Ira A. Fulton School of Engineering

CSE 593 Applied Project

DEVS Hardware In The Loop (HIL) Mixed Mode Simulation with Bi-Directional Support

Thomas Jackson

thomas.e.jackson@asu.edu

CSE 593

Modeling Simulation and Application

School of Computing, Informatics, and Decision Systems Engineering

Arizona State University, Tempe, AZ, USA

Supervised by

Professor Hessam Sarjoughian

List of Figures

DEVS HIL MIXED MODE SIMULATION

Abstract

This paper presents the hardware-in-the-loop (HIL) mixed mode simulation research using DEVS-Suite simulator that supports parallel DEVS formalisms with visual experimentation design and behavior modeling[2,3]. This paper proposes a DEVS Simulation Hardware Abstraction Layer (DSHAL) with hardware communication API combined with Commercial Off-The-Shelf (COTS) hardware to provide bi-directional communication for HIL mixed mode simulation. Phase 1 of this research utilized COTS 4 and 8 port Relay 10Amp boards.

The mixed mode HIL simulation results show that this research can be extended to control Motors, Servo Controllers, Servo Motors, DC Controllers, DC Motors, Stepper Controllers, Stepper Motors or Sensors within DEVS-Suite simulator with full bi-directional communication.

1. Introduction

This project presents an innovative approach to augment existing DEVS analytical simulations. There is a clear distinction from Distributed Interactive Simulation (DIS) and Distributed Virtual Environments (DVEs) in that the primary objective of this proposal is to define the mechanism for establishing a bi-directional communication for Parallel DEVS model types directly from DEVS-Suite for COTS hardware. Minsky stated that “A Model (M) for a system (S) and an Experiment (E) is anything to which E can be applied in order to answer questions about S”[1]. This research presents an innovative alternative to answer even more questions about “S” with the additional HIL mixed mode simulation analysis. The ability to understand more about the system allows us to better predict the performance and behavior of these systems under varying input and circumstances. Additionally, using DEVS analytical simulations we can study the system behavior before it is actually built and possibly communicate a better system design. “What-if” questions can also be asked about these systems allowing us to have a deeper understanding of the behavior of these systems. Often these are real-world parallel or distributed systems that are too costly or dangerous to experiment with in order to answer these types of questions. This paper will discuss the analysis, design, implementation, and testing of this research project. Starting with the motivation for the HIL mixed mode simulation it will be shown how the project evolved and was refined to fit better into existing DEVS models and simulations.

1.1 Motivation for HIL Mixed Mode Simulation

The motivation for HIL mixed mode simulation began with the question, can the parallel DEVS formalisms or coupled models be also used to express interaction to cyber physical systems or COTS hardware? In other words, can we establish a method for driving an external physical device via one of the elements in the set of the output port and values in the DEVS formalisms? This area of research has a wide range of potential applications for many companies ranging from aerospace and defense, consumer electronics to networking. The commonality among these applications is that all of them can benefit from the ability to conduct mixed mode simulation in addition to the existing DEVS analytical simulations.

2. Analysis

The analysis of this project presents the analysis, in three parts, Initial Feasibility Test Report, Execution Time Requirements (early stages) and Kernel Libraries and Source Investigation that were taken before the initial design stage. Although Taxonomy I, II, and III are important model types for analysis, this project focuses primarily on Taxonomy IV Model Types. Of particular interest in this project are the I/O observation data points, in other words the trajectories of the inputs and outputs that are observed over a specific period of time and the coupled system trajectories based on interacting I/O systems. In our case, the interaction of the hardware and simulation coupling.

2.1 Initial Feasibility Test Report

The initial feasibility test was a check to see if communication in parallel to power an external device was even feasible directly from DEVS-Suite simulator while running a parallel DEVS model simulation. For the initial feasibility test a simple digital output relay I/O board was used as a basic building block with the following specifications:

- 4 Relay Outputs for switching AC or DC power
- Rated at 250VAC, 10 Amps or 100VDC, 5 Amps
- Connects directly to a computer’s USB 2.0 port

The I/O board was connected via USB and would allow to digitally control 4 SPDT (Single Pole Double Throw) relay outputs. This was an important initial test because the relay provided a mechanism to interface and control various higher-voltage devices. Figure 1 below is an image of COTS hardware by Phidgets[6].

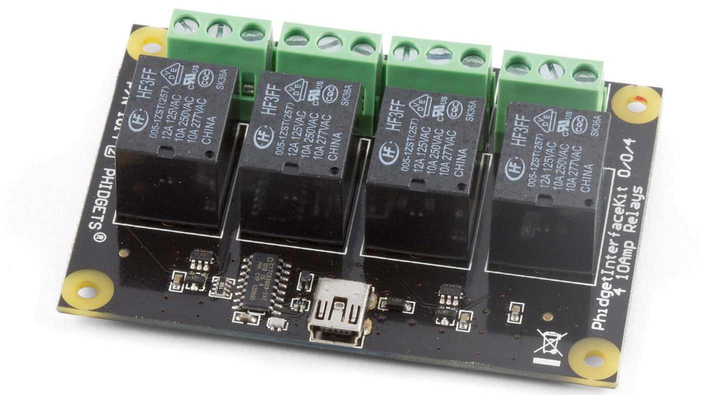
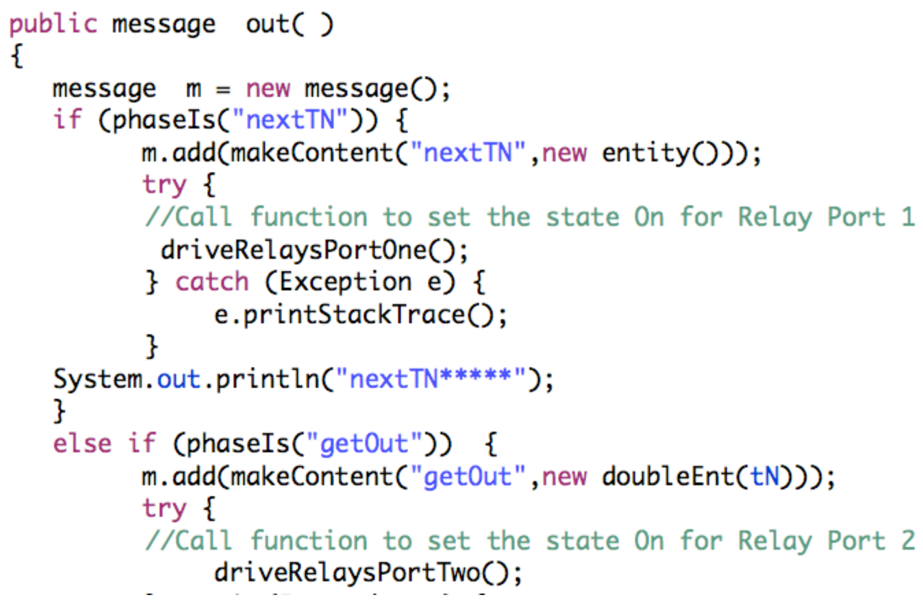


Figure 1. PhidgetInterfaceKit 0/0/4 4 port digital relay

The actual feasibility test was to see if it was possible to drive external power from the output function within a parallel DEVS model in DEVS-Suite. The feasibility test would pass if there was the ability to conditionally control each of the relay ports to set state for power on/off to an LEDS on any of the four digital 10 Amp relays while running a parallel DEVS model simulation. Using the existing COTS USB drivers and open source libraries simple methods were developed to communicate directly to the digital relays. At this stage there was no need for abstraction layers or advanced API since this was a feasibility test of the hardware and open source libraries. At this stage there was also no bi-direction communication. If the feasibility tests passed, the design would include abstraction layers and API calls for further refinement.

Figure 2 below, shows the placement of two function calls driveRelayPortsOne() and driveRelayPortsTwo() within the model's lamda output function to drive the outputs of relays port 1 and 2 respectively.



```
public message  out( )
{
   message  m = new message();
   if (phaseIs("nextTN")) {
         m.add(makeContent("nextTN",new entity()));
         try {
         //Call function to set the state On for Relay Port 1
          driveRelaysPortOne();
         } catch (Exception e) {
             e.printStackTrace();
         }
   System.out.println("nextTN*****");
   }
   else if (phaseIs("getOut"))  {
         m.add(makeContent("getOut",new doubleEnt(tN)));
         try {
         //Call function to set the state On for Relay Port 2
             driveRelaysPortTwo();
```

Figure 2. DEVS-Suite code example of output

The feasibility tests passed: the relays could be controlled independently while performing DEVS model simulation in parallel. Table 1. shows the results of the feasibility tests where conditional and independent control of the relays were validated before going to the next level in the research.

Feasibility Test	Pass/ Fail
Independently control relay port# 1	Pass
Independently control relay port# 2	Pass
Independently control relay port# 3	Pass
Independently control relay port# 4	Pass
Turn off all ports at once	Pass
Turn on all ports at once	Pass

Table 1. Feasibility Test Report

2.2 Execution Time Requirements (early stage)

This project uses “As Fast As Possible Execution Times” but can also be extended for real-time. The primary consideration for using “As Fast As Possible Execution Times” is that the target is cyber physical systems (CPS) and COTS hardware where mixed mode simulation can either be real-time or as fast as possible execution. In Stage 1of this research we are targeting CPS that do not have real-time constraints initially and are applying as fast as possible execution times. For example, the relay boards in this project use As Fast As Possible Execution Time. The entire process from receiving the relay's input to controlling the load is in the order of tens of milliseconds. Real Time execution requires further work in this area. Figure 3 illustrates the execution time at the hardware level for this project.

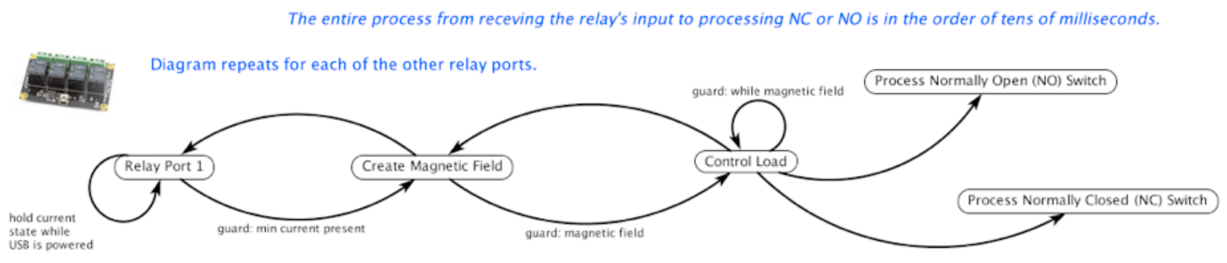

The entire process from receving the relay's input to processing NC or NO is in the order of tens of milliseconds.
Diagram repeats for each of the other relay ports.
Relay Port 1
Create Magnetic Field
Control Load
Process Normally Open (NO) Switch
Process Normally Closed (NC) Switch
guard: while magnetic field
guard: min current present
guard: magnetic field
hold current state while USB is powered

Figure 3. Execution Timing - Relay Board

2.3 Kernel Libraries and Source Investigation

The open source libraries and drivers were reviewed for gap analysis to provide a better understanding of what were the necessary modules and additions in order to satisfy the requirements of HIL mixed mode with bi-directional support. From the review it was clear that the reference board drivers are all based on Universal Serial Bus (USB) and the only pre-requisite packages for 32-bit and 64-bit Windows, Mac OS X and Linux 2.6.30 kernel were POSIX threads and USB 2.0 development header files such as "libusb”. For example, in most Linux distributions libusb-1.0-0-dev is already installed because libusb is widely used for many kernel drivers. The flexibility to customize the COTS drivers and libraries to our specific needs was an important consideration during this stage. Since DEVS-Suite uses Java as the programming language, we have ability to create new libraries written in Java as a JNI interface.

3. Design

In principle, the design had to provide the necessary abstractions for the various types of COTS hardware by Phidgets without impacting the performance and execution of the DEVS model simulations. Additionally, the goal was to provide the notion of a hardware port instead of directly referring to a relay, servo motor, etc.

In the initial feasibility test the model had references to a relay port. However, the goal was to create a higher level of abstraction so other COTS hardware can be used for HIL mixed mode simulation. For Stage 1 of this project, digital relays were used but the research is not limited to digital relays. The objective was to create a design approach that can be used for various types of COTS hardware without the DEVS model having to know anything about the device. This design approach fits perfectly within DEVS-Suite and Parallel DEVS coupled models.

$$\mathrm{EIC} = \{((\mathrm{N}, \text{“in”}), (p_0, \text{“in”})\}$$
$$\mathrm{EOC} = \{((p_2, \text{”out”}), (\mathrm{N}, \text{”out”})\}$$
$$\mathrm{IC} = \{((P_0, \text{“out”}), (P_2, \text{“in”}))\}$$

where:

$$X = \{(\mathrm{p},\mathrm{v}) \mid p \in \mathrm{IPorts}, \mathrm{v} \in X_p\}$$
$$Y = \{(\mathrm{p},\mathrm{v}) \mid p \in \mathrm{OPorts}, \mathrm{v} \in Y_p\}$$

D is the set of component names, includes the specific hardware component name

$\forall\, d \in D$

$M_d = \langle X_d, Y_d, S, \delta_{\mathrm{ext}}, \delta_{\mathrm{int}}, \lambda, ta \rangle$ is a DEVS model

$$X_d = \{(\mathrm{p},\mathrm{v}) \mid p \in \mathrm{IPorts}_d, \mathrm{v} \in X_p\}$$
$$Y_d = \{(\mathrm{p},\mathrm{v}) \mid p \in \mathrm{OPorts}_d, \mathrm{v} \in Y_p\}$$

Figure 4. Coupled Model DEVS formalism

Since a coupled model is a composition of models and by coupling together output ports of one model to input ports of another, outputs are transmitted as inputs and acted upon by the receiving model[3].

3.1 Definition of DSHAL

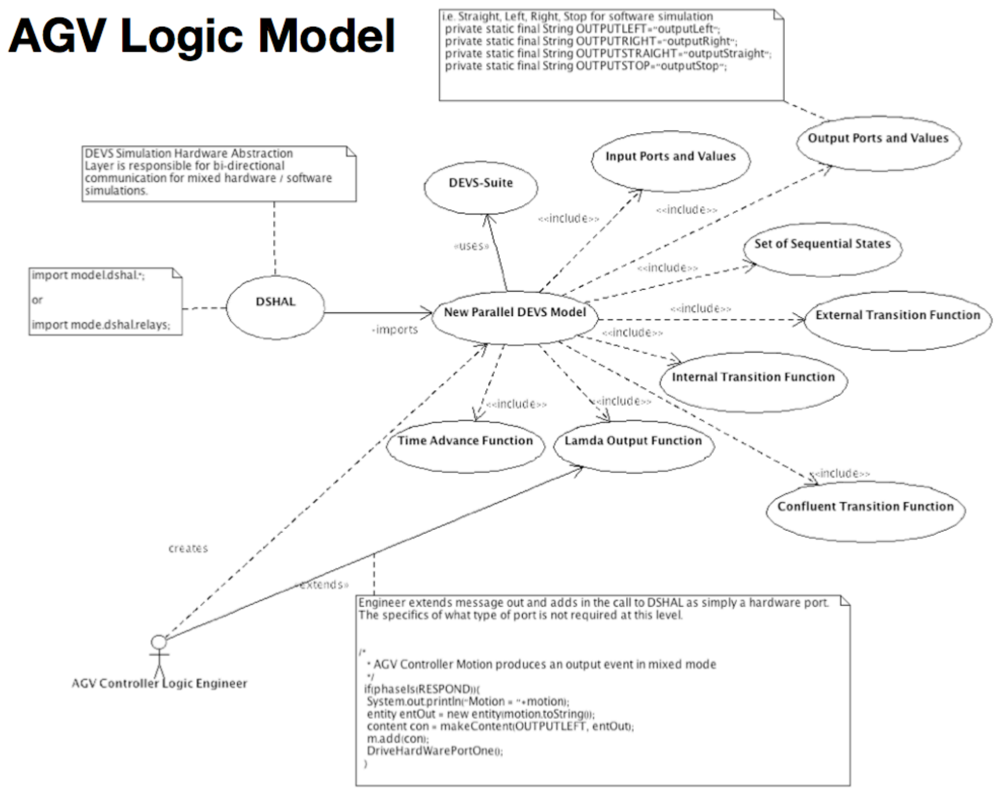
DEVS Simulation Hardware Abstraction Layer was defined to create the necessary abstractions in order for the DEVS models to communicate with real hardware. With these abstractions DEVS-Suite can be used for mixed mode HIL (hardware-in-loop) or RCP (Rapid control Prototyping). DSHAL also brings real-time execution capability of DEVS-Suite to real world use which is a key component for supporting all forms of HIL scenarios for future work. For example, allowing DEVS-Suite to communicate with the real world hardware increases the types of DEVS analytical simulations we can do to answer more questions about the system. The main functions of the DSHAL abstraction are listed below.

•Serves as standard interface for COTS hardware communication
•Acts as a proxy server for both sides
•Hides various hardware types from the simulator by providing a common interface

When the DSHAL was defined a basic assumption was made that calls to the outside for HIL capability incur a non zero time advance. The primary focus of the DSHAL is on parallel DEVS in order to allow for coupling of other models.

3.2 HIL Mixed Mode Use-Cases

To better understand the HIL mixed mode DEVS hardware / software simulations, specific use case examples of a simplified automated guidance vehicle (AGV) controller similar to that used in autonomous robots were studied. The AGV controller is an ideal use case for the HIL mixed mode simulation because at the low level controller it is heavily depending on time based inputs into the model for its guidance. The use case for the AGV highlights the use of four operating modes for guidance; straight, left, right and stop. Even with only four operating modes we can demonstrate a mixed mode for DEVS hardware / software simulations. The example uses the normal parallel DEVS input and output ports for the simulation and a separate DSHAL port for interaction.


AGV Logic Model
DEVS Simulation Hardware Abstraction Layer is responsible for bi-directional communication for mixed hardware / software simulations.
DSHAL
«imports
New Parallel DEVS Model
DEVS-Suite
«uses»
Input Ports and Values
Output Ports and Values
Set of Sequential States
External Transition Function
Internal Transition Function
Time Advance Function
Lamda Output Function
Confluent Transition Function
<<include>>
creates
«extends»
AGV Controller Logic Engineer

```
import model.dshal.*;
or
import mode.dshal.relays;
```

```
i.e. Straight, Left, Right, Stop for software simulation
 private static final String OUTPUTLEFT="outputLeft";
 private static final String OUTPUTRIGHT="outputRight";
 private static final String OUTPUTSTRAIGHT="outputStraight";
 private static final String OUTPUTSTOP="outputStop";
```

Engineer extends message out and adds in the call to DSHAL as simply a hardware port.
The specifics of what type of port is not required at this level.

```
/*
  * AGV Controller Motion produces an output event in mixed mode
  */
 if(phaseIs(RESPOND)){
  System.out.println("Motion = "+motion);
  entity entOut = new entity(motion.toString());
  content con = makeContent(OUTPUTLEFT, entOut);
  m.add(con);
  DriveHardWarePortOne();
 }
```

Figure 5. AGV Logic Model Use Case

Since in Stage 1 digital relays are proposed, the example use case above shows how we can use electrical stimuli to each of the abstracted hardware ports with the use of the DSHAL imported into DEVS-Suite. In this example, the abstracted ports are relays, but could be another type of COTS hardware. Using an AGV controller attached to DEVS Suite while running the parallel DEVS model allows us to study the exact behavior of the controller before deploying it in a production environment. In the example below, an AGV domain expert can verify the correctness of the model using the HIL mixed mode simulation.

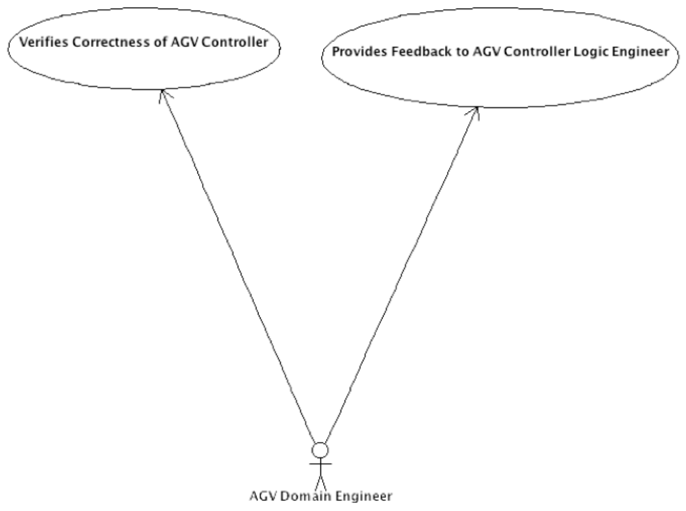

Verifies Correctness of AGV Controller
Provides Feedback to AGV Controller Logic Engineer
AGV Domain Engineer

Figure 6. AGV Domain Engineer

3.3 DSHAL Communication Layers

The communication use cases for DSHAL are defined in three different layers that aim at different objectives for separating simulation from hardware communication to provide HIL mixed mode simulation. Figure 7 below highlights the three distinct layers. Layer 1 is the overall DSHAL layer which is the standard interface for all hardware communication and also acts as a proxy for both sides. Layer 2 is the DEVSPhidget API communication layer with Phidgets API developed specifically for DEVS-Suite. Although Layer 4 Connects higher layers (DSHAL) to Phidget hardware drivers this layer can be easily extended to handle other COTS hardware support. For example, in addition to supporting Phidget boards we can also support various COTS sensors in a HIL mixed mode simulation environment. This is an important factor in the design of communication layers because COTS is widely supported in many sectors including Aerospace and Defense. Layer 3 is the hardware and driver layer which for this research is using open source libraries for the hardware drivers. Each of these layers will be described in further detail in the sections below.

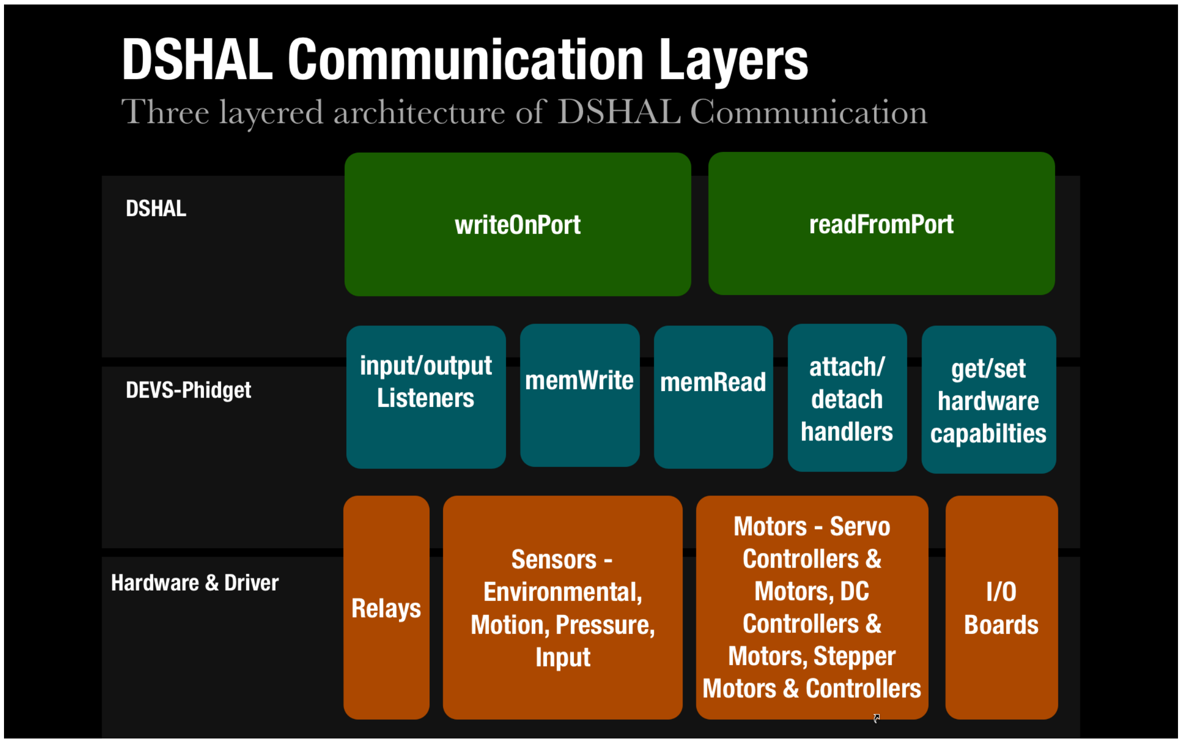

DSHAL Communication Layers
Three layered architecture of DSHAL Communication
DSHAL
writeOnPort
readFromPort
DEVS-Phidget
input/output Listeners
memWrite
memRead
attach/ detach handlers
get/set hardware capabilties
Hardware & Driver
Relays
Sensors - Environmental, Motion, Pressure, Input
Motors - Servo Controllers & Motors, DC Controllers & Motors, Stepper Motors & Controllers
I/O Boards

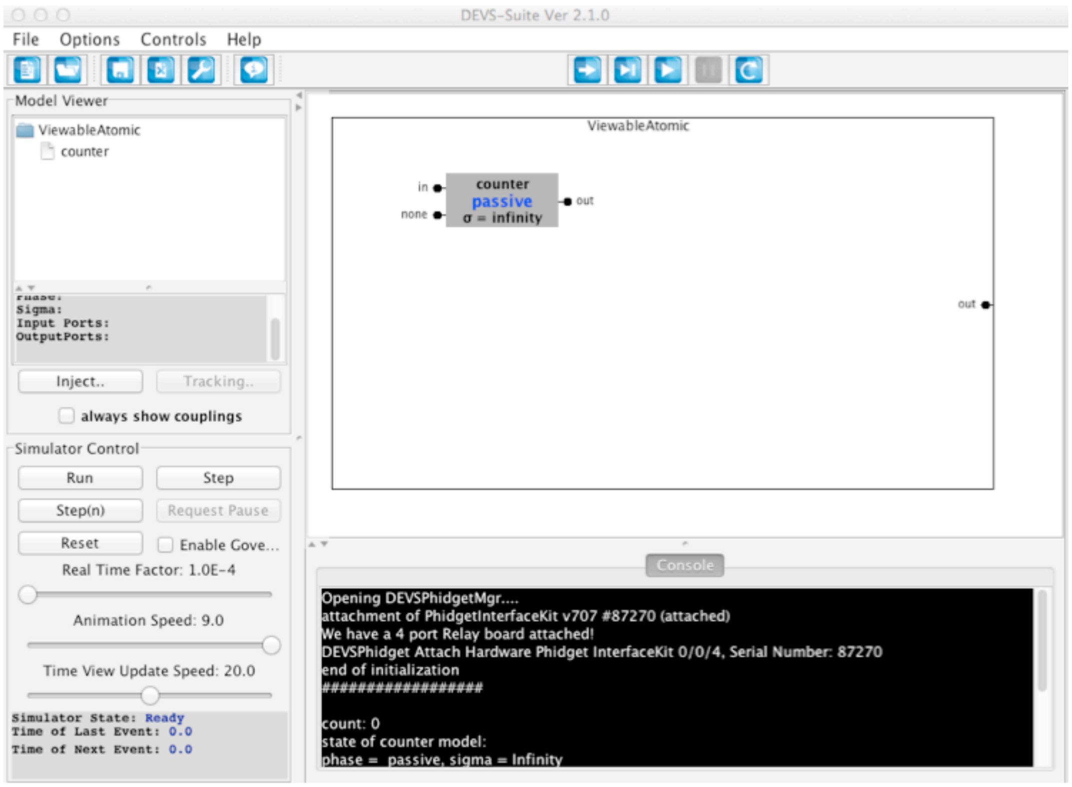
Figure 7. Communication Layers

3.4 Hardware Communication API

The design philosophy for the hardware communication API is that regardless of the hardware type or API the DEVS models should remain unchanged. In other words, the inclusion of HIL mixed mode should not affect our models or their realization in DEVS-Suite. Different hardware types can be used for mixed mode simulation as long as a kernel driver and API exists similar to DEVSPhidget described in this paper. Since closure under coupling and the overall hierarchical construction form the basis of the DEVS composition framework, the hardware communication API was designed that the internal models do not directly call send/receive functions of the hardware API.

3.5 Definition of DEVSPhidget API for DEVS-Suite

The DEVSPhidget API is the level responsible for handling the events as well as reading and writing into the hardware driver specified memory port. The DEVSPhidget API includes a DEVSPhidget manager as a way to keep track of attached devices. The manager will send attach and detach events as devices are added or removed from the system via USB. This was an important consideration in the overall design because the attach and detach handlers are multi-threaded. All of these events are handled in its own process thread so it does not affect DEVS-Suite interaction. A hardware can be attached or detached from the system at any time. This degree of separation allows for rapid control prototyping. The screen capture below illustrates how the DEVSPhidget API announces the attachment of a HIL device and already knows the capabilities. In this case, DEVSPhidget was aware that the new device attached is a 4 port digital relay.



```
Phase:
Sigma:
Input Ports:
OutputPorts:
```

```
Simulator State: Ready
Time of Last Event: 0.0
Time of Next Event: 0.0
```

```
Opening DEVSPhidgetMgr....
attachment of PhidgetInterfaceKit v707 #87270 (attached)
We have a 4 port Relay board attached!
DEVSPhidget Attach Hardware Phidget InterfaceKit 0/0/4, Serial Number: 87270
end of initialization
##################

count: 0
state of counter model:
phase =  passive, sigma = Infinity
```

Figure 8. DEVSPhidget USB Attach/Detach

The manager also retrieves all pertinent hardware information including device type and capabilities. Depending upon the type of hardware that is attached the HIL capabilities are updated accordingly. For example, attaching a 4 port digital relay initializes the ports array to hold 4 ports with the default value of and sets the output state to false. Attaching an 8 port digital relay device initializes the ports array to hold 8 ports.

3.6 DevsPhidget Bi-Directional Support

In designing the hardware communication API we developed a communication mechanism that does not assume that the hardware processing is reliable. That is to say, for every input event sent to the hardware we cannot assume it is processed without fault. For that reason we established bi-directional communication. Output change event listeners were added for complete bi-directional communication within the DEVSPhidget API. The output change listeners have the ability to provide a callback or notification to the coupled model described earlier in this paper. The output listeners also captures the current state, new state and hardware port associated with the event. With this support we know when any of the hardware ports have changed its status. The example below was the first prototype of bi-directional support that simply printed hardware state information to the console through the function "outputChanged". This function is called whenever an output for a device has changed. The DEVSPhidget API would instantiate and add OutputChangeListeners automatically in its constructor so there is no need to manually call this function.

```
outputListener = new OutputChangeListener() {
public void outputChanged(OutputChangeEvent outputChangeEvent) {

System.out.println("DEVSPhidget Output changed - Send Event to DEVS-Suite HWPort: "
                   +outputChangeEvent.getIndex() +" New State "
                   +outputChangeEvent.getState());

                   activePort = outputChangeEvent.getIndex();
                   newState = outputChangeEvent.getState();
            }
        };
```

Figure 9. Output change function - first prototype

The output change events listeners have been refined to handle non trivial communication and communication to the coupled model to drive simulation.

This additional support increased the potential range of applications for this research. By adding bi-directional support we are not only able to control the hardware from the DEVS model simulations, but we can take it even further to enable the hardware to control the models and simulation. This bi-directional support provides us with new trajectory data points that were not possible before. This is a new area of research that has a wide range of application. This feature also allows us to fine tune the research and choose real world systems with bi-directional communication that will be discussed later in this paper.

3.7 Layers Specification

The five communication layers as described in the earlier section are something that was proposed for this applied project. Below is an illustration that specifies each layer of DEVS/Phidget. However, implementing all of the communication layers is something that is not feasible in the short term allotted to this research. The amount of work that has to be done in the area of layers specification is quite substantial and will be left for future work.

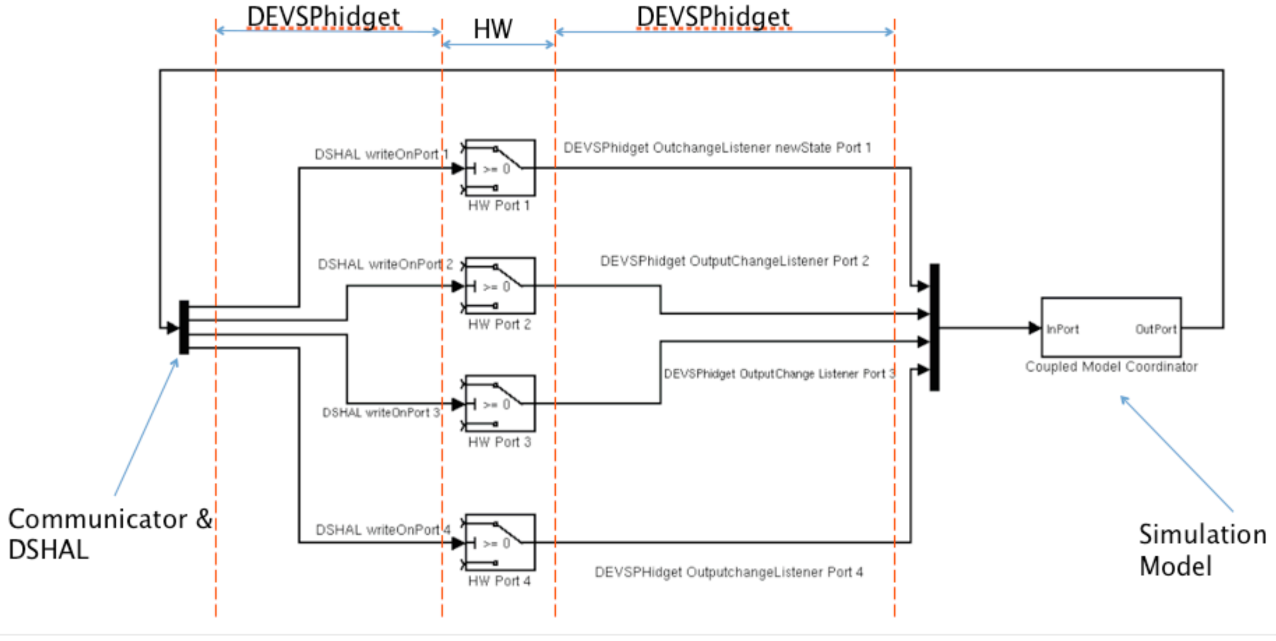


Figure 10. Layers Specification

3.8 Finite-State Machines

The discrete dynamics of the HIL mixed mode components were studied with finite-state machines in order to understand how each reaction maps valuations of the input valuations to output valuations[5]. The first set FSM studied were the asynchronous side-by-side composition of the Phidgets state machines .

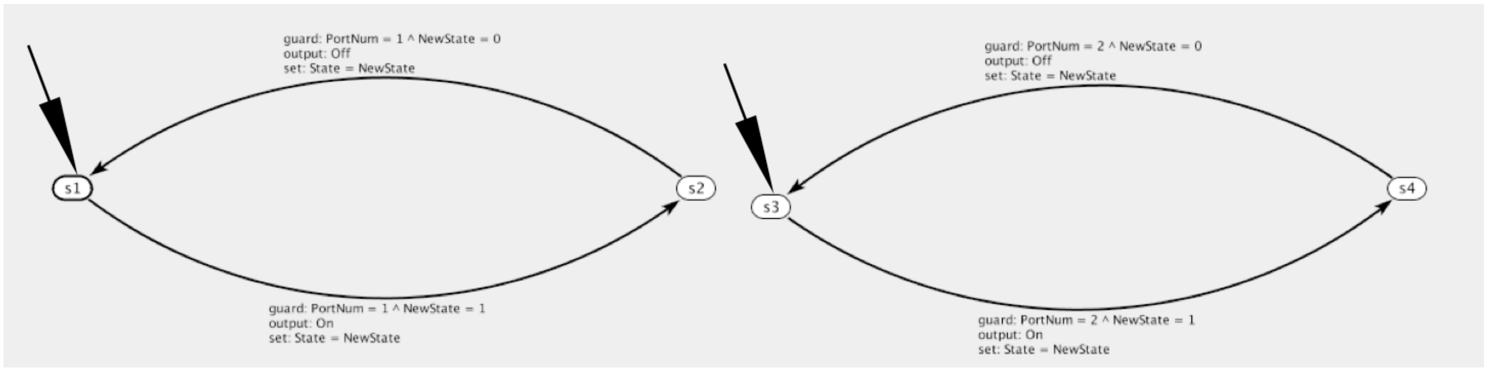


Figure 11. Side-by-Side Ports 1 and 2

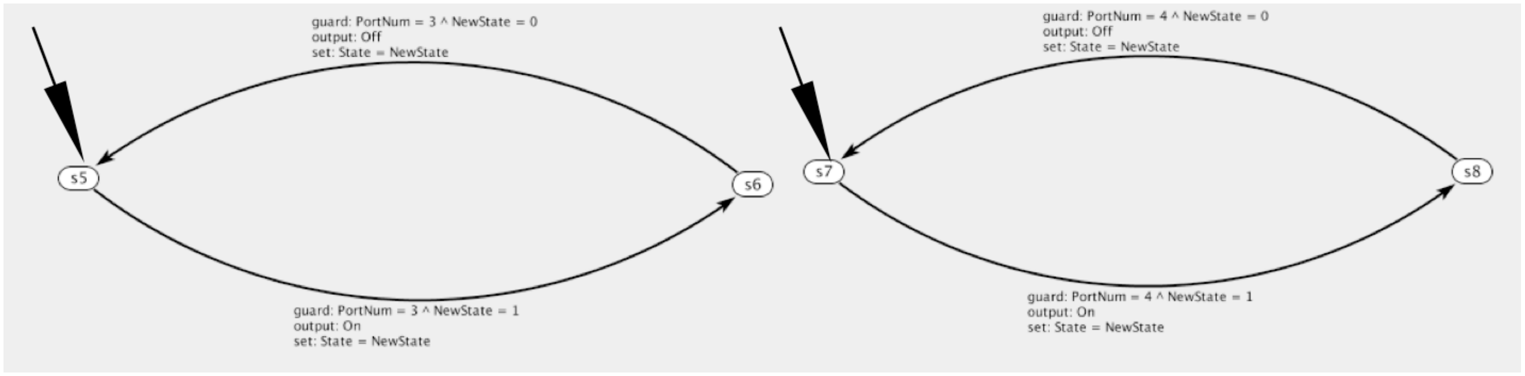


Figure 12. Side-by-Side Ports 3 and 4

The composition is based on four asynchronous actors in parallel for each of the hardware ports. The next FSM studied the asynchronous processing of events.

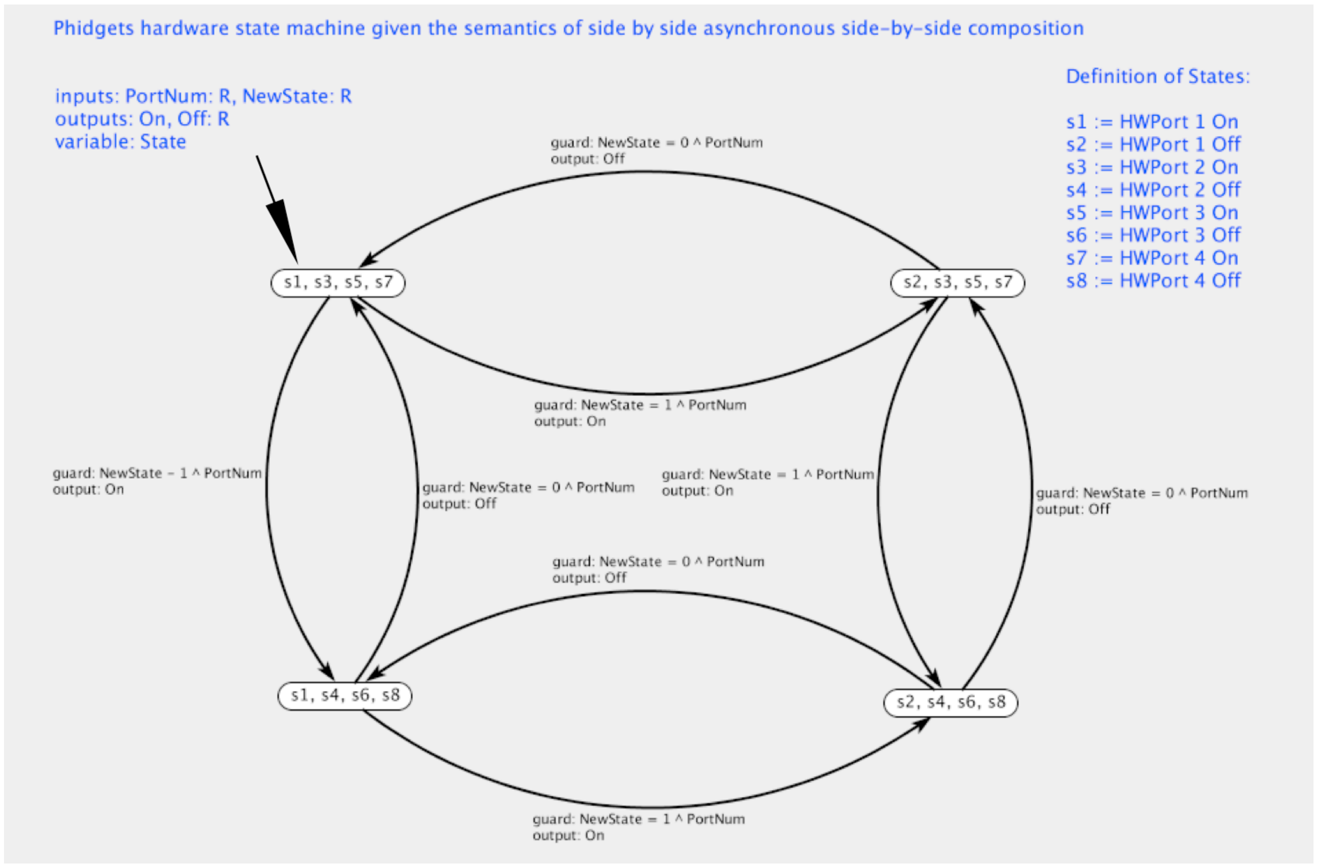
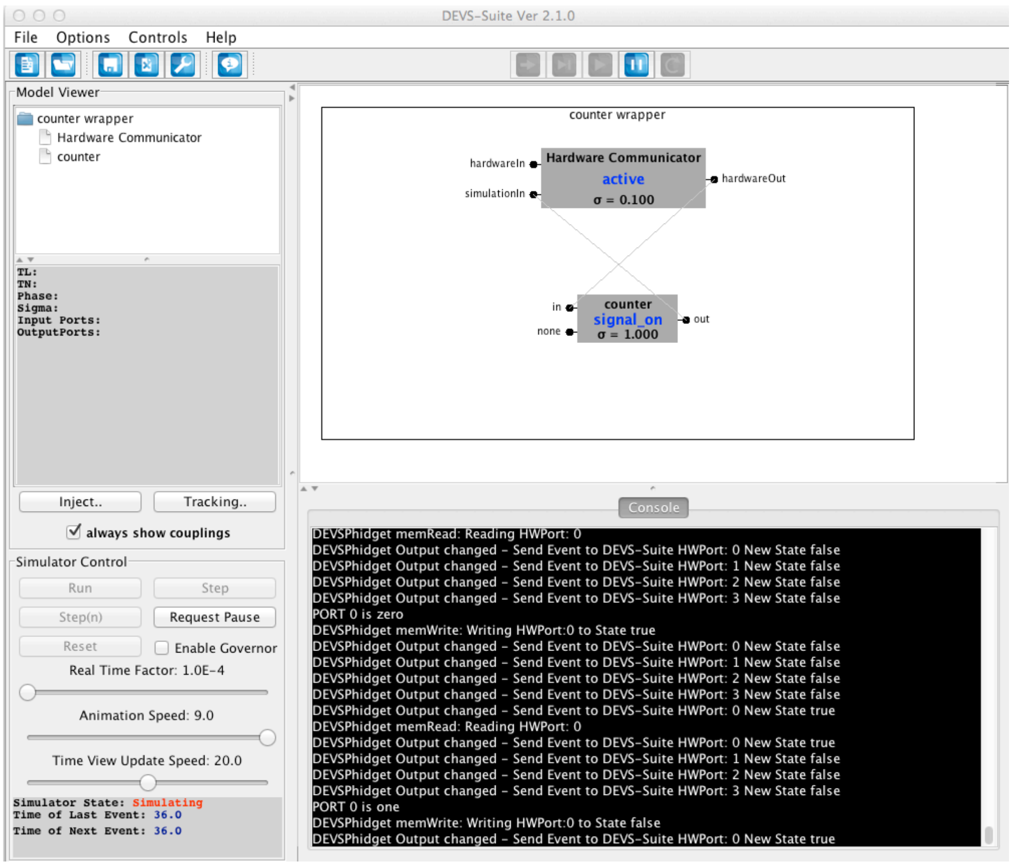

Phidgets hardware state machine given the semantics of side by side asynchronous side-by-side composition
inputs: PortNum: R, NewState: R
outputs: On, Off: R
variable: State
Definition of States:
s1 := HWPort 1 On
s2 := HWPort 1 Off
s3 := HWPort 2 On
s4 := HWPort 2 Off
s5 := HWPort 3 On
s6 := HWPort 3 Off
s7 := HWPort 4 On
s8 := HWPort 4 Off
s1, s3, s5, s7
s2, s3, s5, s7
s1, s4, s6, s8
s2, s4, s6, s8
guard: NewState = 0 ^ PortNum output: Off
guard: NewState = 1 ^ PortNum output: On
guard: NewState = 1 ^ PortNum output: On
guard: NewState = 0 ^ PortNum output: Off
guard: NewState = 1 ^ PortNum output: On
guard: NewState = 0 ^ PortNum output: Off
guard: NewState = 0 ^ PortNum output: Off
guard: NewState = 1 ^ PortNum output: On

Figure 13. Side-by-Side Asynchronous Composition

4. Implementation

New packages were added into DEVS-Suite with separation from the existing modeling base classes under model.simulation.hardware and DEVSPhidget. The model.simulation.hardware package includes the Communicator and DSHAL classes. The DEVSPhidget package includes both DEVSPhidget and DEVSPhidgetMgr classes as described earlier in the paper. The primary reason for the naming convention and ordering of the packages was the separation of concerns. For the implementation a parallel DEVS coupled model was created to test the full interaction between Communicator and the hardware. Figure 14 illustrates the hardware simulation coupling and memory read and write interactions to control the hardware via the input and output ports that are embedded in the top level coupled model.



```
TL:
TN:
Phase:
Sigma:
Input Ports:
OutputPorts:
```

Real Time Factor: 1.0E-4

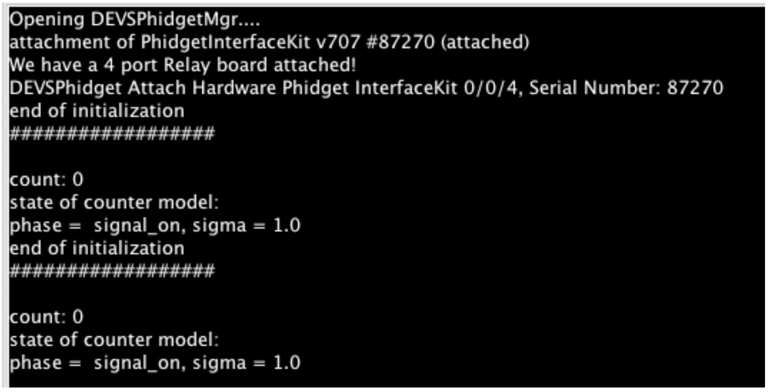
Animation Speed: 9.0

Time View Update Speed: 20.0

```
Simulator State: Simulating
Time of Last Event: 36.0
Time of Next Event: 36.0
```

```
DEVSPhidget memRead: Reading HWPort: 0
DEVSPhidget Output changed - Send Event to DEVS-Suite HWPort: 0 New State false
DEVSPhidget Output changed - Send Event to DEVS-Suite HWPort: 1 New State false
DEVSPhidget Output changed - Send Event to DEVS-Suite HWPort: 2 New State false
DEVSPhidget Output changed - Send Event to DEVS-Suite HWPort: 3 New State false
PORT 0 is zero
DEVSPhidget memWrite: Writing HWPort:0 to State true
DEVSPhidget Output changed - Send Event to DEVS-Suite HWPort: 0 New State false
DEVSPhidget Output changed - Send Event to DEVS-Suite HWPort: 1 New State false
DEVSPhidget Output changed - Send Event to DEVS-Suite HWPort: 2 New State false
DEVSPhidget Output changed - Send Event to DEVS-Suite HWPort: 3 New State false
DEVSPhidget Output changed - Send Event to DEVS-Suite HWPort: 0 New State true
DEVSPhidget memRead: Reading HWPort: 0
DEVSPhidget Output changed - Send Event to DEVS-Suite HWPort: 0 New State true
DEVSPhidget Output changed - Send Event to DEVS-Suite HWPort: 1 New State false
DEVSPhidget Output changed - Send Event to DEVS-Suite HWPort: 2 New State false
DEVSPhidget Output changed - Send Event to DEVS-Suite HWPort: 3 New State false
PORT 0 is one
DEVSPhidget memWrite: Writing HWPort:0 to State false
DEVSPhidget Output changed - Send Event to DEVS-Suite HWPort: 0 New State true
```

Figure 14. HIL Mixed Mode Implementation Running



```
Opening DEVSPhidgetMgr....
attachment of PhidgetInterfaceKit v707 #87270 (attached)
We have a 4 port Relay board attached!
DEVSPhidget Attach Hardware Phidget InterfaceKit 0/0/4, Serial Number: 87270
end of initialization
##################

count: 0
state of counter model:
phase =  signal_on, sigma = 1.0
end of initialization
##################

count: 0
state of counter model:
phase =  signal_on, sigma = 1.0
```

Figure 15. DEVSPhidget Initialization

4.1 Prototype Using COTS - Phidgets 4 port Relay I/O boards

The initial prototype to use Phidgets 4 port digital relay port was enough to conduct various experiments and satisfy the mixed mode simulation requirements. The prototype consisted of a A PhidgetInterfaceKit 0/0/4 connected via USB, four separate LEDs, battery connectors and 9 Volt batteries connected to the Normally Open load on the digital relay connected to Mac OS X with DEVS-Suite Version 2.1.0. Normally Open was chosen for the load type because it most closely resembles the real world scenarios. Normally Open loads will only be powered when the relay coil is powered via USB. Normally Closed loads will remain powered as long as the relay coil is not powered via USB. This is a good demonstration because some devices are potentially dangerous when left on. By using separate colored LEDs it was easy to visualize which hardware port is being controlled. Figure 15 is an image of the initial prototype using LEDs connected to the Normally Open load on the digital relay.

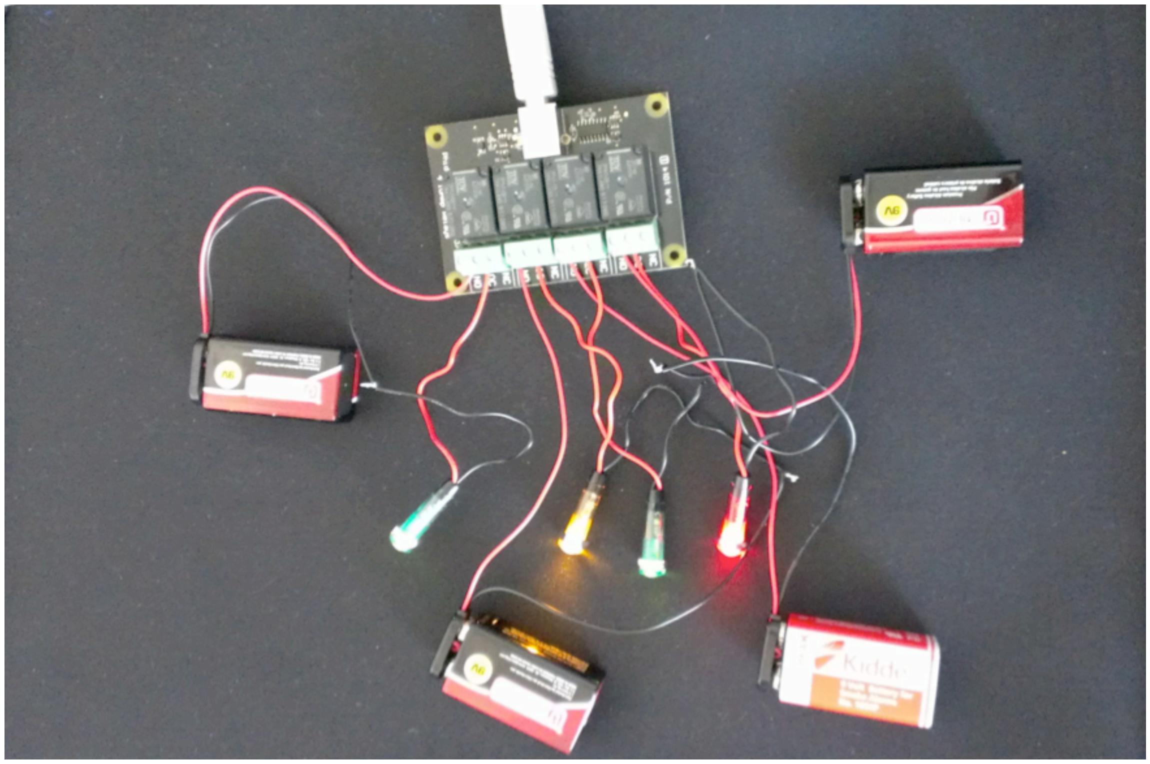


Figure 16. Early Prototype

5. Testing & Experiments

Practical tests and experiments were performed against the DEVS HIL Mixed Mode implementation. Real world problems were presented so that the issues van be better understood and solved using the approach in this research.

5.1 Real World Problem Examples

As a means of evaluating the DEVS HIL Mixed Mode implementation it was necessary to provide a world real problem example. Software Defined Networking (SDN) was chosen as the real world example which is an emerging new approach to networking which basically allows the decoupling of the control plane from the data plan in network switches and routers. SDN has attracted a lot of attention recently from academia, industry, and government mainly because it is an innovation that allows for the control and program of the network in a way to make it responsive to networking events in a more proactive fashion. Data flow tables can be monitored and controlled based on user-defined rules instead of fixed firmware defined rules. The new approach allows for traffic reshaping. For example, reconfiguring a network dynamically to enforce packet forwarding, blocking, redirection, reflection, MAC or IP address changing, limiting the packet flow rate are all potential use-cases of SDN. Instead of user-defined rules, the hardware itself created the SDN data flow rules. The SDN data flow rules are dynamically generated based on the events at the network switch or router. The SDN is a real world problem that can be understood better and solved with mixed mode simulation. This research allows for the testing on any scale, small or large to see if they work. The following state chart is an example to illustrate a possible data flow using bi-directional mixed mode simulation. In SDN, the data flow tables can be monitored and controlled. This example shows simple traffic reshaping for four states; {Normal_Network_Routing, Congested_Network_Routing, Prioritized_Network_Routing, Full_Networking_Routing} based on the information received from the OutputChangeListener event in the DEVSPhidget API. Based on the events coming back from the hardware we can iterate through the various states. This real world example highlights the capabilities of bi-directional mixed mode simulation to test the control data plane for SDN.

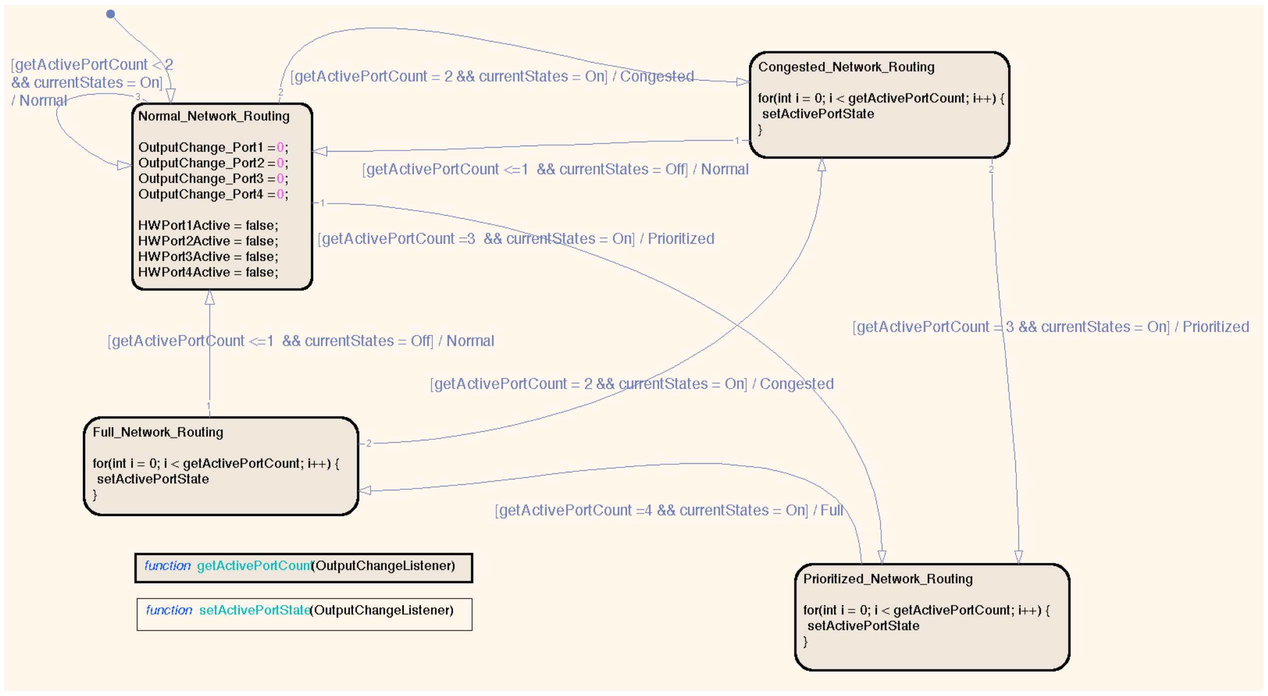

[getActivePortCount < 2 && currentStates = On] / Normal
[getActivePortCount = 2 && currentStates = On] / Congested
Congested_Network_Routing
for(int i = 0; i < getActivePortCount; i++) {
setActivePortState
}
Normal_Network_Routing
OutputChange_Port1 = 0;
OutputChange_Port2 = 0;
OutputChange_Port3 = 0;
OutputChange_Port4 = 0;
HWPort1Active = false;
HWPort2Active = false;
HWPort3Active = false;
HWPort4Active = false;
[getActivePortCount <=1 && currentStates = Off] / Normal
[getActivePortCount =3 && currentStates = On] / Prioritized
[getActivePortCount = 3 && currentStates = On] / Prioritized
[getActivePortCount <=1 && currentStates = Off] / Normal
[getActivePortCount = 2 && currentStates = On] / Congested
Full_Network_Routing
for(int i = 0; i < getActivePortCount; i++) {
setActivePortState
}
[getActivePortCount =4 && currentStates = On] / Full
function getActivePortCount(OutputChangeListener)
function setActivePortState(OutputChangeListener)
Prioritized_Network_Routing
for(int i = 0; i < getActivePortCount; i++) {
setActivePortState
}

Figure 17. SDN State chart Example

The following is an illustration of the sample sequence of software and simulation events and responses during the operation of the SDN example.

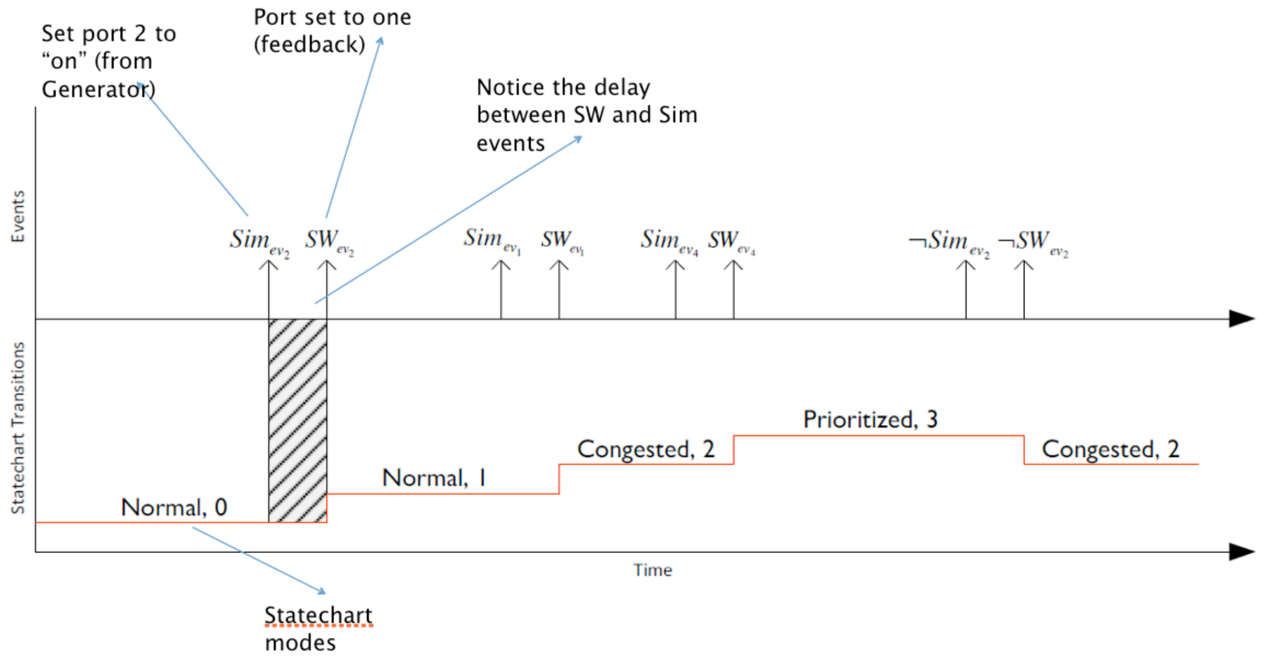

Set port 2 to "on" (from Generator)
Port set to one (feedback)
Notice the delay between SW and Sim events
Events
Sim_{ev_2} SW_{ev_2} Sim_{ev_1} SW_{ev_1} Sim_{ev_4} SW_{ev_4} $\neg Sim_{ev_2}$ $\neg SW_{ev_2}$
Statechart Transitions
Normal, 0
Normal, 1
Congested, 2
Prioritized, 3
Congested, 2
Time
Statechart modes

Figure 18. Sample Sequence of events - SDN

5.2 Simulation / Hardware Coupling

The bi-directional communication for the SDN example is illustrated in further detail in Figure 19 below.

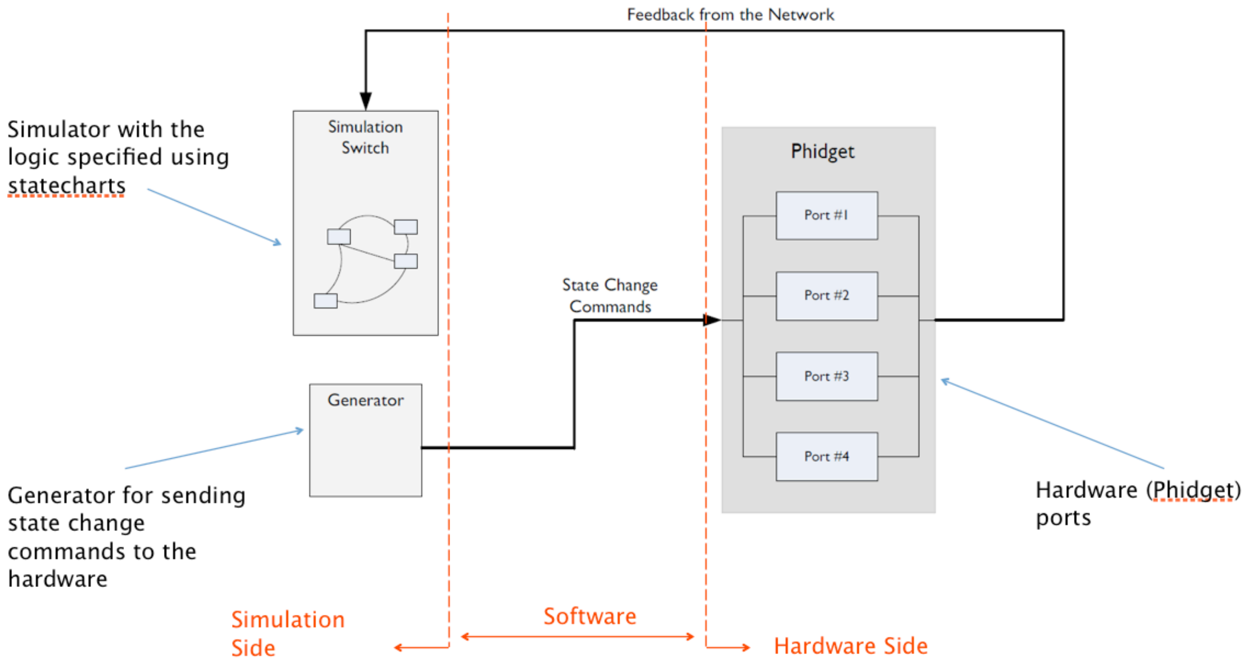


Figure 19. Bi-Directional Communication - SDN

5.3 Hardware Input Port Analysis

At the hardware level each of the relays ports are truly independent since the electro-mechanical relays are nothing more than a controllable switch and can receive current at ay time in parallel to any of the other ports. There is a clear distinction with the software level. At the software level each relay port is associated to a hardware device driver. For example, the 4 port board used in this research is associated to a Phidgets InterfaceKit 0/0/4 which comprises of four relay ports. There is no direct connection to a relay port without going through the device driver which creates a relationship to the other relay ports on the board. Each hardware device is independent of each other and within the device each relay port is considered asynchronous.

5.4 Software Tools

The software stack consisted of the Phidget21.framework library, Phidget.kext kernel extension, libphidget21.jnilib, DEVS-Phidget, DSHAL and DEVS-Suite Version 2.1.0. The Phidget.kext is unique to Mac OS X but equivalent kernel extensions are available for Windows and Linux operating systems. Phidget21.framework contains the actual Phidget C library, which is used at run-time. libphidget21.jnilib is the JNI library for Java. DEVS-Phidget and DSHAL which were described earlier provides the necessary hardware abstractions and support for two-way communication within DEVS-Suite.

5.5 Hardware

In this research 4 port SPDT (Single Pole Double Throw) digital output relays were used for switching AC or DC power. The relays are an electromagnetic switch consisting of a coil. If a minimum current is present on input it creates a magnetic field to energize the coil which then toggles the switch depending on how the load is connected, i.e. NO or NC. Digital relays were proposed for Stage 1 of the research because if proven to work relays can also switch various higher-voltage devices such as motors, servo controllers, servo motors, DC controllers, DC motors, stepper controllers, stepper motors or sensors. The minimum switching current of the relay in this experiment is 100mA @ 5VDC which is not ideal for switching signals. If signal switching is desired an I/O board can be easily be used in the design which includes both analog and digital inputs with digital outputs. However, DSHAL and DEVS-Phidget API proposed in this paper can accommodate this board as well.

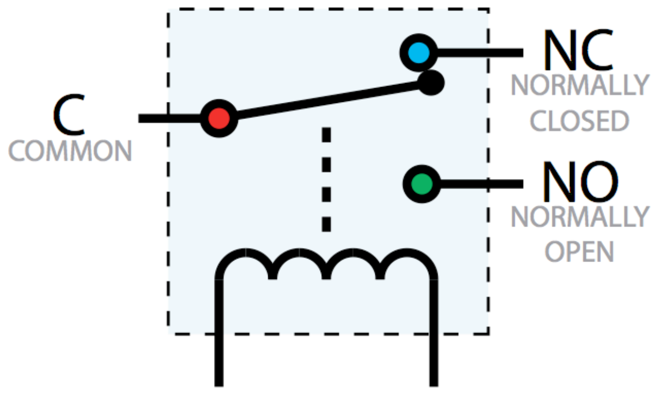


Figure 20. Simple SPDT Digital Relay Schematic

6. Conclusions / Future Work

We have shown in this paper how Parallel DEVS model types directly from DEVS-Suite can be combined with COTS hardware to create HIL Mixed Mode Simulation. We illustrated the use of the system based on the real world problem SDN use case and studied the possibility of a self-reconfiguring network routing made possible with bi-directional HIL support. Our goal is to extend the DSHAL Communication Layers as described earlier in the Design section of this paper to include more work around the Communicator Layer and Outer Coupled Model to allow for non-trivial communication. For example, the next step in improving the system is to add communication event handlers. This improvement will allow us to perform experiments with various types of communication events for Parallel DEVS models. The second area of future work is to extend the DEVSPhidget API to include more hardware types such as motors, servo controllers, servo motors, DC controllers, DC motors, stepper controllers, stepper motors or sensors that was described earlier in the Hardware Section. This research work is an interesting prospect for V&V for Simulation Models. With this research one has the foundation to prove that the simulation model and the original system are functionally identical.

References

[1] Minsky M., “Models, Minds, and Machines” Proceedings of IFIP Congress, pp. 45-49, 1965.

[2] Kim S., et al, H.S. “DEVS-Suite: A Simulator Supporting Visual Experimentation Design and Behavioral Monitoring”.Arizona Center for Integrative Modeling and Simulation-University of Arizona, Arizona State University (2009)

[3] Sarjoughian, H., Zeigler B., “Introduction to DEVS Modeling & Simulation with JAVATM:Developing Component-based Simulation Models”, Arizona Center for Integrative Modeling and Simulation- University of Arizona, Arizona State University (2003)

[4] Ziegler B.P., et al, “Theory of Modeling and Simulation” , “Integrating Discrete Event and Continous Complex Dynamic Systems” Second Edition 2000

[5] Ziegler B., Sarjoughian, H.S. “Introduction to DEVS Modeling & Simulation with JAVA: Developing Component-based Simulation Models”. 2003

[6] phidgets. “Products for USB Sensing and Control, 1014_2 - PhidgetInterfaceKit 0/0/4 Internet: http://www.phidgets.com/products.php?product_id=1014l, July, 2012 [Aug. 2, 2012].

[7] Sarjoughian, H., CSE561 Class Lecture, Topic: “A Real-Life V&V Example: NCES Collaboration Tools”, Arizona Center for Integrative Modeling and Simulation- University of Arizona, Arizona State University (Spring 2011)

Appendix I: Source Code - Sample Memory Read/Write

Example Memory Read/Write Functions

```
public boolean memRead(int hwport) throws PhidgetException {
       System.out.println("DEVSPhidget memRead: Reading HWPort: " + hwport);

       ik.openAny();
       ik.waitForAttachment();
       boolean state = ik.getOutputState(hwport);
       ik.close();

       return state;
}


public void memWrite(int hwport, boolean state) throws PhidgetException{

       System.out.println("DEVSPhidget memWrite: Writing HWPort:" + hwport + " to
       State " +state);

       ik.openAny();
       ik.waitForAttachment();
       ik.setOutputState(hwport, state);
       ik.close();

}
```

Appendix II: Source Code - Sample Constructor

Example DEVSPhidget Constructor

```
       public DEVSPhidget() throws PhidgetException
       {
              ik = new InterfaceKitPhidget();
           // ========== Event Handling Functions ==========
           attachHandler = new AttachListener() {
           public void attached(AttachEvent event) {

              deviceName = new String();

               try {
                   setSerialNo(((Phidget)event.getSource()).getSerialNumber());
                   setHardwareName(((Phidget)event.getSource()).getDeviceName());
                   setHILCapabilties();// Internal call to obtain the capabilities of the h/w
               } catch (PhidgetException exception) {
                   printError(exception.getErrorNumber(), exception.getDescription());
               }

               System.out.println("DEVSPhidget Attach Hardware " + getHardwareName() +
               ", Serial Number: " + getSerialNo());
           }
        };


        detachHandler = new DetachListener() {
           public void detached(DetachEvent event) {
               int serialNumber = 0;
               String name = new String();

               try {
                   serialNumber = ((Phidget)event.getSource()).getSerialNumber();
                   name = ((Phidget)event.getSource()).getDeviceName();
               } catch (PhidgetException exception) {
                   printError(exception.getErrorNumber(), exception.getDescription());
               }

               System.out.println("DEVSPhidget Detach Hardware " + name + ", Serial Number: " +
              Integer.toString(serialNumber));

           }
       };

       outputListener = new OutputChangeListener() {
           public void outputChanged(OutputChangeEvent outputChangeEvent) {

               System.out.println("DEVSPhidget Output changed - Send Event to DEVS-Suite HWPort:
              "  +outputChangeEvent.getIndex() +" New State "  +outputChangeEvent.getState());
               activePort = outputChangeEvent.getIndex();
               newState = outputChangeEvent.getState();

           }


       };

       inputListener = new InputChangeListener() {
           public void inputChanged(InputChangeEvent inputChangeEvent) {

               System.out.println("DEVSPhidget Input changed - Send Event to DEVS-Suite HWPort:
              "  +inputChangeEvent.getIndex() +" New State "  +inputChangeEvent.getState());
            }


       };


       // No exception thrown on create
       manager = new DEVSPhidgetMgr();

       manager.addAttachListener(attachHandler);
       manager.addDetachListener(detachHandler);
       ik.addOutputChangeListener(outputListener);
       ik.addInputChangeListener(inputListener);


       System.out.println("Opening DEVSPhidgetMgr....");
       try {
           manager.open();
       } catch (PhidgetException exception) {
           printError(exception.getErrorNumber(), exception.getDescription());
       }


   }
```