Creating Heterogeneous Simulations by Interoperating SST with Hardware Description Frameworks

Sabbir Ahmed
Booz Allen Hamilton
ahmed_sabbir@bah.com

Abstract

Implementing new computer system designs involves careful study of both programming models and hardware design and organization, a process that frequently introduces distinct challenges. Hardware and software definitions are often simulated to undertake these difficulties. Structural Simulation Toolkit (SST), a parallel event-based simulation framework that allows custom and vendor models to be interconnected to create a system simulation [1], is one such toolkit. However, SST must be able to support models implemented in various hardware-level modeling languages (PyRTL, SystemC, Chisel, etc.) and hardware description languages (VHDL, Verilog and SystemVerilog). Establishing communication with these modules would allow SST to interface numerous existing synthesizable hardware models. SST Interoperability Toolkit (SIT) is a toolkit developed to provide interoperability between SST and other frameworks. SIT aims to achieve this capability in a modular design without interfering with the kernels by concealing the communication protocols in black box interfaces.

1 Introduction

The increasing size and complexity of systems require engineers heavily rely on simulation techniques during the development phases. Typically, simulations of these complex systems require both custom and off-the-shelf logic functionality in application-specific integrated circuits (ASIC) or field programmable gate arrays (FPGA). High-level commercial tools simulate and model these components in their native environments. On the other side, developers create the register transfer level (RTL) models representing the systems to simulate them with computer-aided design (CAD) tools and test benches. These duplicative strategies require a method that simulates the entire system in one heterogeneous model.

SST is an event-based framework that has the capabilities to simulate not only functionality but timing, power or any other information required. Each SST components can be assigned a clock to synchronize tasks. They communicate events with each other via SST links by triggering their corresponding event handlers. The SST models are constructed in C++ and consist of the functionality of the element, the definition of each links' ports and the event handlers. The models are connected and initialized through the SST Python module. PyRTL and SystemC are similar platforms that deliberately mimic hardware description languages in Python and C++ respectively. These system-level modeling languages provide event-driven simulation kernels along with signals, events and synchronization primitives.

Implementing a heterogeneous system to synchronize signals and events between the frameworks would allow the developers to work cooperatively and efficiently. This paper provides a demonstration of the interoperability by simulating a vehicular traffic intersection with traffic lights driven by PyRTL and SystemC processes.

Note: For the sake of simplicity and consistency in the nomenclature, the languages, toolkits or libraries in the following categories will be simply labeled as "HDL" or "external HDL":

- hardware description languages (SystemVerilog, Verilog, VHDL, etc.)
- hardware level modeling languages (Chisel, PyRTL, SystemC, etc.)

2 Black Box Interface

SIT conceals the communication implementation in black box driver files. This strategy allows the SST component to connect with the HDL processes via SST links as if they were a component itself.

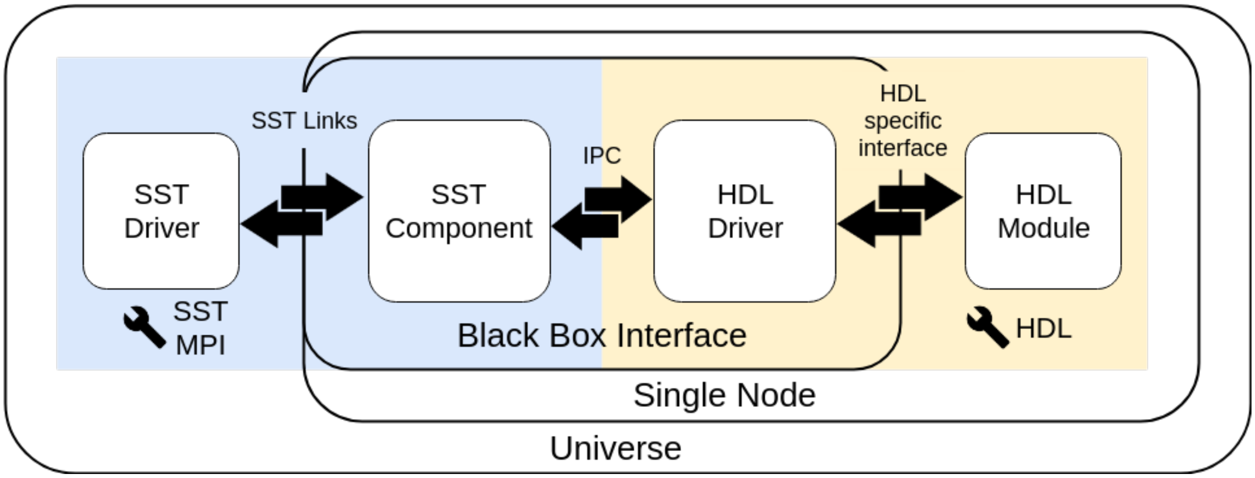

Figure 1: Components of SIT

The interface consists of:

1. an HDL driver
2. an SST component

2.1 HDL Driver

Each HDL modules must have their corresponding driver file to interoperate with the SST kernel within the black box interface. The language must be able to bind to interprocess communication (IPC) ports to send and receive data. The driver must be compiled separately from the SST component.

2.2 SST Component

The black box SST component consist of the following methods:

- Constructor
- `void setup()`
- `bool tick(SST::Cycle_t)`
- `void handle_event(SST::Event *)`

2.2.1 Constructor

2.2.2 setup

2.2.3 tick

The method simply returns `false`.

2.2.4 handle_event

2.3 Boilerplate Code Generator

The toolkit includes a Python class that generates the boilerplate code required for the black box interface.

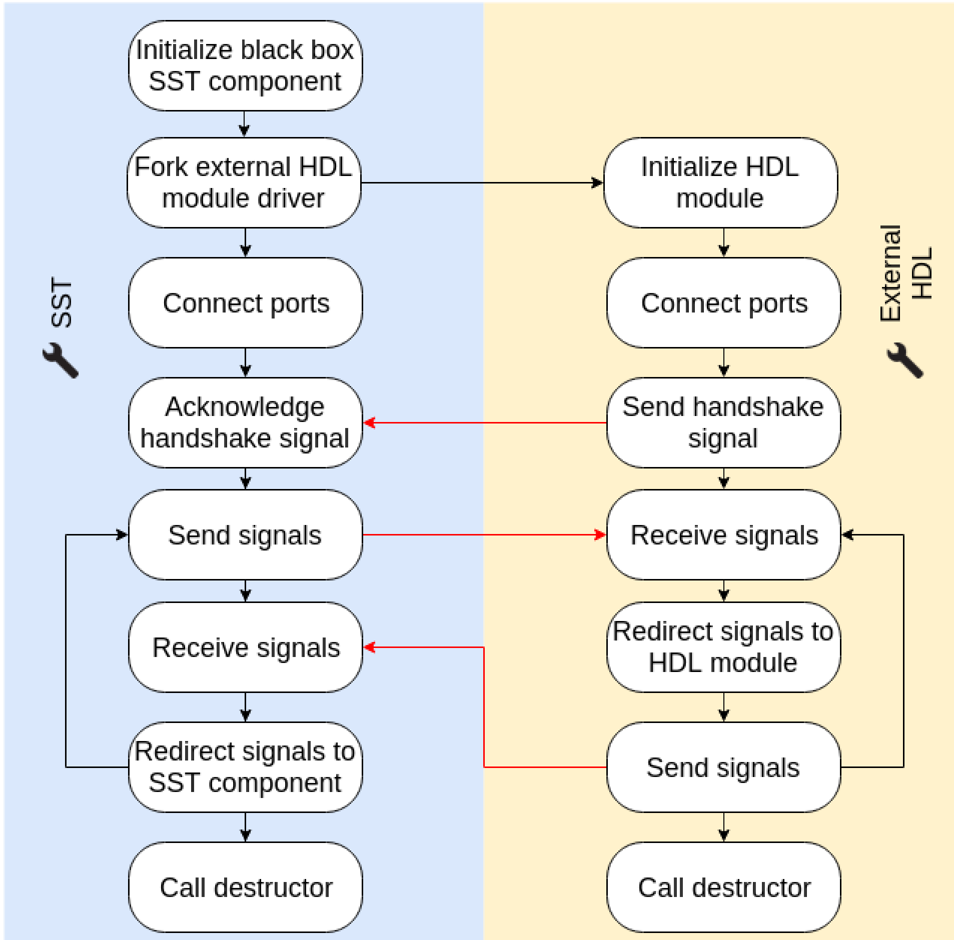

Figure 2: Black Box Interface Data Flow Diagram; Arrows Highlighted in Red Indicate Communication Signals

3 Communication

3.1 Inter-Black Box Communication

The data is represented in a standard vector of strings with the indices generated by the black box interface and then serialized with MessagePack methods [4]. The components inside the black box interface are spawned in the same node and therefore communicate via interprocess communication (IPC) transports. The following is a list of supported IPC transports:

1. Unix domain sockets
2. ZeroMQ

It is possible to integrate additional IPC protocols to the interface such as named pipes and shared memories.

3.2 SST-Black Box Communication

SST links are used to interface the SST component with the black box. The data is received as a `SST::Interfaces::StringEvent` object which is casted to a standard string. SIT provides a custom event handler as part of its black box interface to allocate the substring positions and lengths for the ports.

3.3 HDL-Black Box Communication

The HDL module utilizes its program specific mechanism of communication to interface the black box driver. The method may be source file inclusion or importing modules.

4 Extensibility

4.1 Communication

As mentioned in Section 3.1, it is possible to integrate additional IPC protocols to the interface by implementing a derived class of `sigutils::SignalIO` with customized sending and receiving methods. The base `sigutils::SignalIO` class provides methods of serializing and deserializing the data structures utilized within the black box interface. The derived sending and receiving methods would have to simply implement their specific approaches of reading and flushing buffers.

4.2 Interface

This entire paper focuses on the interoperability established between SST and SystemC processes. However, the concept was derived off of the efforts already established by the SST-PyRTL project [3]. In fact, a hybrid version of the Traffic Intersection Simulation has been implemented where both SystemC and PyRTL take control over one traffic light using the same IPC method. The black box SST component had to be provided distinct instructions to spawn and communicate with a SystemC and a non-SystemC process. Meanwhile, the other side of the black box interface consisted of a SystemC driver and a PyRTL driver with their respective native configurations for establishing communication with the parent process.

This extensibility was possible due to the generic structure of the black box interface. In theory, the interface is able to establish interoperability between SST and almost any other synthesizable HDL.

References

[1] SST Simulator - The Structural Simulation Toolkit. `sst-simulator.org`.

[2] 1666-2011 - IEEE Standard for Standard SystemC Language Reference Manual. IEEE, 9 Jan. 2012, `standards.ieee.org/standard/1666-2011.html`.

[3] Mrosky, Robert P, et al. *Creating Heterogeneous Simulations with SST and PyRTL.*

[4] "MessagePack." *MessagePack: It's like JSON. but Fast and Small.,* msgpack.org/index.html.