2021 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE OCTOBER 26-27, 2021

Language Agnostic Communication for SystemC TLM Compliant Virtual Prototypes

Smruti Khire, Infineon Technologies, Bangalore, India (SmrutiSanjay.Khire-EE@infineon.com)
Kunal Sharma, Infineon Technologies, Bangalore, India (Kunal.Sharma@infineon.com)
Vishal Chovatiya, Infineon Technologies, Bangalore, India (Vishal.Chovatiya@infineon.com)

***Abstract*—Interaction with the Virtual Prototype (VP) during runtime is crucial for fault injection and the subsequent system testing that contributes to early software development, architecture exploration and verification. However, the VP is delivered as an executable and establishing runtime interaction comes with a lot of challenges in terms of the rigid dependencies, costs and efforts involved. In this paper, we propose a solution to eliminate the rigid dependencies by deploying a server-client based framework. This plug and play utility is capable of language agnostic communication with any industry standard VP in an easy and efficient manner.**

Keywords—virtual prototype; simulator gateway; runtime interaction, ,simulation probe, SystemC, TLM2.0

I. INTRODUCTION

A Virtual Prototype, also known as a simulator or digital-twin, is a fully functional software model of an electronic system. Given the rapid increase in design complexity of modern-day System on Chips (SOCs), the phases of development and verification of chips have become longer. A VP enables early software development and verification. For safety and robustness of products, especially in automotive, medical and defense applications, the VP is useful for aggressive hardware testing, performing sophisticated experiments in a cost-efficient manner and analyzing the system performance thoroughly.

The VP is developed using industry standard SystemC and TLM-2.0. It is usually delivered as an executable binary (shared library, .exe, etc.) by the hardware vendors. As a result, it is difficult to establish runtime interaction with the VP entities like ports, registers, memory etc. This interaction is vital for the purpose of fault injection, updating bit-fields, driving internal signals, etc. The runtime interaction feature is provided in a high-level programming language to the user. There are licensed tools as well as other solutions available in the market which address this challenge. However, they tend to introduce a tight binding of the tools and dependencies on certain packages and language-specific binaries in the VP development. Thus, also limiting the possibility for a customized runtime interaction feature.

II. MOTIVATION

It is established that the existing solutions pose limitations in facilitating runtime interaction with the VP. Additionally, the time and effort involved to make the VP compatible for runtime interaction is also high. The challenges include complexity of setup and debugging, lots of busy code even for exposing simple utilities. These challenges add to the complications while introducing new features to the project. For example, Simplified Wrapper and Interface Generator (SWIG) [1] is an open-source software tool that supports interfacing of programs or libraries written in C++ with high-level programming languages (like Python, Perl, Java, etc.). However, it has a drawback of being tightly coupled to the system and challenges in supporting additional datatypes. This negatively impacts the extensibility of the system. Tools such as Simprobe (Synopsys proprietary) come with a licensing cost.

Thus, we aim to introduce a state-of-the-art approach to overcome the challenges for the industry standard SystemC/TLM-2.0 compliant simulators with a lightweight and pluggable utility i.e. *Simulator Gateway*. The Simulator Gateway provides a scripting interface for customized runtime interaction. For example, the user can specify the intent in the runtime interaction script and observe the execution of the requested task.

III. SIMULATOR GATEWAY ARCHITECTURE

The Simulator Gateway implements a wide range of features to interact with the VP during runtime. These include features to read/write to ports, registers or signals, to inject fault into the system, scheduling events at a specific time and certain helper functions to get the status of the simulation. These features are generic to most industry standard simulators; hence this framework can be readily plugged into any VP. The major functionalities available in the VP are listed in the table below:

Read and Write Interfaces	Port Read/Write
	Register Read/Write
	Software Variable Read/Write
Fault Injection Interfaces	Inject Fault
Scheduling Stimulus Injection	Wait on Time
	Wait on Event
General Helper Functions	Get Current Simulation Time

Table 1- Functionalities of the Simulator Gateway

The Simulator Gateway is developed with a mindset to overcome the rigid requirements and dependencies in the existing solutions. It employs OSCI standard API's to connect and initiate write/read action on ports. Likewise, to interact with memory/registers it employs the TLM-2.0 protocol. The framework is based on a Server-Client architecture that supports runtime interaction with the VP. Figure 1 shows the Server-Client architecture where the VP is on the Server side with the Simulator Gateway loosely attached to the VP and the script for runtime interaction on the Client side. The Server and Client interface layers facilitate the connection and communication on their respective sides. The foundation of this framework is based on three major parts - Network Programming, Data Serialization & Deserialization and Multithreading.

Once simulator is launched, the Simulator Gateway on the server side gets ready to accept connection requests from the runtime interaction script on the client side. This connection is established by the means of Network Programming. The client side can send messages in any programming language. Here Data Serialization and Deserialization come into picture which help both the sides to interpret the messages that are communicated. Once the message is interpreted, the simulator performs the requested task. At the same time it continuously listens to the client side to check for incoming requests, thus introducing a Multithreading aspect.

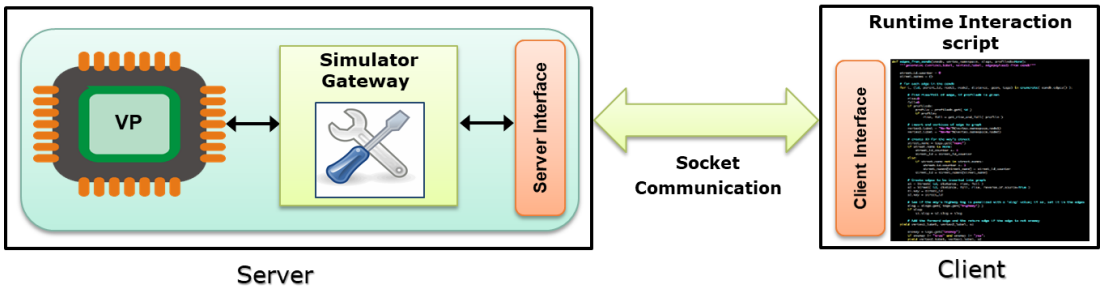

Figure 1- Server-Client based architecture for runtime interaction

Below is an elaboration of the three major parts that form the foundation of the Simulator Gateway:

A. *Network Programming*

The Simulator Gateway is based on the general idea of the Transmission Control Protocol. On the Server side, the Boost.Asio [2] library is used for establishing socket communication. Boost.Asio is a cross-platform C++ library used for network programming that provides us with a consistent asynchronous model using a modern C++ approach. Once the simulator starts, a socket is launched that waits for a client connection request. Upon accepting the request, it reads the incoming data and sends an appropriate response back to the client side in an asynchronous fashion. Similarly, on the client side, the network programming is done by using the language-specific socket communication libraries. For example, if the client side is programmed using Python, the socket [3] library can be used to establish connection and communicate with the server.

B. *Data Serialization and Deserialization*

As highlighted earlier, the main objective of the framework is to make the system language-agnostic and eliminate the rigid dependencies. Hence, the data serialization and deserialization play a vital role in order to pass the data across the network in a language-independent manner. The serialized request from the client side is sent over the network where the server deserializes and interprets it according to the fixed communication protocol. The same methodology is followed while sending responses back to the client. This enables the user to use any scripting language provided (s)he complies with the network communication protocol. The network communication protocol is defined using Google Protocol Buffers [4]. Protocol Buffers provide a language-neutral, platform-neutral and extensible way to serialize structured data. The message structure is defined in a protobuf file with the data members required in the specific message. These data members can be of different scalar value types like string, bool, uint32, int32, float, etc. For example, if the request for Register Write is to be made then the register name will be a data member of the message with its type as a string. The protobuf file is then passed over the protocol buffer compiler - protoc compiler which generates the source code for C++ on the server side and for the scripting language on the client side. Protocol Buffers currently support generated code in C++, Python, Ruby, C#, Objective-C, Java, Go, Dart, with many more languages to come. The generated source code implements a class for automatic encoding and parsing of the data. It has getter and setter functions for the data members of the message. As a result, the user can set a particular data member on the client side using the setter function, encode it and pass it to the server side. Then it can be parsed and interpreted with the help of the getter function. This generated source code constitutes an integral part of the Server and Client Interface layer as shown in Figure1.

C. *Multithreading*

The server side of the framework is responsible for performing two tasks in parallel - first is to perform the simulation and the second is listening to the incoming client requests. Therefore, the main process consists of two threads - one is the simulation thread and second is the server thread as shown in Figure 2. The server thread is responsible for listening to the incoming client requests. Upon reception, the incoming message is read and the simulation thread is notified. The server thread then transitions into a blocked state until the simulation thread completes the execution of the task for the requested intent. Once the request is fulfilled, the server thread comes back to its running state and sends back the appropriate message to the client that is waiting for a response from the server. The synchronization between the threads is a deciding factor for the reliable execution of the requests from the clients. This is achieved by employing the standard C++ mutex and conditional variables.

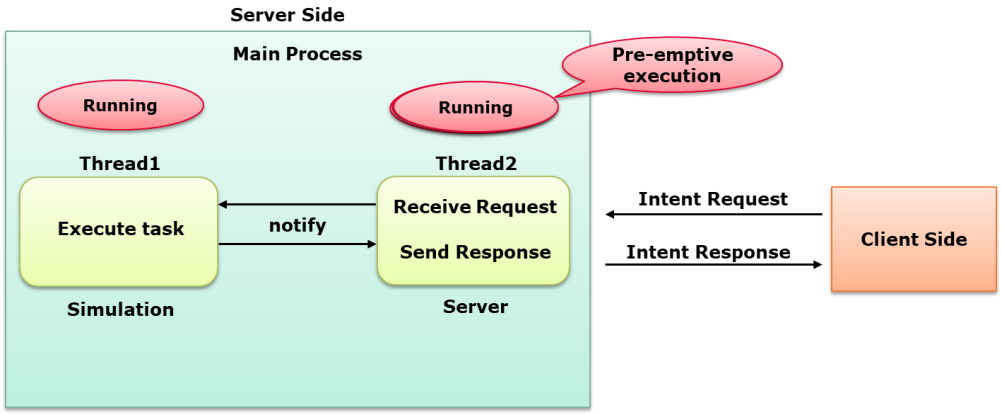

Figure 2 – Multithreading on Server Side

IV. BENEFITS

The Simulator Gateway provides numerous generic functionalities that enable the users to easily get the maximum benefits out of the system. The utility is lightweight and easily pluggable for SystemC/TLM-2.0 compliant simulators. Since it follows a language-agnostic approach, the user gets the complete flexibility of utilizing any high-level language for the client side. For example, tasks like system level testing can be automated via scripts. Thus, it is easy to create an aggressive test environment. In scenarios like architectural exploration or pre-silicon validation, you can validate use cases in any higher-level language thus eliminating the need to write firmware in low level languages like C/C++. Moreover, this utility can also help in customer software bring-up or identify gaps while creating board support package (BSP). It provides a scope to extend the functionalities as per the requirements of the user in a clean and easy manner. It eliminates the dependency on any licensed tools. The ease of extensibility and scalability results in very less integration efforts.

V. RESULTS

The comparison with existing tools/approaches shown in the following table sums up the impact achieved with this solution:

Criteria	Existing Tools	Proposed Solution
Language-Independence	Need to generate a new wrapper for every language	Language-Agnostic
Scalability and Extensibility	Low, Tightly Coupled	High, Loosely Coupled
Installation requirements of packages for VP development	Needed	Not needed, since it is a Plug and Play utility
License Cost	High	Zero
Integration effort	High	Minimal

Table 2 – Comparison between existing and proposed solution

VI. Conclusion and Future Scope

The Simulator Gateway is almost a zero/minimal cost solution for enabling runtime interaction capability in any SystemC/TLM-2.0 compliant VP. The server-client based architecture provides flexibility for its use with different client-side scripting implementations. For example, the client side can be adapted to provide a Ruby based scripting interface. It also overcomes the rigid installation requirements for VP development. Moreover, this approach can be easily extended to provide any customized functionality to interact with VP during runtime.

Acknowledgment

We would like to express our heartfelt gratitude to former contributors and employees of Infineon Technologies, Chethan Nayak and Nishant Kathiriya, who have been of great help during the implementation of this project.

References

[1] SWIG- Simplified Wrapper and Interface Generator, http://www.swig.org/

[2] Boost.Asio- C++ library for network programming, https://www.boost.org/doc/libs/1_76_0/doc/html/boost_asio.html

[3] socket- low-level networking interface in Python, https://docs.python.org/3/library/socket.html

[4] Google Protocol Buffers- for Data Serialization & Deserialization, https://developers.google.com/protocol-buffers