Adding a Java GUI to SystemC Simulation for Virtual Prototyping of Embedded Systems *

Hadipurnawan Satria
Sun Moon University
Department of Computer Sciences
Kalsan 100, Tangjeong, Asan, Chungnam
Republic of Korea

Jin B. Kwon
Sun Moon University
Department of Computer Sciences
Kalsan 100, Tangjeong, Asan, Chungnam
Republic of Korea

Abstract: Since a SystemC simulation program is a text-based program that uses files for its inputs and outputs, it is minimally interactive with users. To use the SystemC simulation core for virtual prototyping of embedded systems, a Graphical User Interface (GUI) front end for visualizing behaviors of systems and providing user interaction is required. We propose a method for adding such a GUI front end by implementing an API library, called SCJlib, to link the SystemC simulation program and a Java GUI application.

Key–Words: Virtual Prototyping, Simulation, SystemC, Embedded Systems

1 Introduction

Nowadays, embedded system products are found everywhere and are becoming more and more advanced. To keep up with the market competition, the products must be more sophisticated and feature rich, but manufacturers also require a shorter time to market.

Virtual prototyping is one way to speed up the development process and achieve a shorter time to market without reducing the complexity of the products. A virtual prototype can be defined as a computer-based simulation of an embedded system with a degree of functional realism similar to a physical prototype[4]. Virtual prototyping is the process of using a virtual prototype, instead of a physical prototype, to test and evaluate specific characteristics of a candidate design[3]. Virtual prototyping can be applied with different objectives and at different levels of the development process; for instance, when determining the feasibility of the product, establishing and validating the client's requirements, validating the functional specifications, and estimating the performance and the cost. Thus, by using virtual prototypes, developers can carry out the above functions on the desired system in the early stages of the development process without actual physical hardware. This results in a shorter development process.

Rapid prototyping is a form of collecting information on requirements and on the adequacy of possible designs. The prototype is eventually thrown away, although it is an important resource during product development. Incremental prototyping[2] enables large systems to be installed in phases to avoid delays between specification and delivery. After the customer and supplier agree on certain core features, the installation of a skeleton system occurs as soon as possible. Important requirements can be checked out in the field, enabling changes to core features while extra, less important, features can be added later. Reusable software and highly modular languages are often the most useful tools for piecing sections together[4]. In incremental prototyping[2], a prototype is developed through various steps from a very abstract model to a very detailed hardware-accurate model, leading to the real end-user application.

SystemC[5] is a system-level modeling language based on C++ that can be used to develop prototypes of embedded systems. One of its most important advantages is that SystemC provides multiple abstraction levels of modeling, unlike VHDL and Verilog[1, 6]. Instead, it simply adds class libraries, which are important for designing embedded systems. Because SystemC is an extension of C++ but does not add any new syntax to C++, it may be familiar to software developers. This is another main advantage in embedded system development, where software developers and hardware developers must work together[1, 6]. It has become quite popular in the embedded system community since it was introduced in 2002. A model written in SystemC can be compiled and then executed like an executable file, which is the

*This research is supported by university IT research center (ITRC) project funded by Korean Ministry of Information and Communication

virtual prototype of the model. With the help of the SystemC built-in simulation kernel, the behavior described by the model is simulated by running this executable. This simulation program, often called an executable specification, is a console text-based program and typically is not interactive. The program usually works on files: it reads its input from files and writes its output to files. The input files contain all the possible combinations of inputs, and the output files contain outputs corresponding to each input. Such input and output files are sometimes referred as wave files. The correctness of the model is validated by analyzing these files, which may be a very tedious task for the developers.

In this paper, we provide a method for adding an interactive GUI to a SystemC simulation program. This will make it possible to create a GUI model of an embedded system using SystemC, which can be analyzed, tested, and verified interactively. The GUI for SystemC simulation is a visible component, and it requires no changes to the core SystemC source code. The front end is integrated with the SystemC model by using a set of APIs designed and implemented in this work. The GUI is developed in Java; thus, the APIs are divided into two parts: one for SystemC and one for the Java GUI. The GUI application acts as the front end for the SystemC program. The SystemC simulation is connected to the environment through this GUI, instead of by reading and writing files. The SystemC simulation requests inputs from, and gives outputs to, the GUI, which in turn requests input from, and gives outputs to, the user. This graphical interaction reduces the development process, because it is easier to use and less prone to error.

2 System Architecture

The objective of this work is to provide a way of integrating two independent programs, the SystemC simulation program and the Java GUI program. The SystemC simulation is the core program, also called the back end, and the Java GUI acts as a front end. In order to create seamless integration between the two, there are two considerations. First, the GUI program should represent the running simulation graphically. Thus, the GUI should know every user-visible output from the simulation. These output data must be sent out from simulation program to the GUI continuously during the simulation. Second, users should be able to interact with the simulation via the GUI. Hence, every input received from users should be transferred to the simulation program continuously during the simulation. These inputs may cause the some reaction from the simulation, which then should be visualized in the GUI.

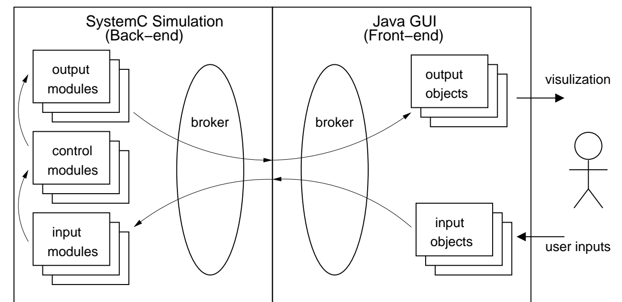


Figure 1: System architecture

Our proposed system architecture is shown in Figure 1. We classify SystemC modules into three types based on their user-interactivity: *input modules*, *output modules*, and *control (or computing) modules*. Control modules get data from input modules and put data to output modules. Since input modules and output modules should interact with users, they are visualized by *output objects* and *input object* in the GUI part, respectively. The *control modules* do not have any counterparts in the GUI part, because they never interact directly with users and the GUI does not do any computations except graphical visualization. The communication between the modules in back end part and their counterparts in the front end part are supervised by two *brokers* resident in each part.

Two scenarios are presented below to provide better understanding of the data transfer mechanism which happens between two parts. First, suppose an output module T in the back end generates data t(e.g., temperature) regularly at any rate. The data t then transferred to the broker in the back end, which encodes the data into a message packet with the unique identifier of the destination object T', i.e., the counterpart of the module T, in the front end. Then, the broker informs the broker in the front end of the packet and immediately sends the packet. The broker in the front end decodes the message packet to determine which object is the destination object from the identifier in the packet, and then passes t to the destination object T'. Finally, the corresponding output object displays its own graphical representation of data t. How the data represented graphically depends completely on the imagination of system developers and only limited by the capability of the Java GUI library in use. For instance, data can be represented by showing text messages, chaning colors, or even animating itself. Second, suppose an input visual object C receive an a data c (e.g., a coin of 50 cents) from a user. c then is sent out to the front-end broker, which encodes the data c into a message packet with the identifier of the destination input module C'in the back

end. Subsequently, the front-end broker cooperates with the back-end broker to transfer the packet. The back-end broker decodes the message packet and determines which input module should receive the data. Finally, the input module C' receives and processes the data c. How the user inputs are entered is also freely up to the system developers and the capability of the chosen GUI library. For example, the developers can use text boxes and buttons.

3 Broker-to-Broker Communication

The brokers communicate with each other by using *sockets*. A socket is a software endpoint that establishes bidirectional communication between a server program and one or more client programs. The socket associates the server program with a specific port on the machine where it runs, so that any client program anywhere in the network can communicate with the server program by connecting with socket to the specified port of the server.

The SystemC program and the GUI program are interoperated through the brokers, which communicate via sockets. Figure 2 shows the socket communication mechanism between the two brokers. Two separate sockets are used for communication. The first one is used to send output from the back-end broker to the front-end broker. The back-end broker sends data while the front-end broker listens for data. The second one is used for the opposite process: the front-end broker sends data while the back-end one listens for data. Each socket must be associated with a predetermined port, where the front-end broker always acts as the socket server while the counterpart is always the socket client.

Socket communication can be synchronous or asynchronous. With synchronous communication, the listening endpoint will block while waiting for data. In contrast, in asynchronous communication, the listening endpoint will not block and will continue executing with or without data. The implementation of synchronous communication is simpler, but the blocking may cause some problems. Java can utilize its multi-thread support and use a separate thread to listen to data synchronously, without blocking other activities. On the other hand, SystemC's own scheduling system does not allow for generic multi-thread support such as POSIX threads. As a result, the SystemC part must use the more complex asynchronous mechanism for listening to socket data.

Using socket communication allows the SystemC part and the Java part to be in separate machines connected through a local area network or even the Internet. However, the two predetermined ports must be available and open on both machines.

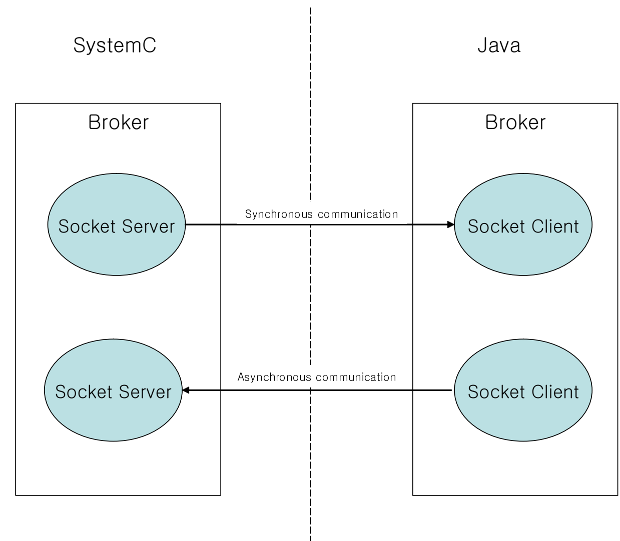


Figure 2: Socket Communication between Brokers

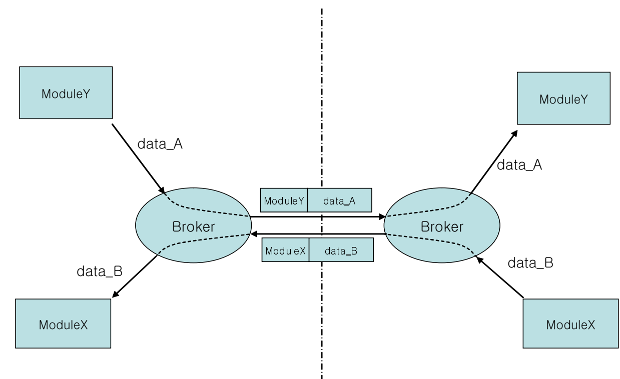


Figure 3: Data encoding

Each data element is encoded before being transmitted. The encoded data has two parts, the header and the body. The data header contains a module name. This is the name of the source module that sends the data, and it is also the name of the destination module that receives the data. The receiving broker uses this information to determine the module to which the data must be delivered. The body part contains the data itself as string of bytes. The receiving module is responsible for any type casting necessary. Figure 3 shows how the data transferred and being modified on the way.

The sockets are implemented for each side using the corresponding language, C++ for the SystemC part and Java for the GUI part. The C++ implementation is a class approach which use classes to encapsulate all socket functionalities from operating system. The operating system for SystemC machine is assumed to be POSIX compliant and the socket func-

tions used are low level POSIX System Calls. This can be accomplished in Linux or Windows with Cygwin[11] installed. On the other and, the java implementation is operating system independent, due to java portability. The java implementation is event based, using listener / adapter classes to listen and react for events, in this case event of data proceeding.

The SystemC part is the server side, so it must do these following steps in order to accept connection from the client. First, it must create a socket, using socket() system call. Then, it must bind the socket to an address using the bind() system call. For a server socket on the Internet, an address consists of a port number on the host machine. Subsequently, it can now listens for connection using the listen() system call. After it receipts a connection, it can accept that connection using the accept() system call. This call typically blocks until a client connection connects with the server. Finally theSystemC part can send and receive data. All these steps are encapsulated inside a class, so in point of fact the SystemC part only has to create the instance of the encapsulating class and automatically able to send and receive data, after a client connection received. The GUI part is the client side of socket connection, and utilizes java built-in Socket class. The main class is Socket class which is a built-in java class, and some more classes are implemented on top of Socket class to achieve desired functionalities. These classes enwrap socket class and provide events mechanism to notify if there is any data available[12]. The GUI program must listen to this event and get the data whenever it is available. On the other hand, the classes also provide APIs for the GUI program to request sending data to the SystemC part. These implementations are using java multi-threading system.

4 Broker APIs

We defined and implemented the broker APIs for each side, which are a little different due to the use of the different languages. To make it easy to understand our APIs, we use a simple example prototype. The prototype structure is shown in Figure 4.

The prototype does a very simple process, it waits for a number as an input and returns zero when the input is an even number and returns one when the input is an odd number. The InputObjectX and its corresponding InputModuleX accept the input number from user. While the OutputModuleY and its counterpart OutputObjectY display the consequential even or odd number outcome.

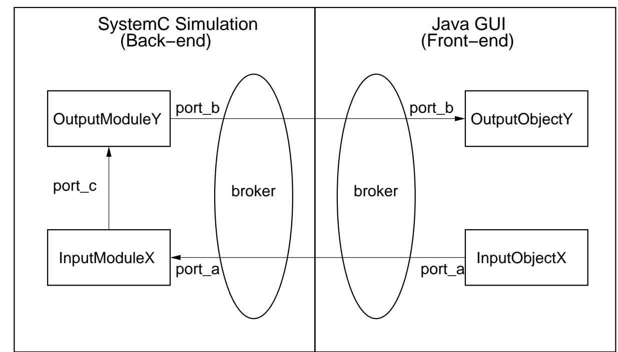


Figure 4: Structure of the simple prototype

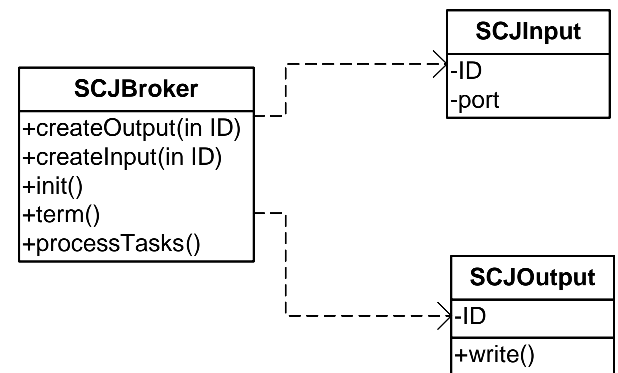


Figure 5: Class diagram for SystemC program

4.1 SystemC-side APIs

The SystemC part of the example program consists of three modules: `Broker`, `InputModuleX`, and `OutputModuleY`, and two ports: `port_a` and `port_b`. `Broker` is responsible for getting an input from the front end and passing it to `InputModuleX`, which waits for an input from `Broker` and passes it to `OutputModuleY`. Then, `OutputModuleY` processes the input and produces an output. The two ports are used as communication channels to transport signals and data between modules in the SystemC program. This library is designed so that the ports can also be used to transport signals and data between the SystemC program and the java (GUI) program.

Figure 5 shows the class diagram of the SystemC-side APIs. They should be implemented so that they fit into the SystemC program structure. There are three classes: `SCJBroker`, `SCJInput`, and `SCJOutput`. `SCJBroker` is a class for the brokers, and the other two are classes for the communication channels to exchange data between the broker and other modules.

The `SCJBroker` class must be handled in a special way, because of the way that the SystemC simulation mechanism works. The SystemC modules connect to each other using ports and signals. During the

run-time of the simulation, the SystemC simulation kernel creates an execution schedule for the modules and runs them in sequence according to the schedule. Each module has a special function, called by the simulation kernel at its execution, to turn on the schedule. However, the module is scheduled and run only when it is in a ready state. A module is triggered and becomes ready when the value of a sensitivity port changes. That is, only modules in the ready state can be scheduled to be run by the kernel. A module is not blocked unless it yields execution itself. Then, when the module is blocked, the kernel runs and switches to the next module. The running time of a module is called one cycle. Therefore, the `SCJBroker` class should be wrapped by a SystemC module so that it can be scheduled and run by the SystemC simulation kernel. The wrapped class of `SCJBroker` must be run as often as possible, i.e., every cycle, to make it virtually always running and available for the front end program. The class also requires write access to the input ports, because it writes to them each time inputs from the front end arrive.

`SCJBroker` is wrapped by a `SC_THREAD` module that is sensitive to a clock edge (positive or negative edge), so it can be scheduled every clock cycle. Each time the function is triggered, program control must be passed to `SCJBroker` inside the body of the module. The `SCJBroker` class has the processTasks() method for this purpose. Figure 6 shows the main part of the wrapper module of `SCJBroker`, `Broker`. Inside the main function BrokerLoop(), the broker calls the `Brkr.processTasks()` method that transfers control to `SCJBroker`. `SCJBroker` receives an input from the front end and processes it only when the method is called, so the method should be run as often as possible. That is why it is called inside the main function of the `Broker` module, which is scheduled every cycle. In this way, `SCJBroker` can look as though it is always running.

The wrapper module must also be bound to the ports from which the input modules receive the inputs through the broker from the front end. When it receives an input, `Broker` writes it to the input ports. Thus, the input modules expecting inputs from the front end can listen to these ports, without directly accessing `SCJBroker`. To allow `SCJBroker` to write directly to the SystemC ports, the ports should be registered first. Inside `Broker`'s constructor, each registration is done by calling `createInput()` of `SCJBroker` and passing the ID and instance of the port as parameters, as shown in Figure 6. `createInput` and `createOutput` methods create instances of `SCJInput` and `SCJOutput`, respectively, and automatically register the IDs to `SCJBroker`. The `init()` and `term()` methods

```
SCJBroker Brkr;
SC_MODULE(Broker) {
  sc_out<int> port_a;
  sc_in_clk Clk;

  SC_CTOR(Broker) {
     SC_CTHREAD(BrokerLoop, Clk.pos());
     Brkr.createInput("port_a", &port_a);
  }

  void BrokerLoop() {
     while(1) {
        Brkr.processTasks();
        wait();
     }
  }
};
```

Figure 6: `Broker` module

```
SC_MODULE(InputModuleX) {
  sc_in<int> port_a;
  sc_in<bool> port_b;

  SC_CTOR(InputModuleX) {
    SC_METHOD(inputX);
    sensitive << port_a;
  }
  void inputX() {
    if(port_a.read()%2)
      port_b.write(true);
    else
      port_b.write(false);
  }
};
```

Figure 7: `InputModuleX` module

must be called before the simulation starts and before the simulation ends, respectively. In the example, the `Broker` module is bound to `out_port_a`, which delivers inputs to be sent to `InputModuleY`. Inside the constructor of `Broker`, port `out_port_a` is also bound to `port_a`, an instance of `SCJInput`.

Whenever `SCJInput` receives an input from `SCJBroker`, it transforms the input into a SystemC signal and writes it to the corresponding port. The modules bound to this port receive the signal and then process it. Here, we can see that `InputModuleX` does not have any API object (see Figure 7. The module simply declares a port `port_a`. The input value from the front end is automatically converted into a signal in the port.

The implementation of the `SCJOutput` class is quite straightforward. Any modules that must send output simply create instance of SCJOutput and call its write() method directly. The corresponding SCJOutput requests SCJBroker to transfer the output to the other part. SJCManager transfers this by using its transferOutput() method. The ID of SCJOutput determines the destination of its output. It always arrives at the SCJOutput of the other part that has the same ID identifier.

```
SC_MODULE(OutputModuleY) {
  sc_in<bool> port_b;
  SCJOutput *sout;

  SC_CTOR(OutputModuleY) {
    SC_METHOD(displayY);
    sensitive << port_b;
    sout = Brkr.createOutputStream("port_b");
  }

  void displayY() {
    *sout->write(port_b.read());
  }
};
```

Figure 8: OutputModuleY

```
int sc_main(int argc, char *argv[]) {
  sc_signal<int> port_a;
  sc_signal<bool> port_b;

  sc_clock TestClk("TestClock", 10,0.5,0.0);

Broker BR("Broker");
BR.port_a(port_a);
BR.Clk(TestClk);

InputModuleX imX("InputModuleX");
imX.port_a(port_a);
imX.port_b(port_b);

OutputModuleY omY("OutputModuleY");
omY.port_b(port_b);

Brkr.init();

sc_start(-1);

Brkr.term();
}
```

Figure 9: OutputModuleY

Figure 8 shows the application of the write method in the example model. OutputModuleY uses the write() method to send output to users.

In the sc_main function, the init() function must be called before the simulation begins, i.e., before calling the function sc_start(). This function will block the simulation program and wait until it receives a connection from the Java front end. Before the simulation finishes, the term() function must be called to end the connection to the Java front end.

4.2 GUI-side APIs

Figure 10 shows the class diagram of the Java GUI-side implementation. This implementation is simpler than the SystemC part, because it is straightforward and uses the basic mechanisms found in most Java programs. The user input mechanism uses the popular Listener interface and the Adapter class provided in Java. It also exploits Java's event and thread mechanisms. Figure 11 shows the front end

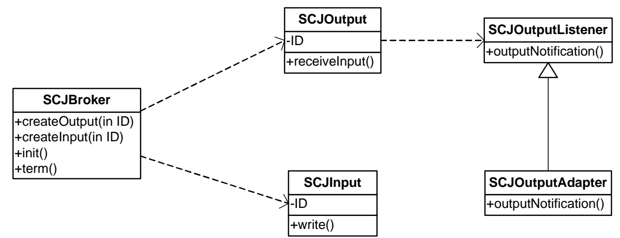


Figure 10: Class diagram for the GUI front end

Figure 11: Class diagram for the GUI front end

of the example prototype, which is implemented using standard Swing objects such as JButton and JTextField.

Similarly to the SystemC Part, inputs are sent from the user to SystemC by creating instances of SJCInput and calling their write() methods. The SCJBroker then receives this input and transfers it to the back end, using its transferOutput() method. The ID of SCJOutput determines the destination of its output. It always arrives at the SCJOutput of the back end that has the same ID identifier.

In the example model, the front end uses a button to trigger the sending of the input to the other part. The actionPerformed() method is responsible for handling the button-pushing event. Inside this method, the front end calls the write method of the corresponding input, i.e., port_a.

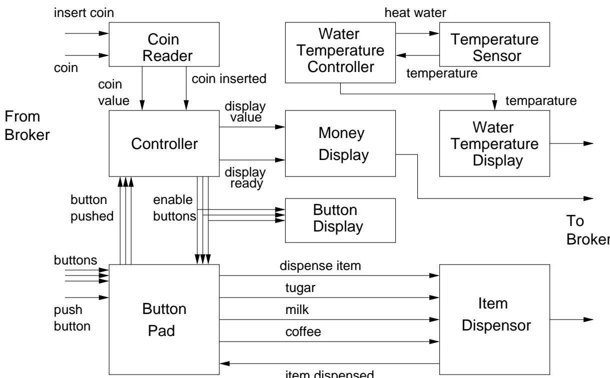
As mentioned above, the input mechanism uses Java events and the Listener interface and Adapter class. Every class that requires a particular input must listen to the SCJInput of that particular input. Whenever SCJBroker receives input from the SystemC part, it passes the input to the corresponding SCJInput. Then SCJInput notifies all of its listeners and passes the input to them.

The example code uses the InputAdapter to create the inner class to handle the input whenever it arrives. Then the handler processes the input; in this case, the handler simply adds some zero digits before the input number if necessary to make it always four digits long.

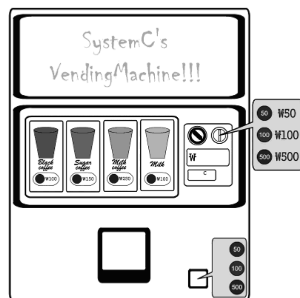
The createInput and createOutput methods create

```
...
port_b = Brkr.createInput("port_a");
...
public void actionPerformed(ActionEvent e) {
port_a.write(textA.getText());
}
```

```
display_value.addInputListener
         (new CommunicationInputAdapter() {
  public void receivedInput(String strInput) {
    System.out.println("read "+strInput);
    textB.setText(strInput);
  }
};
```



(a) SystemC Model



(b) Front end GUI

Figure 12: Vending machine prototype

SCJInput and SCJOutput respectively, and they automatically register their IDs to SCJBroker. The init() and term() methods must be called before the simulation starts and before it ends.

5 Case Study: Vending Machine Prototype

We implemented a vending machine prototype as a case study of our API library. Figure 12(a) shows the simulation model of our vending machine. The model consists of modules connected to each other by ports, and each module may be a composition of some submodules. Figure 12(b) shows the GUI front end of the vending machine prototype developed with Java Swing. The typical use scenario assumed in this vending machine is as follows. An incoming user inserts enough coins for an item that he or she wants to buy and then presses the button for the item, waits for a moment, and takes the item. Optionally, he or she can retrieve change by pressing the "Get Change" button.

```
SC_MODULE(MoneyDisplay) {
    sc_inout<bool> display_ready;
    sc_in<int> display_value;
    SCJOutput *sout;

    SC_CTOR(MoneyDisplay) {
        SC_METHOD(showDisplay);
        sensitive << display_ready.pos();
        sout = Brkr.createOutput("display_value");
    }

    void MoneyDisplay::showDisplay() {
        if(display_ready.read() == true) {
            *sout->write( \
                 i_to_a(display_value.read()) );
            display_ready.write(false);
        }
    }
};

display_value.addInputListener(\
            new CommunicationInputAdapter() {
  public void receivedInput(String strInput) {
    \\ display strInput
  }
};
```

All the modules except `Controller` are categorized into input modules and output modules. These modules connect to their counterpart objects in the front end. The output modules such as `ButtonDisplay`, `MoneyDisplay`, and `Water- TemperatureDisplay` require instances of `SCJOutput` in the back end and of `SCJInput` in the front end. For example, module MoneyDisplay shows how much money has currently been deposited by the user. This module calls its output "display_value" and creates an instance of SCJOutput with that identifier. Whenever the module must produce output, it calls the write() method of the corresponding SCJOutput instance.

The Java part listens to this input, "display_value", by using a listener interface. The Java part has the corresponding SCJInput with the same identifier, "display_value". The Java part registers its listener to this SCJInput so it knows every time a display_value is received from the other part. Then it displays the value after doing some processing of the display value. In this case, the value is formatted so that it always has four digits to be displayed, by adding any necessary zero digits in front of it.

On the other hand, to process input received from the Java part, the SCJBroker in the SystemC part must be wrapped within a module. As in the previous example model, in this vending machine model, the SCJBroker is wrapped within a module called Broker.

For example, the CoinReader module waits for the user to deposit coins. Thus, it must wait for input from the Java part. The implementation of the CoinReader module does not differ from the normal non-GUI implementation. The CoinReader module must

```
SC_MODULE(CoinReader) {
    sc_inout<bool> insert_coin;
    sc_in<int> coin;


    SC_CTOR(CoinReader) {
        SC_THREAD(readCoin);
        sensitive << insert_coin.pos();
    }


};

SC_MODULE(Broker)
{
    sc_out<bool> insert_coin;
    sc_out<int> coin;

    ...
    SC_CTOR(Broker)
    {
        ...
        Brkr.createInput("insert_coin", insert_coin);
        Brkr.createInput("coin", coin);
...
    }
};
```

simply be connected to a port, from which it receives coin inputs. In this case, the module connects to two ports, insert_coin and coin. The insert_coin port has a boolean value and acts as a flag that tells the module whenever a coin has been inserted. The coin port has an integer value and contains the value of the inserted coin.

The Broker module is responsible for delivering data received from the Java part to the module that expects the data, as in the case of the CoinReader module. The Broker module converts the data into a SystemC signal and sends it through the SystemC port connected to the respective module. Thus, the Broker module must be connected to all ports through which it can send data. Initially, the Broker module registers all possible inputs that it can receive and the port to which it must send the input.

The Java part itself receives input from the user interactively. The mechanism of retrieving input from the user is only limited by the Java GUI capabilities. After receiving input in some arbitrary way, the Java part sends the input through SCJOutput instances.

For example, when the user inserts a coin, the Java part receives this input and sends it to the SystemC module that expects this input, i.e., the CoinReader module. The Java part creates two SJCInputs instances and assigns their identifiers as “insert_coin” and “coin”. These identifiers are the same as the identifiers of SJCOutputs] in the SystemC part to which the input must be sent. When the user inserts a coin, the Java part sends this input by calling method write() of each SJCInput. To the insert_coin SCJInput, it sends the value 1, which means a coin has been inserted, and to the coin SCJInput it sends the money value of the inserted coin.

6 Conclusion

This work attempts to add a GUI front end to SystemC by using an independent Java GUI application and a set of API libraries to integrate the two independent programs. In this way, the SystemC simulation can become a GUI simulation and can communicate interactively with the user. This is in contrast to the normal SystemC simulation, which is text based and uses files for its input and output. With the added GUI front end, SystemC becomes more suitable for virtual prototyping of embedded systems, and developers can easily design and build a virtual prototype of a desired system.

References:

[1] J. Bhasker. *A SystemC Primer, 2nd Edition*. Star Galaxy Publishing, 2003.

[2] M. Hallmann. A Process Model for Prototyping. In *Proc. of Software Engineering and its Applications*, pages 9–13, Toulouse, France, December 1991.

[3] Frank Vahid and Tony Givargis. *Embedded System Design: A Unified Hardware/Software Introduction*. Wiley, 2002.

[4] Primer on Virtual Prototyping. http://www.gcrmtc.org/sbdc/protoprime_print.html.

[5] SystemC Community. http://www.systemc.org.

[6] SystemC v2.0.1 White Paper: A Summary of v2.0 Capabilities. http://www.systemc.org/projects/sitedocs/document /v201_White_Paper/en/1.