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 Interoperable (SSTI) is a toolkit developed to provide interoperability between SST and other frameworks. SSTI aims to achieve this capability in a modular design without interfering with the kernels by concealing the communication protocols in black box interfaces. This project includes a demonstration of the interoperability by simulating a vehicular traffic intersection with traffic lights driven by SystemC and PyRTL processes.

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 ASICs or FPGAs. 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.

Successful attempts have been made to establish interoperability between Structural Simulation Toolkit (SST) and the Python-based RTL language, PyRTL [3]. This project generalizes that capability to include other HDLs such as SystemC.

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 simplicity, the hardware-level modeling and description languages will be referred to "HDL" or "external HDL" in this paper.

2 Black Box Interface

SSTI 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.

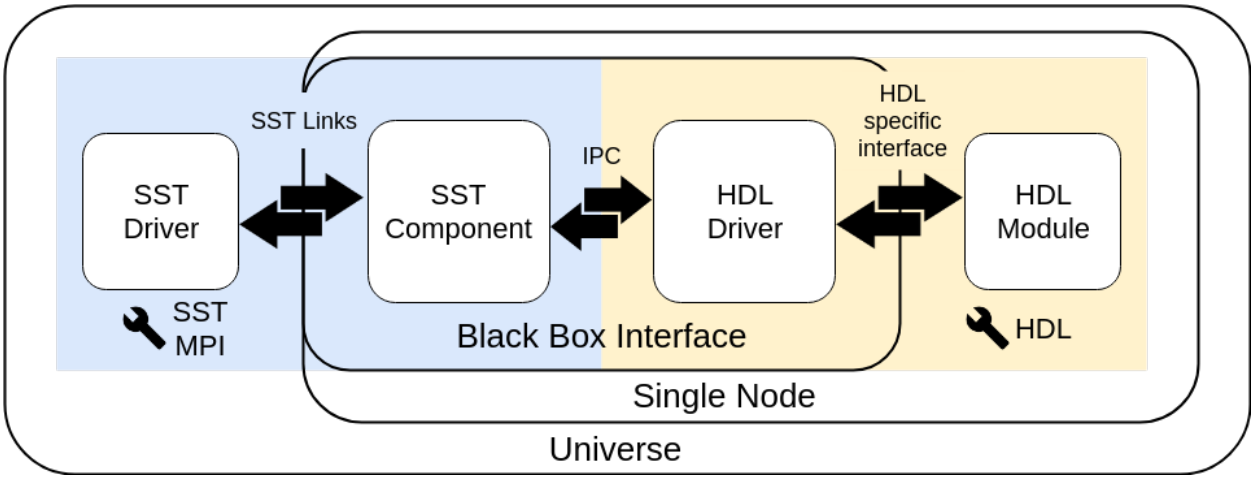

Figure 1: Components of SSTI

The interface consists of:

1. an HDL driver
2. an SST component
3. configurations for inter-black box communication

2.1 HDL Driver

Each HDL modules must have their corresponding driver file to interoperate 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

2.3 Configuration File

The interface also includes a header file with configurations for the communication between the components. These configurations include the number of elements being shared and their corresponding data structure indices. Some HDLs, such as PyRTL, do not require an extra configuration file.

2.4 Boilerplate Code Generator

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

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

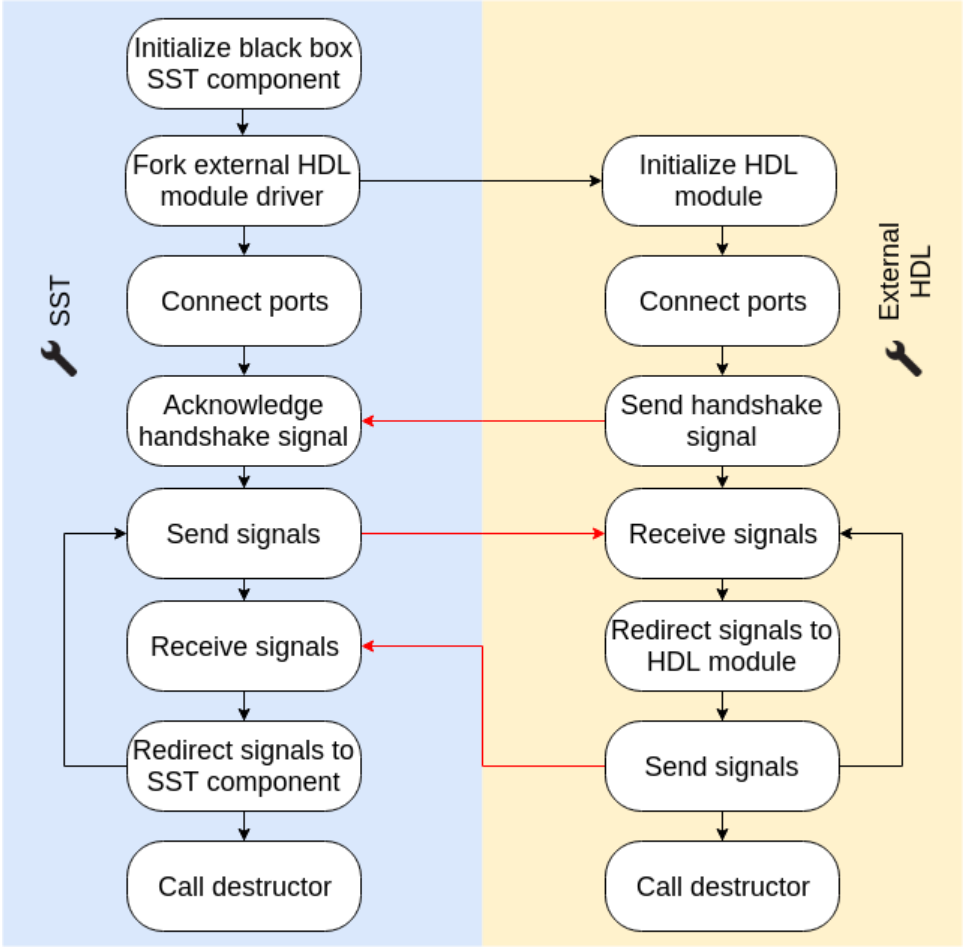

Figure 2: Black Box Interface Data Flow Diagram; Arrows Highlighted in Red Indicate Communication Signals

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

An SST component can interface the black box via standard SST links. The data is received as a `SST::Interfaces::StringEvent` object which is casted to a standard string. SSTI 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

4 Proof of Concept: Traffic Intersection Simulation

A simulation model has been developed to demonstrate the project. The model simulates a traffic intersection controlled by two traffic lights. A flow of traffic is simulated through a road only when its traffic light generates a green or yellow light with the other generating a red light. The number of cars in a traffic flow is represented by random number generators.

The concept of this simulation is derived from the original project that established interoperability between SST and PyRTL. [3]

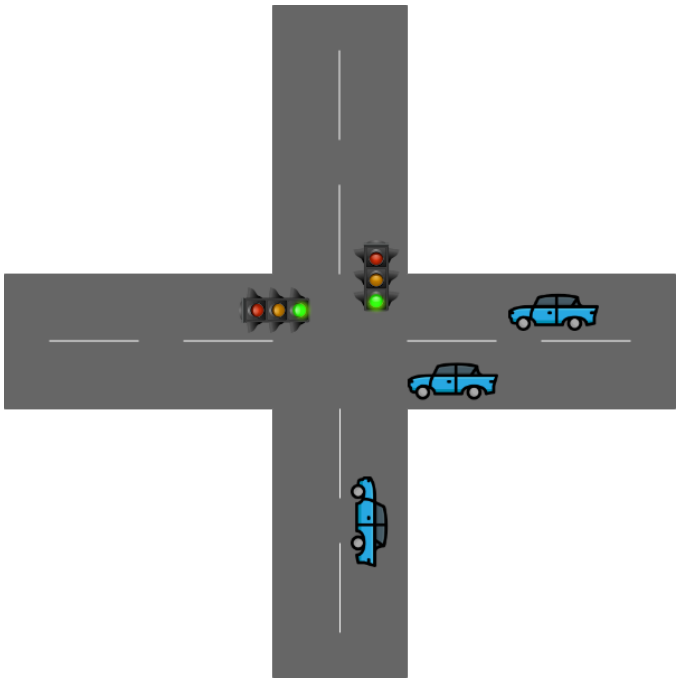

Figure 3: Simple Two-Road Intersection

4.1 SystemC and PyRTL Drivers

The simulation project includes a SystemC module and a PyRTL module with identical functionalities. Their corresponding drivers interact with the SST component `traffic_light`. The module is a clock-driven FSM of three states representing the three colors of a traffic light: green, yellow and red. The FSM proceeds to the next state when indicated by its internal counter initialized in the beginning. The input variables to the module include: the three durations for the three colors of the light, `green_time`, `yellow_time` and `red_time`, the preset variable `load` to initialize the FSM, and `start_green` to indicate if the first state should be "green" or "red".

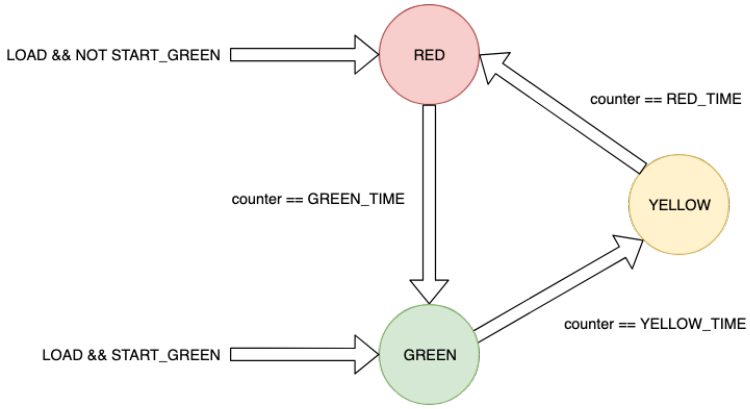

Figure 4: Traffic Light Finite State Machine

4.2 SST Components

The project also consists of three SST components: `car_generator`, `traffic_light_controller` and `intersection`. All the components with the exception of most of `traffic_light_controller` were inherited from the original project.

4.2.1 car_generator

The `car_generator` component consists of a random number generator that yields 0 or 1. The output is redirected to `intersection` via SST links.

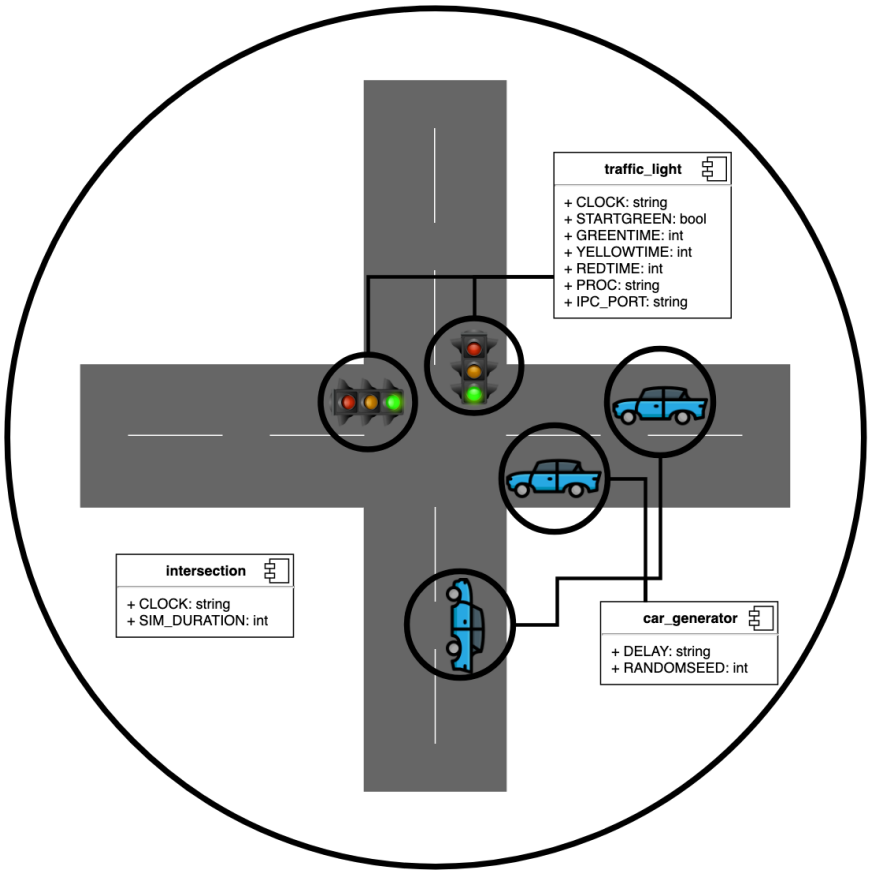

Figure 5: Simple Two-Road Intersection Represented by SST Components

4.2.2 traffic_light

The `traffic_light` component generates the light colors of the traffic lights using a simple finite state machine (FSM). The component delegates the FSM portion of its algorithm to the HDL module `traffic_light_fsm` and interprocedurally communicates with it via Unix domain sockets. The component initializes the FSM with the SST parameters and sends its output to `intersection` via SST links every clock cycle.

4.2.3 intersection

`intersection` is the main driver of the simulation. The component is able to handle n instances of `traffic_light` subcomponents and therefore expects the same number of `car_generator` instances. For the purposes of this simulation, two instances of the subcomponent pairs are set up. The driver keeps track of the number of cars generated and the color of the light per clock cycle for each subcomponent pairs and stores them in local variables. The variables are summarized in the end to generate statistics about the simulation.

The component does not check for any collisions in the intersection, i.e. comparing if both the `traffic_light` subcomponents yield "green" during the same clock cycle. The SST Python module is responsible for setting up the `traffic_light` components with the proper initial values.

4.3 Example Simulation

A sample output of the simulation has been generated and provided below. The SST components were linked along with the parameters in the SST Python module.

```
traffic_light-Traffic (SystemC) -> GREENTIME=30, YELLOWTIME=3, REDTIME=63, STARTGREEN=0
traffic_light-Traffic (PyRTL) -> GREENTIME=60, YELLOWTIME=3, REDTIME=33, STARTGREEN=1
car_generator-Car Generator 0 -> Minimum Delay Between Cars=3s, Random Number Seed=151515
car_generator-Car Generator 1 -> Minimum Delay Between Cars=5s, Random Number Seed=239478
intersection-Intersection -> sim_duration=24 Hours
```

```
intersection-Intersection -> Component is being set up.
traffic_light-Traffic (SystemC) -> Component is being set up.
traffic_light-Traffic (SystemC) -> Forking process "/path/to/traffic_light_fsm.o"...
traffic_light-Traffic (SystemC) -> Process "/path/to/traffic_light_fsm.o" successfully synchronized
traffic_light-Traffic (PyRTL) -> Component is being set up.
traffic_light-Traffic (PyRTL) -> Forking process "/path/to/traffic_light_fsm.o"...
traffic_light-Traffic (PyRTL)
    -> Process "/path/to/traffic_light_fsm_driver.py" successfully synchronized
intersection-Intersection -> ----------------------------------------
intersection-Intersection -> -------- SIMULATION INITIATED --------
intersection-Intersection -> ----------------------------------------
intersection-Intersection -> Hour | Total Cars TL0 | Total Cars TL1
intersection-Intersection -> -----+---------------+---------------
intersection-Intersection ->    1 |           618 |           369
intersection-Intersection ->    2 |          1238 |           719
intersection-Intersection ->    3 |          1851 |          1074
intersection-Intersection ->    4 |          2432 |          1426
intersection-Intersection ->    5 |          3041 |          1774
intersection-Intersection ->    6 |          3688 |          2121
intersection-Intersection ->    7 |          4290 |          2467
intersection-Intersection ->    8 |          4892 |          2813
intersection-Intersection ->    9 |          5467 |          3175
intersection-Intersection ->   10 |          6054 |          3525
intersection-Intersection ->   11 |          6644 |          3885
intersection-Intersection ->   12 |          7228 |          4233
intersection-Intersection ->   13 |          7813 |          4607
intersection-Intersection ->   14 |          8435 |          4973
intersection-Intersection ->   15 |          9047 |          5337
intersection-Intersection ->   16 |          9656 |          5691
intersection-Intersection ->   17 |         10255 |          6059
intersection-Intersection ->   18 |         10843 |          6428
intersection-Intersection ->   19 |         11448 |          6791
intersection-Intersection ->   20 |         12025 |          7140
intersection-Intersection ->   21 |         12617 |          7499
intersection-Intersection ->   22 |         13225 |          7867
intersection-Intersection ->   23 |         13807 |          8223
intersection-Intersection ->   24 |         14400 |          8580
intersection-Intersection ->
intersection-Intersection -> ------------------------------------------
intersection-Intersection -> ---------- SIMULATION STATISTICS ----------
intersection-Intersection -> ------------------------------------------
intersection-Intersection -> Traffic Light | Total Cars | Largest Backup
intersection-Intersection -> --------------+------------+--------------
intersection-Intersection ->             0 |      14400 |            18
intersection-Intersection ->             1 |       8581 |             7
intersection-Intersection -> Destroying Intersection...
car_generator-Car Generator 1 -> Destroying Car Generator 1...
car_generator-Car Generator 0 -> Destroying Car Generator 0...
traffic_light-Traffic (SystemC) -> Destroying Traffic (SystemC)...
traffic_light-Traffic (PyRTL) -> Destroying Traffic (PyRTL)...
Simulation is complete, simulated time: 86.4 Ks
```

Listing 1: Sample Simulation Output

5 Extensibility

5.1 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.

5.2 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.