Design of a Flexible Integration Interface for PIL Tests

Abstract This article describes a concept for an integration interface for simulation with processor-in-the-loop (PIL). PIL is an important tool in model-based software development. It allows early verification of the implementation. The concept for the integration interface is an extension of previous approaches that includes automatic interface analysis at the source code level and allows the existing development environment to be used. As a result, this solution is highly portable and scalable. To implement the concept, a prototype for Simulink was created and verified based on a test scenario.

1 Introduction

Over the last few years, the use of model-based methods for software development has become more and more important. The big advantage of model-based methods is the early and continuous verification of the software design [1]. Safety standards from a wide variety of industries are taking this development into account by applying model-based methods. Two important examples are:

• ISO 26262 [2] for safety-related electrical/ electronic systems in road vehicles
• DO-178C [3] for software development in the safety-critical area of aviation

Although the advantages of model-based methods are clearly recognized by everyone involved, case studies have shown that there is a series of obstacles to overcome when implemented in real applications [4]. As is often the case, software development using model-based only goes smoothly when the entire toolchain is supported by the manufacturer of the modeling environment.

This toolchain consists of the following:

• Code generator
• Build environment
• Target platforms

Modeling environments currently do not provide a uniform exchange format, making it very difficult to transfer data to an external tool. Another difficultly is the use of manually created code since the toolchains available are designed especially for use with code generated automatically by code generators.

Processor-in-the-loop (PIL) is used in model-based development to test an algorithm that has been derived from a model on a hardware platform or an instruction set simulator. With the help of a code generator, it is possible to implement a model in source code so that an executable application can be created that will run in the target environment.

This article presents a proposal for a scalable integration interface for PIL simulations in Simulink. In comparison to previous implementations, the proposed interface provides significantly greater flexibility in that the structure of the source code after the generation is analyzed and an appropriate interface for conducting a PIL test is created dynamically based on this data. In addition, a debugger is used for communication with the target platform so that almost every target platform can be supported without additional adjustments.

After that, this article provides an overview of model-based development methods in the framework of ISO 26262 and PIL simulations with Simulink. The limitations of the previous approach are discussed and advancements to the approach are presented. Finally, the new approach is demonstrated using a sample scenario.

2 Model-based development according to ISO 26262

A reference workflow (Fig. 1) has been established for the use of model-based development methods to enable fulfillment of the requirements in various safety standards.
Just like in the conventional development approach, the first step is to determine the requirements that describe the response of the software to be created together with all required

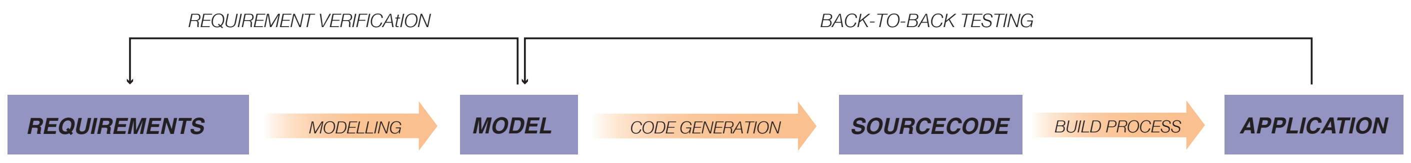


Figure 1: Recommended Workflow for model-based development [5]

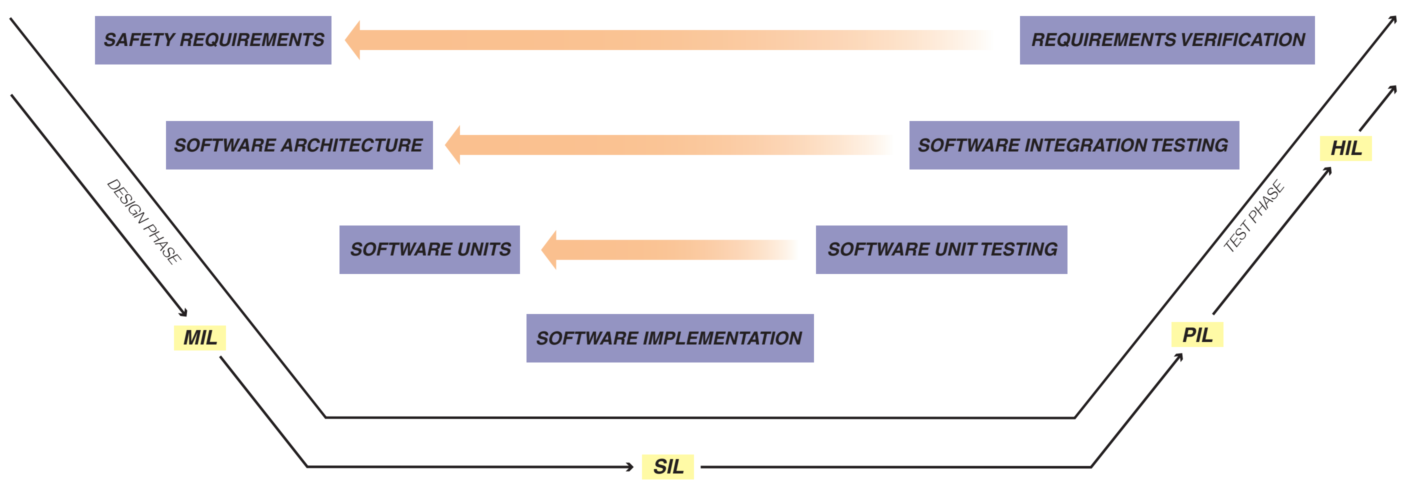


Figure 2: Softwaredevelopment according to ISO 26262

details. The application of model-based methods now offers the ability to depict the requirements very easily in the form of graphic models. In contrast to purely textual requirements, the big advantage of models is that they are able to clearly describe the response of a system. With the help of a modeling environment, it is possible to use models for software-in-the-loop (SIL). This allows the underlying requirements to be verified early on [6].

When using model-based methods, it is very easy to generate source code since there is a series of code generators available on the market that are able to automatically create an equivalent source code representation from an executable model. With the help of this source code, it is possible to generate applications that can be tested on a hardware platform or an instruction set simulator.

The workflow of the software development process in ISO 26262 (Fig. 2) is based to a great extent on the well-known V-model. This model specifies that after the software design and implementation phases, the software must pass through a verification process consisting of several phases. The availability of executable models opens up additional possibilities for verifying the results of the individual subprocesses. The correctness of a model can also be determined through simulations in combination with test cases that are based on the requirements. An application that has been created based on a model can also be checked with the help of back-to-back testing methods that compare the response of the model and the application to each other.

Models can be used with a series of different simulation modes for verification and validation purposes [7]. The term SIL refers to the execution of an application that is based on a model in the host environment. In the course of the simulation, test stimuli are applied to the application, and the return values are measured and verified. Direct extensions of this procedure are PIL simulations. In this case, an application is compiled for use in the subsequent target environment and executed on a hardware platform or in a simulator. The last stage of model-based testing is hardware-in-the-loop (HIL) simulation in which the application is tested in a complex hardware environment. All modes offer the ability to execute back-to-back testing in which the same test stimuli are applied to the model and the application, after which the output values of both are compared. If an executable model is used for the simulation, then the method is referred to as model-in-the-loop (MIL).

To avoid having to increase the complexity of the test because there are too many differences between the two environments, the goal should be to integrate the PIL simulation into the test procedure as early as possible. Evaluation boards are an easily accessible hardware solution to this problem. atform

3 PIL simulation with Simulink

The Simulink modeling environment allows the user to create executable models with help of a graphic interface and test the models in various simulation modes. In the case of PIL, a special block must be integrated into the existing model. The

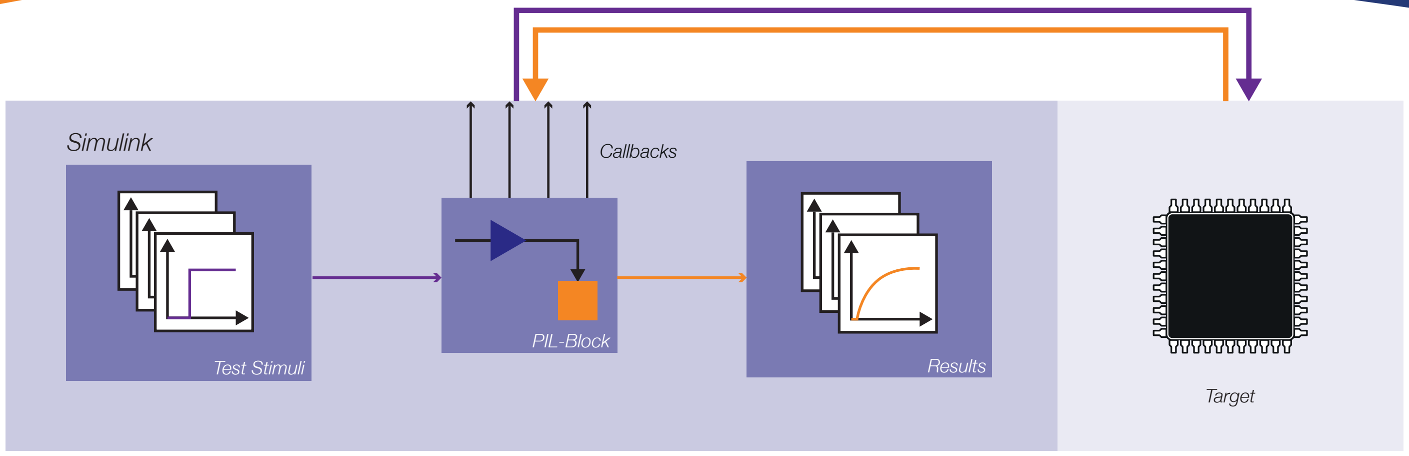


Figure 3: PIL with Simulink

job of the PIL block is to connect the target environment to Simulink and control the execution of the algorithm to be verified on the target platform [8].

The functionality of the PIL block is determined by a customizable S-function. It contains a set of callbacks that are used for integration into the target environment (Fig. 3). For this purpose, each callback is linked in advance to certain simulation results and is called up when the corresponding event occurs. Each call then triggers the action previously assigned to it in the target environment. This additional level of abstraction allows you to link a wide variety of different target platforms to Simulink. Whether the target platform is a real hardware platform or a simulation environment is not important in this case. However, it is necessary for each target platform to support communication with Simulink as well as the ability to control it using callbacks. Code generators in particular make it much easier for the users to generate a compatible interface for the required callbacks.

In the case of PIL, the algorithm is not normally run in real time in the target environment, but is executed incrementally in each simulation interval [9]. In each interval, all parameters required must be initialized, the input values passed to the target environment, and the resulting output values read.

4 Limitations

The previous implementation of PIL simulations with Simulink had a series of disadvantages when used in actual applications:

- The drivers used for communication with Simulink must be customized to match the configuration of the particular target platform.
- The drivers used for communication with Simulink must be customized to match the configuration of the particular target platform.
- Toolchains are only designed for the use of automatically generated source code.
- There is no support for the migration of legacy modules already containing source code.
- Allocation conflicts can arise when using interfaces on the target platform that are not separated functionally from the application layer for communication with Simulink.
- Restricting the number of PIL blocks simultaneously allowed at the model level [10] limits the scalability.

5 Further development of the integration interface

To compensate for the disadvantages of the previous solutions for executing PIL simulations, a new concept for a flexible integration interface was developed. The goal was to develop a universal approach that is independent from the code generation process used and that can be ported to new target platforms very easily.

To ensure that the new approach can be combined flexibly with automatically or manually generated source code, the interface for the S-function callbacks is created dynamically. It is necessary in this case to analyze the structure of the source code when generating the PIL infrastructure. Furthermore, a debugger was integrated into the toolchain as new communication link between Simulink and the target platform. From the point of view of Simulink, the debugger provides a uniform

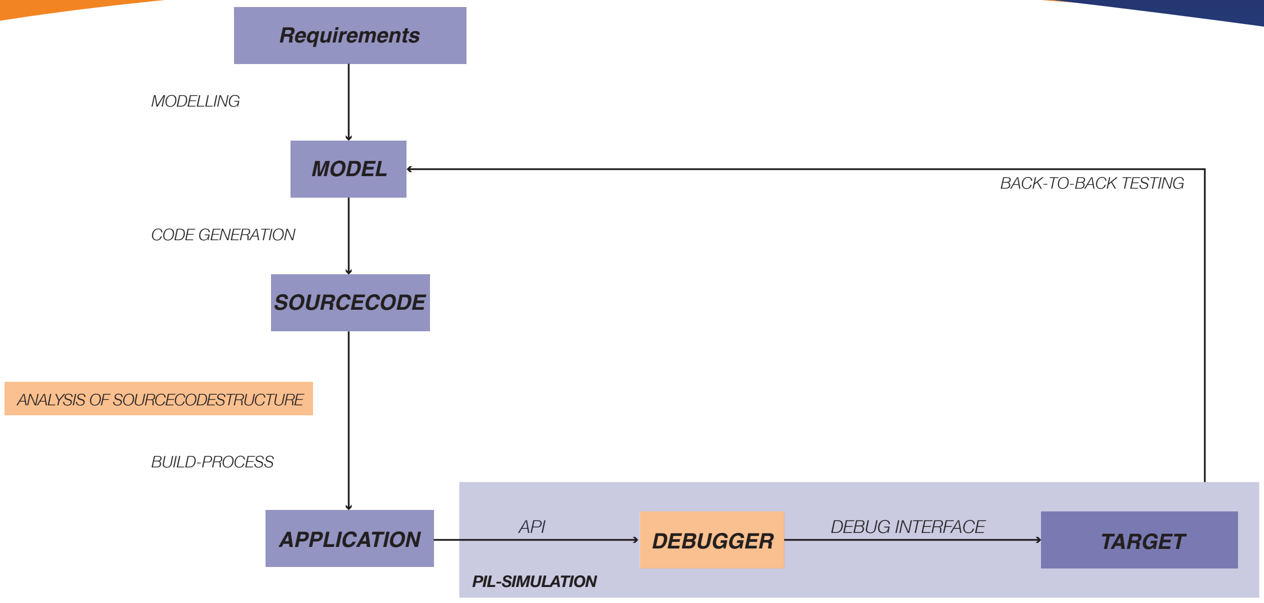


Figure 4: Design of the integration interface

interface for connection to various target platforms. In this manner, development environments that were already used during the software design phase can also be reused seamlessly to execute the PIL simulations. A schematic diagram of the proposed integration interface is shown in Fig. 4.

5.1 Analysis of the source code structure

The user has the ability to configure the interface between Simulink and the application on their own. For this purpose, the structure of the source code is analyzed and displayed in a graphic. Via a dialog, the user can define a rule for the mappings between the S-function callbacks and the functions in the source code. In the next step, the parameters can then be linked in the model level to their counterparts in the source code level. After the user has completed the configuration of the interface, appropriate wrappers are automatically generated for the target environment as links between the S-functions and the source code (Fig. 5). The job of the wrappers during the simulation is to implement the callbacks output by the PIL block so that the assigned functions are called and the variables are set to the correct values. In additional, source code is generated to create the S-function of the PIL block based on the interface configuration.

The method described is highly portable since it can be used in combination with code generators as well as with manually created source code.

Structural differences caused by different code generation processes can be taken into account by the interface configuration. In order for the new integration interface to be used effectively in a wide range of environments, the focus of development must be placed on the automatic analysis of the code structure

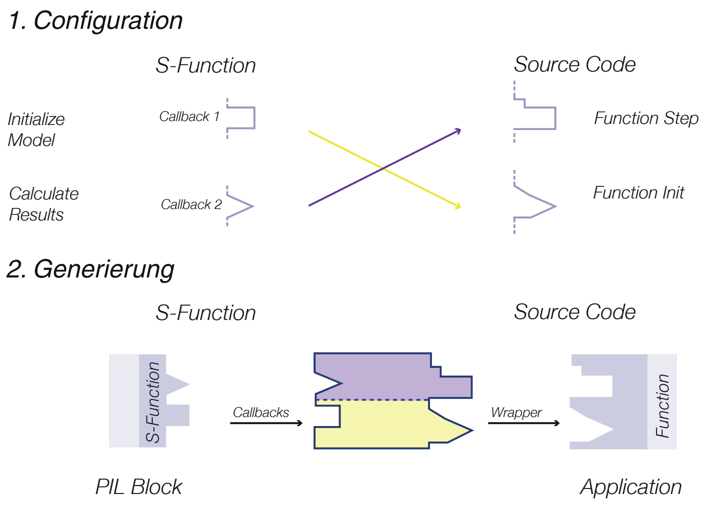


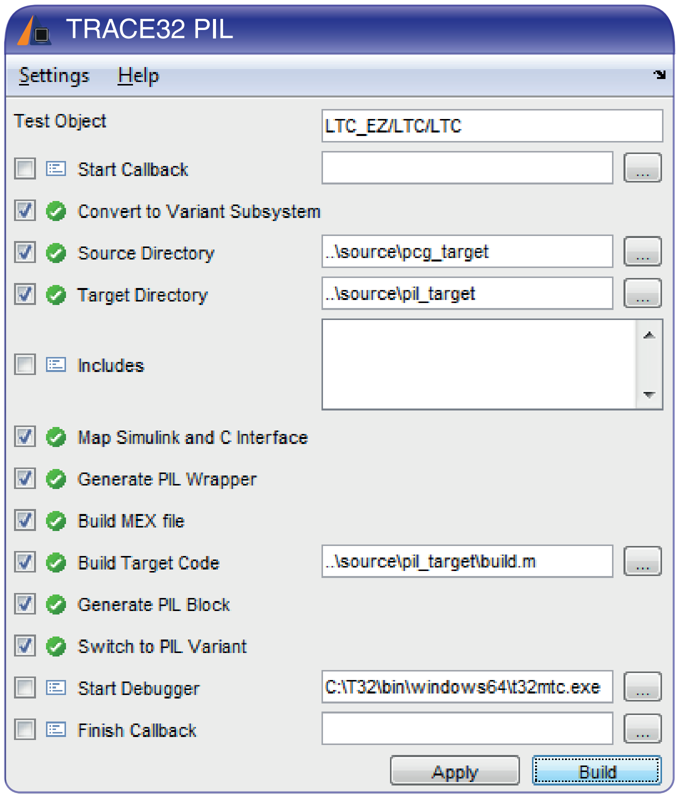
Figure 5: Analysis of source code and model interface

5.2 Uniform connection of the target environment

The second important factor for easy portability of the integration interface is the use of a debugger with an API interface. This allows different target platforms to be connected to the modeling environment over a generic interface. No other software customizations are necessary for this purpose. Using special interfaces such as JTAG, for example, the debugger can then control the execution of the program in the target environment and access the model parameters. It is possible to ensure that the debugger offers support in principle for the target environment as early as the software design phase.

5.3 Design of a prototype of the integration interface

To implement the proposed concept in an actual application, a prototype of the integration interface was developed. The prototype consists of a framework with a graphic interface (Fig. 6) for the generation of the PIL infrastructure. The user can thus specify all required settings one after the other to covert a block of the model to a PIL block. If not all steps are required, then it is possible to disable the steps individually. Starting the simulation now triggers the automatic execution of a PIL simulation with the help of the target environment. The prototype is able to analyze source code written in the programming language C and process functions with scalar arguments. To execute the PIL simulation, support for the TRACE32 API interface was implemented. Furthermore, the build process can be modified to integrate the automatically generated wrapper and subsequently load the application in the target environment.


TRACE32 PIL

Settings Help

Test Object	LTC_EZ/LTC/LTC
☐ Start Callback	
☑ Convert to Variant Subsystem	
☑ Source Directory	..\source\pcg_target
☑ Target Directory	..\source\pil_target
☐ Includes	
☑ Map Simulink and C Interface	
☑ Generate PIL Wrapper	
☑ Build MEX file	
☑ Build Target Code	..\source\pil_target\build.m
☑ Generate PIL Block	
☑ Switch to PIL Variant	
☐ Start Debugger	C:\T32\bin\windows64\t32mtc.exe
☐ Finish Callback	

Figure 6: Graphical User Interface

Due to its modular design, the proposed approach reaches a high level of scalability. PIL blocks can be combined flexibly with the other elements of the modeling environment so that it is possible to slightly extend existing models and easily integrate into new models. In additional, using the API interface of the debugger also allows multiple target environments to be connected simultaneously.

6 Example scenario

For the purpose of verification of the prototype, it was used for the adaption of an existing test scenario. The test scenario consists of back-to-back testing of a light control system using SIL. The goal of using the new integration interface was to port the scenario to a hardware platform with a TriCore processor. The following components were used for the implementation:

- Simulink R2014b
- EZTEST
- TASKING VX toolset for TriCore v4.3r3
- TRACE32
- TriBoard TC297TF

The output signal of the light control system is the signal used to switch the light on and off. This signal is activated and deactivated based on the input parameters defined. With the help of a test framework, test stimuli were applied to the inputs of the control model over the course of the simulation, and the output was then compared to the permissible reference values (Fig. 7). The corresponding C source code was derived from the model of the light control system in the context of the SIL simulation. This allowed the code to be used for the PIL simulation without requiring any changes.

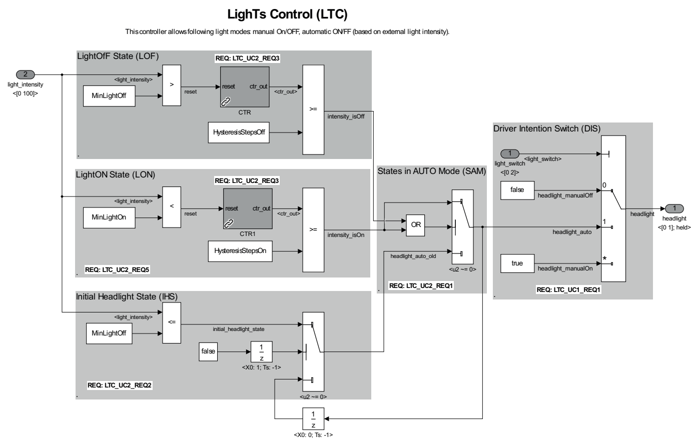


Figure 7: Model of a lights control algorithm

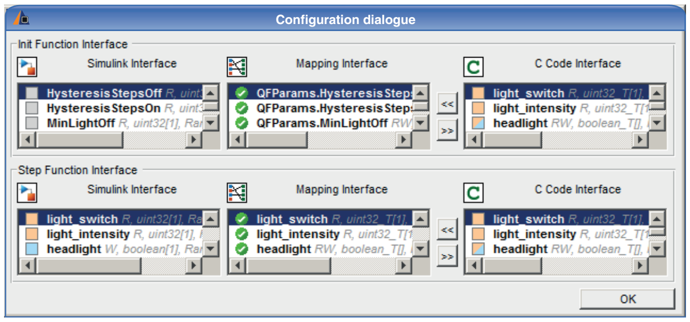

Configuration dialogue
Init Function Interface
Simulink Interface
Mapping Interface
C Code Interface
HysteresisStepsOff R, uint
HysteresisStepsOn R, uint3
MinLightOff R, uint32[1], Ra
QFParams.HysteresisStep
QFParams.HysteresisStep
QFParams.MinLightOff RW
<<
>>
light_switch R, uint32_T[1],
light_intensity R, uint32_T[1
headlight RW, boolean_T[],
Step Function Interface
Simulink Interface
Mapping Interface
C Code Interface
light_switch R, uint32[1], Ra
light_intensity R, uint32[1],
headlight W, boolean[1], Ra
light_switch R, uint32_T[1],
light_intensity R, uint32_T[1
headlight RW, boolean_T[],
<<
>>
light_switch R, uint32_T[1],
light_intensity R, uint32_T[1
headlight RW, boolean_T[],
OK

Figure 8: Interface Configuration

Using the graphic user interface of the prototype, the interface was analyzed and configured. For connection to the callbacks of the S-function, the source code contains a function for initializing all model parameters and the determination of the output of the model after executing a step of the simulation. During the structural analysis of the source code, both functions were detected successfully, so it was possible to create an appropriate configuration for the assigned callbacks (Fig. 8). Based on this interface configuration, it was possible to integrate the S-function as well as the wrapper into the build process. During the simulation, the hardware platform was successfully controlled through the API interface of the debugger. The evaluation of the back-to-back test after the test had completed showed that the results of the PIL and MIL simulations matched (Fig. 9).

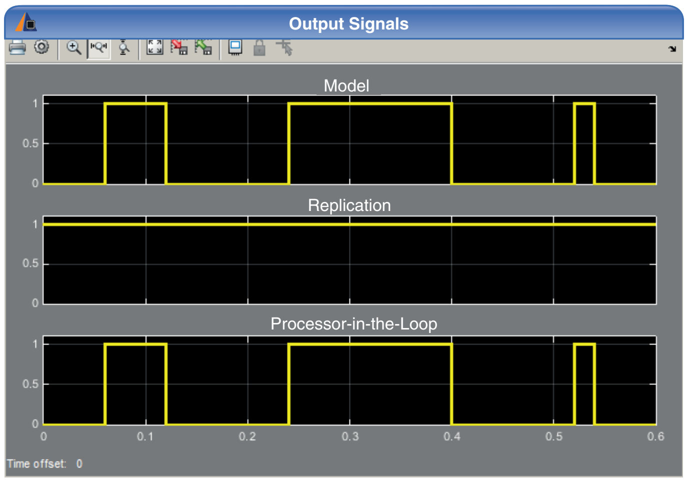


Figure 9: Results Back-to-Back Test

Summary and outlook

This article presents a concept for a flexible integration interface to execute PIL simulations that in the future should contribute to eliminating the obstacles to the use of PIL in practical applications. The following goals are realized by the concept:

- Flexible support for code generators and manually created code
- Easy portability to different target platforms
- Seamless integration into existing development processes
- Cost neutrality through the reuse of the existing development environment

It was also possible to verify the implementation of the integration interface in a prototype based on a specific test case.

In the future, the robustness of the prototype in terms of its interactions with various toolchains, programming languages, and test concepts must be examined and extended where required. Furthermore, the scalability of the solution needs to be tested based on additional scenarios.

For more information, please visit: www.lauterbach.com/1877

Bibliography

[1] E. Dillaber, et al. Pragmatic Strategies for Adopting Model-Based Design for Embedded Applications. SAE Technical Paper. 2010.

[2] ISO 26262. Road vehicles - Functional Safety. 2011.

[3] RTCA/EUROCAE. DO-331/ED-216 - Model-Based Development and Verification Supplement to DO-178C [ED-12C] and DO-278A [ED-109A]. 2011.

[4] S. Kirstan, and J. Zimmermann. Evaluating Costs and Benefits of Model-Based Development of Embedded Software Systems in the Car Industry – Results of a Qualitative Case Study. In Proceedings Workshop C2M: EEMDD - From Code Centric to Model Centric: Evaluating the Effectiveness of MDD. ECMFA. 2010.

[5] M. Beine. A Model-Based Reference Workflow for the Development of Safety-Critical Software. Embedded Real Time Software and Systems. 2010.

[6] S. Lillwitz, D. Krob. Model-Based Development of Safety-Critical Software: Safe and Efficient. MEDengineering. 2012. Available online at http://www.dspace.com [09/2015].

[7] S. A. A. Samie. Towards a Model Driven Approach in the Develop-ment and Validation of Real-Time Embedded Systems. International Journal of Advances in Engineering and Technology. 2015.

[8] T. Erkkinen and M. Conrad. Verification, Validation, and Test with Model-Based Design. SAE Technical Paper. 2008.

[9] S. Lentijo, et al. A New Testing Tool for Power Electronic Digital Con-trol. In Proceedings of the IEEE 34th Annual Power Electronics Specialists Conference. 2003.

[10] D. Divakar and K. G. Ashwini. A Processor in Loop Test Method for Life Critical Systems. In Proceedings of the International Colloquiums on Computer Electronics Electrical Mechanical and Civil. 2011.