Hlp4_1_states[dataIdx + 7] + DUT_U->sig_1) *
48     DUT_P->coef_11;
49   dataIdx = DUT_DW->Hlp4_1_circBuf;
50   if (DUT_DW->Hlp4_1_circBuf >= 8) {
51     dataIdx = DUT_DW->Hlp4_1_circBuf - 8;
52   }
53
```

Figure 25: Functional description of Hlp4 filter (src: DUT.cpp, function "DUT_step")

SystemC TLM 2.0 Interface Wrapper

The memory map information about the component interface, which was provided to Simulink Coder or Embedded Coder for the automatic code generation process by the IP-XACT component description, is taken into account in a specific definition header file of the SystemC TLM 2.0 wrapper (e.g. DUT_scml_def.h).

```
20  //========= SensorInterface SOCKET =========
21  #define SensorInterface_BUSWIDTH       64
22
23  //========= SensorInterface_AdressBlock BANK =========
24  //Bank base address
25  #define SensorInterface_AdressBlock_ADDR 0x00000000
26
27  //========= REGISTERS =========
28  //Address of the registers
29  #define ACC_1_In_ADDR                  0x00000000
30  #define ACC_2_In_ADDR                  0x00000008
31
32  //========= DATA =========
33  #define ACC_1_In_DIM                   1
34  #define ACC_2_In_DIM                   1
35
36  typedef real_T ACC_1_In_T;
37  typedef real_T ACC_2_In_T;
```

Figure 26: Memory map "SensorInterface" socket of SystemC TLM wrapper (src: DUT_scml_def.h)

```
39  //========= BusInterface SOCKET =========
40  #define BusInterface_BUSWIDTH          64
41
42  //========= BusInterface_AdressBlock BANK =========
43  //Bank base address
44  #define BusInterface_AdressBlock_ADDR  0x00010000
45
46  //========= REGISTERS =========
47  //Address of the registers
48  #define ACC1_Out_ADDR                  0x00000000
49  #define ACC2_Out_ADDR                  0x00000008
50  #define COEF_1_ADDR                    0x00000010
51  #define COEF_2_ADDR                    0x00000018
52  #define COEF_3_ADDR                    0x00000020
53  #define COEF_4_ADDR                    0x00000028
54  #define COEF_5_ADDR                    0x00000030
55
56  //========= DATA =========
57  #define ACC1_Out_DIM                   1
58  #define ACC2_Out_DIM                   1
59  #define COEF_1_DIM                     1
60  #define COEF_2_DIM                     1
61  #define COEF_3_DIM                     1
62  #define COEF_4_DIM                     1
63  #define COEF_5_DIM                     1
64
65  typedef real_T ACC1_Out_T;
66  typedef real_T ACC2_Out_T;
67  typedef real_T COEF_1_T;
68  typedef real_T COEF_2_T;
69  typedef real_T COEF_3_T;
70  typedef real_T COEF_4_T;
71  typedef real_T COEF_5_T;
```

Figure 27: Memory map "BusInterface" socket of SystemC TLM wrapper (src: DUT_scml_def.h)

The interface and parameter registers themselves are implemented in the SystemC TLM 2.0 wrapper using SCML registers (see Figure 28). These provide backdoor access for debugging purposes to the Virtualizer virtual prototyping environment. Using SCML registers, it is then possible for the hardware or software developer to view register and bitfield contents in the GUI of the debugging environment Synopsys VPExplorer. Furthermore, it is possible to change register values during simulation through the GUI as well as through the VPExplorer Tcl interface e.g. for error injection purposes.

```
10  using namespace std;
11  DUT_scml::DUT_scml( sc_core::sc_module_name module_name, eTimingType
12                      DefaultTiming)
13    : sc_module(module_name),
14    SensorInterface("SensorInterface"),
15    SensorInterface_adapter("SensorInterface_adapter",SensorInterface),
16    SensorInterface_router("SensorInterface_router",16/sizeof(uint8_T[8])),
17    SensorInterface_AdressBlock("SensorInterface_AdressBlock",16/sizeof(uint8_T[8])),
18    ACC_1_In("ACC_1_In",SensorInterface_AdressBlock,ACC_1_In_ADDR/sizeof(uint8_T[8])),
19    ACC_2_In("ACC_2_In",SensorInterface_AdressBlock,ACC_2_In_ADDR/sizeof(uint8_T[8])),
20    BusInterface("BusInterface"),
21    BusInterface_adapter("BusInterface_adapter",BusInterface),
22    BusInterface_router("BusInterface_router",65792/sizeof(uint8_T[8])),
23    BusInterface_AdressBlock("BusInterface_AdressBlock",256/sizeof(uint8_T[8])),
24    ACC1_Out("ACC1_Out",BusInterface_AdressBlock,ACC1_Out_ADDR/sizeof(uint8_T[8])),
25    ACC2_Out("ACC2_Out",BusInterface_AdressBlock,ACC2_Out_ADDR/sizeof(uint8_T[8])),
26    COEF_1("COEF_1",BusInterface_AdressBlock,COEF_1_ADDR/sizeof(uint8_T[8])),
27    COEF_2("COEF_2",BusInterface_AdressBlock,COEF_2_ADDR/sizeof(uint8_T[8])),
28    COEF_3("COEF_3",BusInterface_AdressBlock,COEF_3_ADDR/sizeof(uint8_T[8])),
29    COEF_4("COEF_4",BusInterface_AdressBlock,COEF_4_ADDR/sizeof(uint8_T[8])),
30    COEF_5("COEF_5",BusInterface_AdressBlock,COEF_5_ADDR/sizeof(uint8_T[8])),
31    m_input_refresh_reg(2),
32    m_start_event("start_event"),
33    m_current_timing(DefaultTiming)
34  {
35    SensorInterface_adapter(SensorInterface_router);
36    SensorInterface_router.map(0x00000000,16,SensorInterface_AdressBlock,0x00);
37    scml2::set_write_callback(ACC_1_In, SCML2_CALLBACK(ACC_1_In_write_cb), scml2::
38      AUTO_SYNCING);
39    scml2::set_write_callback(ACC_2_In, SCML2_CALLBACK(ACC_2_In_write_cb), scml2::
40      AUTO_SYNCING);
41    BusInterface_adapter(BusInterface_router);
42    BusInterface_router.map(0x00010000,256,BusInterface_AdressBlock,0x00);
43    scml2::set_read_callback(ACC1_Out, SCML2_CALLBACK(ACC1_Out_read_cb), scml2::
44      AUTO_SYNCING);
```

Figure 28: Interface of SystemC TLM 2.0 wrapper with SCML registers (src: DUT_scml.cpp)

Link between the SystemC Wrapper and the Functional Description

The link between the functional description of the component's core and the SystemC TLM interface is dependent on the selections made in the model's "Configuration Parameters". It can be based either on SystemC threads or on callbacks [30]. When using SCML registers for the interface memory map the link will be automatically using a SystemC thread, e.g. "targetThread".

Its principle processing flow (see Figure 29) follows the classical structure of a SystemC thread process. It is kept "alive" by an infinite "while" loop and its execution is controlled by a "wait" statement depending on the "default_event".

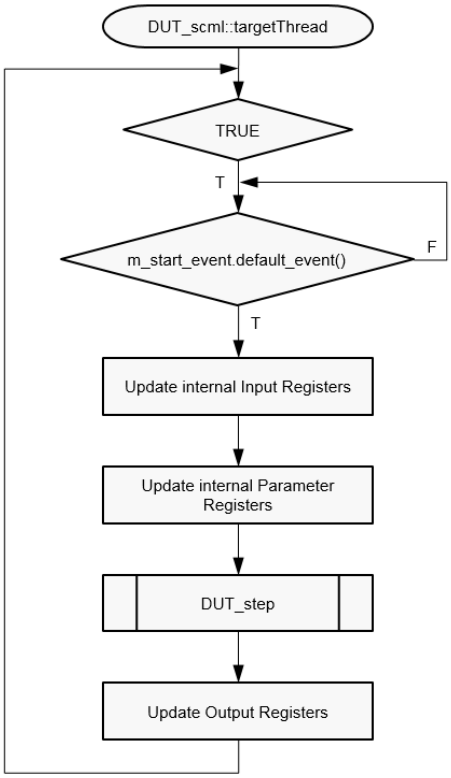


Figure 29: Flow diagram DUT_scml::targetThread

In order to guarantee the same deterministic simulation behavior as in the Simulink reference model of the "Algorithm Design" step (see Figure 7), all input registers need to be updated before their values are passed to the C++ description of the component's functionality (here: "DUT_step"). The SCML output registers of the SystemC TLM 2.0 wrapper are updated after the functional model provides the new values.

3.3.2 Stand-alone SystemC Verification on Component Level

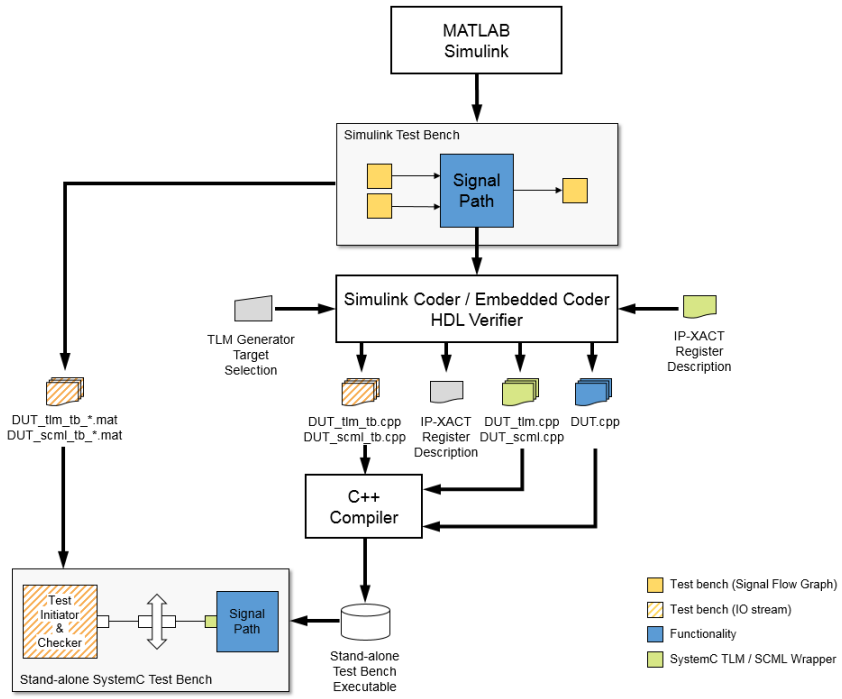


Figure 30: Stand-alone SystemC Verification

Together with the functional description of the component and its SystemC TLM 2.0 compliant wrapper, a stand-alone SystemC test bench and associated makefiles for their compilation and execution are also automatically generated. The purpose of this stand-alone test bench is to verify the functional equivalence between the Simulink reference model and the SystemC component model. The self-testing test bench only contains a Test Initiator and Checker and an instance of DUT.

The execution of this test bench is launched within the model's configuration parameters as shown in Figure 31.

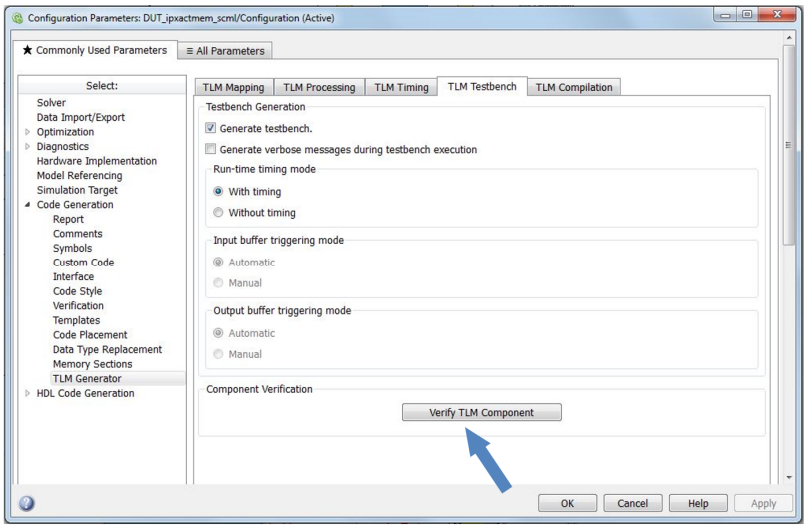


Figure 31: Launching stand-alone SystemC component test bench

As a first step the Simulink simulation is run in order to capture the signal streams at the inputs and outputs of the DUT. Then, these are stored as input stimuli and as expected results at the outputs, respectively. After that, the source files of the SystemC test bench and component wrapper as well as the C++ functional description of the component's behavior are compiled. The SystemC test bench is finally executed within the Open SystemC Initiative (OSCI) SystemC Reference Simulator.

3.4 Integration into the Virtual Prototype and Simulation

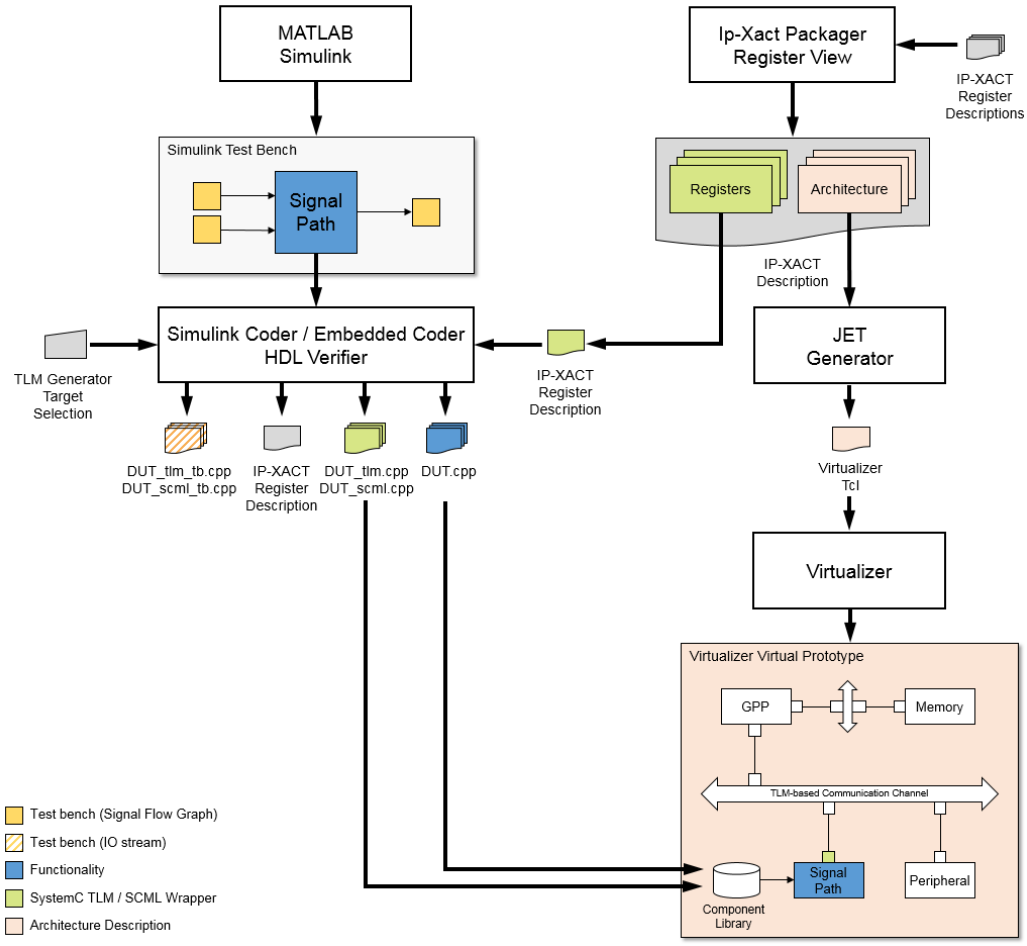


Figure 32: SystemC TLM component integration into Virtualizer VP

The automatically generated SystemC TLM 2.0 wrapper and C++ functional description of the component's behavior can now be integrated into the Virtualizer VP

The first step for the integration of the signal processing component into the Virtualizer VP is the import of the generated SystemC module into Virtualizer. For this, the cpp file of the generated SCML module as well as the folders containing the required header files are passed to Virtualizer using the "Import SystemC Modules" functionality (see Figure 33).

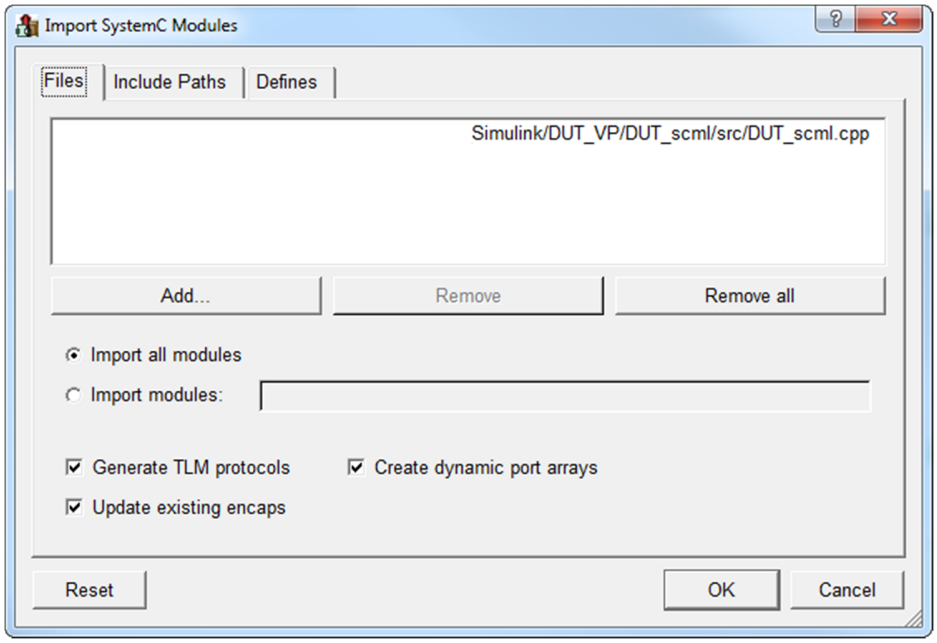


Figure 33: "Import SystemC Modules" dialogue in Virtualizer

After that, the further generated cpp files containing the functional core as well and the location of the MATLAB shared libraries have to be added to the Encap Configuration of the imported SystemC module (see Figure 34 and Figure 35).

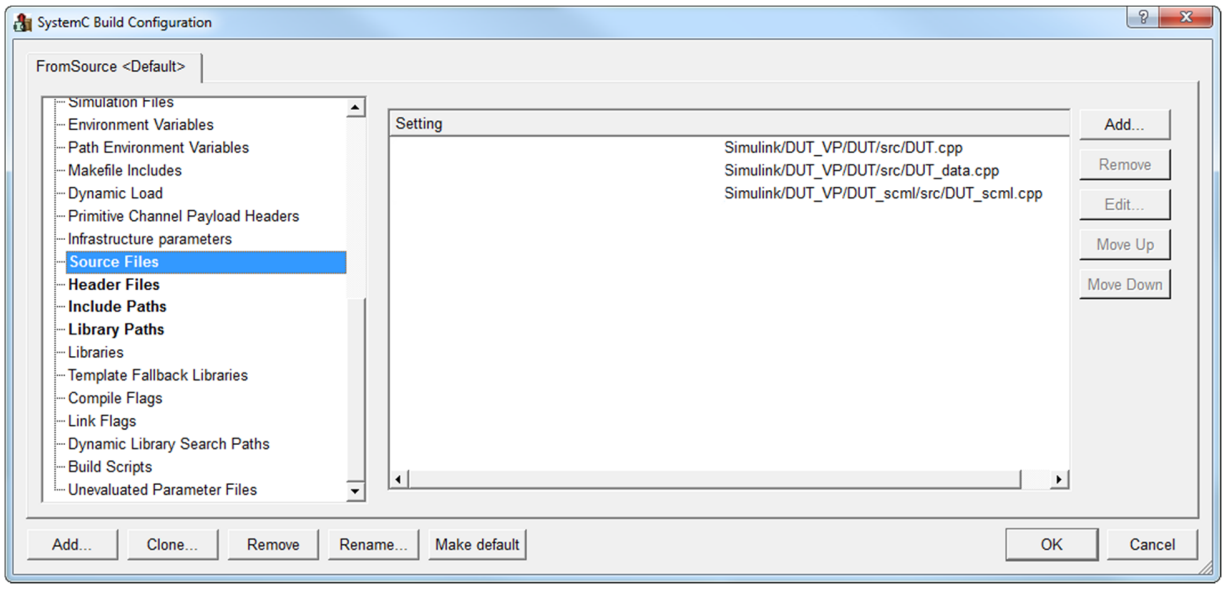


Figure 34: Encap Configuration of imported SystemC module in Virtualizer

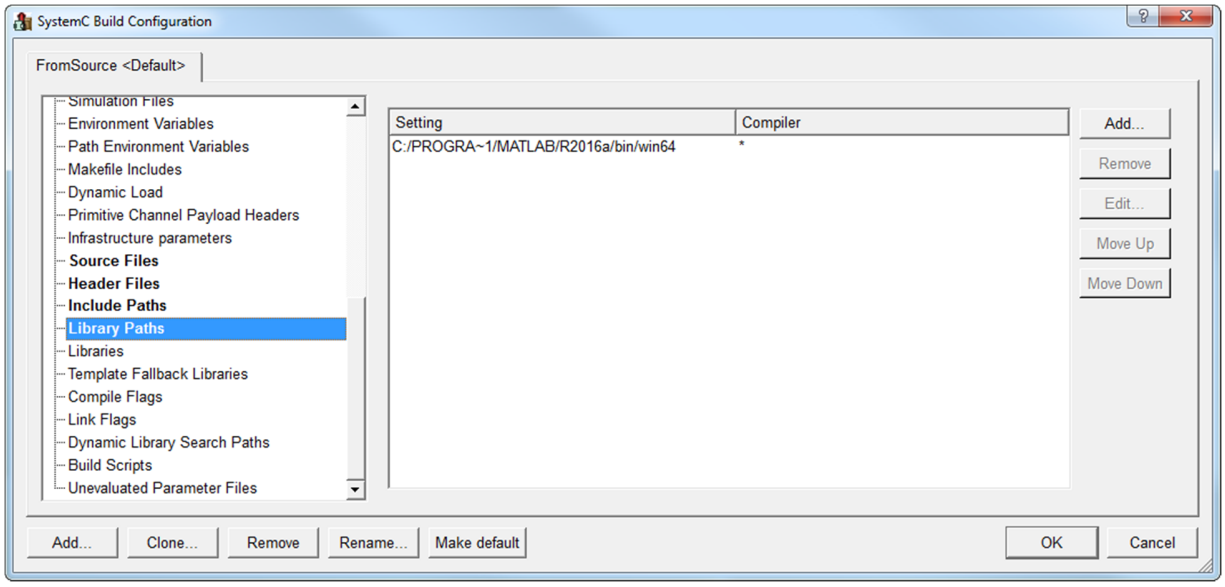


Figure 35: Library path in Encap Configuration

Finally, the imported signal path component can be connected directly to the communication architecture in the virtual platform. This can be done either manually or by using the Java® Emitter Template (JET) based workflow (see Figure 4) which has already been mentioned in chapter 2. This workflow automatically instantiates the generated module and generates the whole communication architecture. Figure 36 shows the Virtualizer architecture including the generated signal path component in Synopsys Platform Creator. We can see that the sensor interface is connected to an ACC sensor model and the bus interface is connected to the on-chip bus architecture. Due to the automatically generated SCML register interface and the connection between registers and in- and outputs of the signal processing core, no additional manual integration tasks are needed.

Now, the register interface is known and visible in the debugging environment. In Figure 37, we can see the debug view during the simulation of the VP in VPExplorer. On the left side, the internals of the on-chip modules including interface registers can be found. On the right side, waveforms of signals connected to the in- and output registers are depicted. In this simulation scenario, chirp signals were generated in the ACC sensor module and transferred to the signal path module. This shows the successful integration of the signal processing behavior modeled Simulink into the Virtualizer VP.

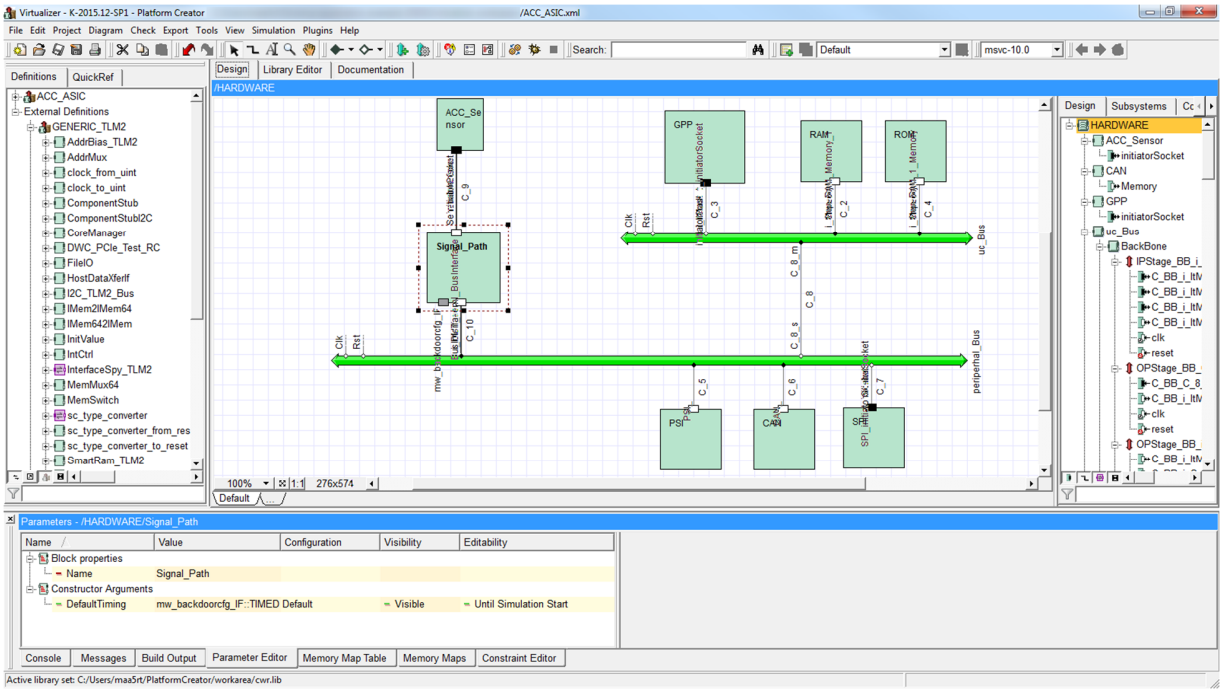


Figure 36: Architecture including generated signal path component in Virtualizer

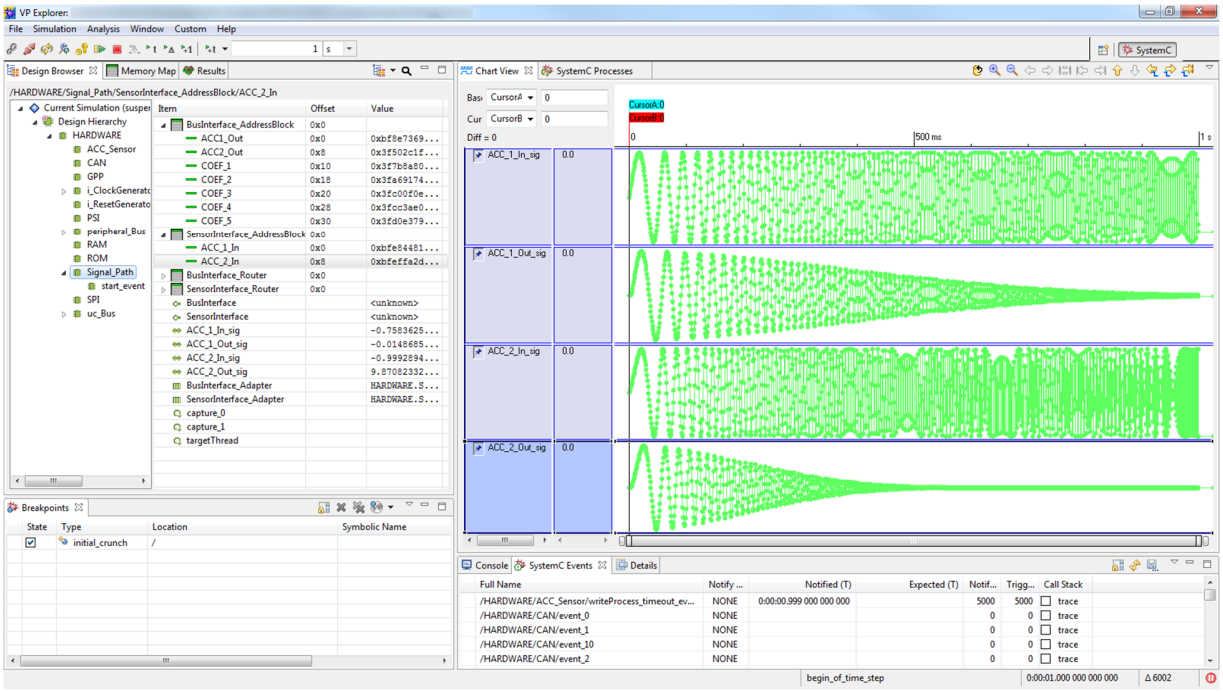


Figure 37: Debug view of VP during simulation in VPExplorer

4. Conclusions

In order to accelerate the construction of Virtual Prototypes representing signal processing ASICs, we developed a new method for the automated integration of signal processing behavior modules in Simulink into VPs. For this, we presented an improvement to our IP-XACT-centric workflow for the automatic generation of signal processing functionality encapsulated by SystemC TLM 2.0 wrappers which contain SCML registers. In this approach, a functional C/C++ description is automatically generated from a Simulink model. A SystemC module is also generated which contains an SCML register interface corresponding to the input IP-XACT description. Furthermore, the link between the functional description of the component's core and the SCML register interface is generated based on the mapping also defined in the IP-XACT description. These automatically generated components can directly be integrated into Virtualizer architectures, in which the components can be simulated and extensively be debugged on SoC level. For this solution we extended IP-XACT descriptions which are used by MathWorks code generation tools containing information about the mapping between bus slave interface registers and in- and outputs of Simulink models.

The presented approach enables an efficient way for integrating the signal path behavior into our VPs. In combination with our already existing approaches for generating TLM 2.0 module wrappers based on register descriptions for arbitrary, not signal processing centric models using Synopsys TLM Creator and for generating Virtualizer architectures using our own JET generator, we have a complete and very fast solution to create an initial functional VP of our signal processing ASICs based on Simulink models of our signal path components and IP-XACT register and design descriptions.

We also showed the application of the complete workflow of creating the behavior of a signal processing component using MBD and integrating the behavior into a Virtualizer architecture using the industrial example of an acceleration sensor ASIC.

Acknowledgement: This work has been funded by the German Federal Ministry for Education and Research (Bundesministerium für Bildung und Forschung, BMBF) under the grant 01IS13022 (project EffektiV). The content of this publication lies within the responsibility of the authors.

5. References

[1] V. Krishnamurthy, A. Mauderer and J.-H. Oetjens, "Automated Generation of Synopsys Virtualizer Architectures based on Hardware Descriptions in IP-XACT," in *Synopsys User Group Germany*, 2015.

[2] F. Kesel, Modellierung von digitalen Systemen mit SystemC: Von der Rtl zur TransactionLevelModellierung, Oldenbourg Wissenschaftsverlag, 2012.

[3] The MathWorks Inc., "Model-Based Design," 2016. [Online]. Available: http://de.mathworks.com/solutions/model-based-design/. [Accessed 12 May 2016].

[4] IEEE, "1685 – IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating and Reusing IP within Tool Flows," 2010.

[5] Eclipse Foundation Inc., "Java Emitter Templates Website," [Online]. Available: http://projects.eclipse.org/projects/modeling.m2t.jet.

[6] "IEEE Standard SystemC® Language Reference Manual," 2005.

[7] O. S. Initiative, "TLM-2.0 User Manual".

[8] Synopsys, Inc., "SCML Website," [Online]. Available: https://www.synopsys.com/cgi-bin/slcw/kits/reg.cgi.

[9] The MathWorks Inc., "MATLAB," 2016. [Online]. Available: http://www.mathworks.com/products/matlab/. [Accessed 12 May 2016].

[10] The MathWorks Inc., "Simulink," 2016. [Online]. Available: http://www.mathworks.com/products/simulink/. [Accessed 12 May 2016].

[11] The MathWorks Inc., "Simulink Coder," 2016. [Online]. Available: http://www.mathworks.com/products/simulink-coder/. [Accessed 12 May 2016].

[12] The MathWorks Inc., "Embedded Coder," 2016. [Online]. Available: http://www.mathworks.com/products/embedded-coder/. [Accessed 12 May 2016].

[13] The MathWorks Inc., "HDL Coder," 2016. [Online]. Available: http://www.mathworks.com/products/hdl-coder/. [Accessed 12 May 2016].

[14] The MathWorks Inc., "Stateflow," 2016. [Online]. Available: http://www.mathworks.com/products/stateflow/. [Accessed 12 May 2016].

[15] A. Mauderer, J. H. Oetjens and W. Rosenstiel, "Bridging the Gap between Simulink and Analog Design Environments Using HDL Code Generation," in *GMM-Fachbericht-ANALOG'11*, 2011.

[16] Magillem S.A., "Magillem Website," [Online]. Available: http://www.magillem.com/.

[17] Magillem S.A., "Magillem m IP-XACT Packager (MIP)," October 2013. [Online]. Available: http://www.magillem.com/wp-content/uploads/2015/02/Magillem-IP-XACT-Packager_1-8.pdf. [Accessed 12 May 2016].

[18] Magillem S.A., "Magillem m Register View (MRV)," May 2014. [Online]. Available: http://www.magillem.com/wp-content/uploads/2016/01/Magillem-Register-View_2.2.pdf. [Accessed 12 May 2016].

[19] The MathWorks Inc., "HDL Verifier," 2016. [Online]. Available: http://www.mathworks.com/products/hdl-verifier/. [Accessed 12 May 2016].

[20] IEEE Standards Association, "SystemVerilog Language Working Group," 2016. [Online]. Available: http://standards.ieee.org/develop/wg/1800_WG.html. [Accessed 12 May 2016].

[21] Accellera Systems Initiative Inc., "Standard Universal Verification Methodology (UVM)," 2016. [Online]. Available: http://accellera.org/downloads/standards/uvm. [Accessed 12 May 2016].

[22] The MathWorks Inc., "UVM Verification," 2016. [Online]. Available: http://www.mathworks.com/discovery/uvm-verification.html. [Accessed 12 May 2016].

[23] H. Thibiéroz, "Optimized Synopsys-MathWorks solution for System-Level Verification," 14 November 2013. [Online]. Available: https://blogs.synopsys.com/analoginsights/2013/11/14/optimized-synopsys-mathworks-solution-for-system-level-verification/. [Accessed 12 May 2016].

[24] The MathWorks Inc., "Documentation "filterbuilder"," 2016. [Online]. Available: http://www.mathworks.com/help/releases/R2016a/dsp/ref/filterbuilder.html. [Accessed 12 May 2016].

[25] The MathWorks Inc., "DSP System Toolbox," 2016, [Online]. Available: http://www.mathworks.com/products/dsp-system/. [Accessed 12 May 2016].

[26] The MathWorks Inc., "Select TLM Generator System Target," 2016. [Online]. Available: http://www.mathworks.com/help/releases/R2016a/hdlverifier/ug/select-tlm-generator-system-target.html. [Accessed 12 May 2016].

[27] The MathWorks Inc., "Target Language Compiler," 2016. [Online]. Available: http://www.mathworks.com/help/releases/R2016a/rtw/block-authoring-with-tlc.html. [Accessed 12 May 2016].

[28] The MathWorks Inc., "Generate SystemVerilog DPI Component," 2016. [Online]. Available: http://www.mathworks.com/help/releases/R2016a/hdlverifier/ug/generate-systemverilog-dpi-component-1.html. [Accessed 12 May 2016].

[29] The MathWorks Inc., "Control Parameter Representation and Declare Tunable Parameters in the Generated Code," 2016. [Online]. Available: http://www.mathworks.com/help/releases/R2016a/ecoder/ug/use-parameter-objects-for-code-generation.html. [Accessed 12 May 2016].

[30] The MathWorks Inc., "TLM Component Architecture," 2016. [Online]. Available: http://www.mathworks.com/help/releases/R2016a/hdlverifier/ug/tlm-component-architecture.html. [Accessed 12 May 2016].

[31] I. Synopsys, "Synopsys Virtual Prototyping Solution," [Online]. Available:

http://www.synopsys.com/prototyping/virtualprototyping/Pages/default.aspx.

[32] Synopsys Inc., "Platform Architect," 2016. [Online]. Available: http://www.synopsys.com/Prototyping/ArchitectureDesign/Pages/platform-architect.aspx. [Accessed 12 May 2016].

[33] Synopsys Inc., "Virtualizer," 2016. [Online]. Available: http://www.synopsys.com/Prototyping/VirtualPrototyping/Pages/virtualizer.aspx. [Accessed 12 May 2016].

[34] The MathWorks Inc., "The TLM Generation Process," 2016. [Online]. Available: http://www.mathworks.com/help/releases/R2016a/hdlverifier/ug/tlm-generation-workflows.html. [Accessed 12 May 2016].