Appendix D

SICOSYS Programming Guidelines

D.1 Introduction

SICOSYS was designed having in mind its versatility and modularity, while keeping the user interface as simple as possible. This idea gave the project a chance of keeping up to date with new developments, therefore lasting a long time. This chapter aims to give a simple introduction to the way SICOSYS can grow by showing how to add a new network, component or traffic pattern.

D.2 Coding Style

Throughout the code of SICOSYS there are various conventions in the programming style. Most of them are just object oriented programming common practice, like keeping class attributes private and writing inline methods to access them or observing `const` correctness. Nevertheless there are two topics that might be new to the reader, constants and runtime type information.

The coding style can be resumed as follows:

- Class names should be preceded with three uppercase letters chosen by the programmer in order to avoid name collisions. The names should be lowercase with first letters of words in uppercase. As in `PRZCompositeComponent`.

- Class method names should be lowercase with first letters of words in uppercase except for the first word. As in `updateMessageInfo`.

- Class attribute names should follow the method naming convention and preceded by `m_`. As in `m_messageQueue`.

- Local variable names should help understand contents of the variable. Single letters should not be used, except for integer indexes in `for` loops.
- When a boolean variable is needed, it should be declared as `short` and the constant variables `true` and `false` should be used to assign and test it.
- Class definitions are written in files with the same name as the class with extension `.hpp` and stored in directory `inc`.
- Class implementations are written in files with the same name as the class with extension `.cpp` and stored in directory `src`.
- Three space indenting should be used.
- Comment should explain tricky pieces of code.
- Class members should be kept private. If they need to be used from outside the class, there should be inline members to access them.
- `const` correctness should be observed.
- References are preferred to pointers in most cases (but not all).

D.2.1 Constants

All constants should be centralized in the same file to allow changing the builtin limits and parameters. These include nearly all of the strings that are used in SICOSYS, including tag names and parameters. All this is grouped in a file called `PRZConst.hpp`. This file implements the strings as extern variables that can be accessed from every class that includes it. There is also a naming convention for the string identifiers. The names should be all uppercase using underscore as word separator and preceded by three letters chosen by the programmer in order to avoid name collisions. Similarly, error message strings are stored in a file called `PRZError.hpp`.

D.2.2 Runtime Type Information

In order to safely cast a base class pointer to a derived class pointer. SICOSYS has a runtime type information class that is included in most classes. The way this is done is very easy. First The definition of the class should have a line like the following:

```
DEFINE_RTTI(<class_name>);
```

Where `<class_name>` is the name of the class. And second, the class implementation file should include the following:

```
IMPLEMENT_RTTI_DERIVED(<class_name>,<base_class_name>);
```

Where `<class_name>` and `<base_class_name>` are the names of the class and its parent class respectively.

D.2.3 Adding a module to SICOSYS

SICOSYS has a directory called `mak` that contains what is necessary to build the binary program. Basically, it has a `Makefile` and `OBJS` directory that contains the object and other intermediate files.

In order to add a new module to SICOSYS, add the name of the module (no extension is needed) to the `MODULES` variable in the `Makefile`. Next execute the command:

```
make depend
```

This will analyze the dependencies between modules and put them in `Makedepend`. Now executing `make` should build SICOSYS.

D.3 Class Structure

SICOSYS has a large number of classes closely interrelated. Nevertheless, to implement new networks, components or traffic patterns, only a small subset of the classes must be known. In the following subsections, a small introduction to the structure of SICOSYS will be given. In figure D.1, a simplified view of the class diagram of SICOSYS is shown.

D.3.1 Components and Builders

Simulations are defined with the three SGML files `Simula.sgm`, `Network.sgm` and `Router.sgm`. Each of these describe the set of components that will conform the simulation. SICOSYS defines an abstract `PRZComponent` class that will be base class for all the components that are constructed as described in the SGML files. One of the aggregators of class `PRZComponent` is the abstract class `PRZBuilder`, its children specialize in reading one of the three SGML files. Thus, `PRZSimulationBuilder` specializes in reading `Simula.sgm` to create `PRZSimulation` components and similarly happens with `NetworkBuil-der`. But a router is defined as a set of components itself, therefore the `PRZRou-`

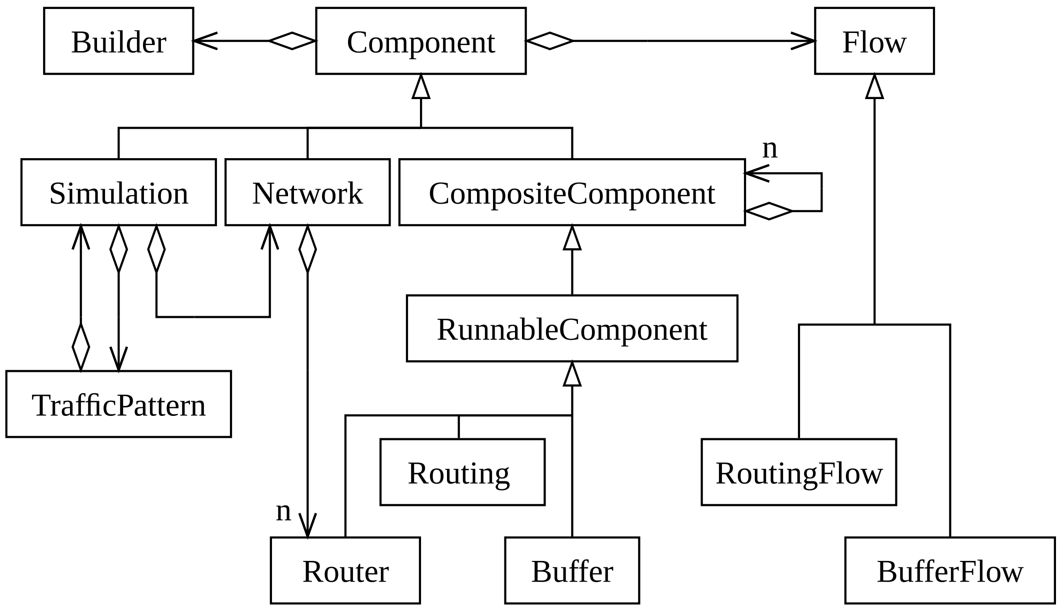

Figure D.1: Simplified class diagram

terBuilder class specializes in constructing a family of components that derive from PRZCompositeComponent. This class is an abstract class that can contain more components, enabling the creation of hierarchically structured components.

Once the building process is finished, SICOSYS has a structure of components that can simulate. The simulation component has a specific traffic pattern object, derived from PRZTrafficPattern, and an instance of a particular network, derived from PRZNetwork. In turn, the network object has a set of PRZRouter objects, one for each node of the network. Each router, as they are derived from PRZCompositeComponent has a set of components, such as buffers, routings or crossbars.

D.3.2 Components and Flow Control

As explained above, router components contain many other components that draw up its behavior. Nevertheless, these components do not have their functionality implemented in them. Each component delegates its functionality to an aggregator derived from the PRZFlow class. In this way a component can have a variety of flow control classes that implement different behaviors of the component. For example, the class `PRZFifoMemory` has the `PRZCTFifoMemoryFlow` class to implement the cut-through behavior and the `PRZWHFifoMemoryFlow` class to implement the worm-hole behavior.

D.3.3 Message Life Cycle

Once the building process is over, the simulation can start. During simulation, messages are created, sent to the network and received. SICOSYS has a class `PRZMessage` that represents the information flowing through the network, it can represent either a single flit, when it is in the network, or a full message, when it is generated by the traffic pattern class.

In the simulator main loop, there is a call to the traffic pattern object to generate a message for each router, this message is represented by a single `PRZMessage`. The message is sent to the network, then to the appropriate router and then enqueued in the injector component within the router. The injector reads the message object and finds out how many packets and flits it has. Then, it sends flits, each represented by a new `PRZMessage` object, through its output.

The `PRZMessage` object progresses from the output of a component through a `PRZConnection` object to the input of the next, eventually crossing to other routers and finally reaching a `PRZConsumer` object. This component notifies the network the arrival of the flit in order to perform statistical calculations.

`PRZMessage` objects are created in the `PRZTrafficPattern` and `PRZInjector` classes and destroyed in the `PRZConsumer`[1]. But in a simulation, there are thousands of messages to be created and destroyed, so SICOSYS has a special class that creates a pool of messages and manages their allocation and deallocation. The `SFSPool` class' `allocate` and `release` methods are much faster than the standard C++ operators `new` and `delete`.

D.3.4 Message Exchange Protocol

In order to transmit messages from a component to the next, a simple protocol has been implemented in SICOSYS. This protocol resembles the one used in real hardware, as show in figure D.2. It is based on two signals, `ready` tells the receiver component that there is a message ready to be read and `stop` tells the sender component that the receiver can not process messages.

To send a message one component calls `sendData` on the output with the message as a parameter. This sends the message to the output port and rises the `ready` signal causing the `onReadyUp` method to be called on the receiver. Normally the receiver class has an overridden version of the `onReadyUp` method

[1]In fact, there are complex routings and crossbars that exchange messages between them.

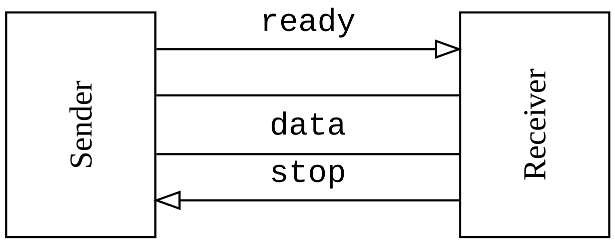

Figure D.2: Signals in component connections

that reads the message from the input port calling `getData` on the input port. Assuming the message is read, the sender component can call `clearData` to reset the `ready` signal at the end of the cycle.

The case might arrive that the receiver can not process more messages. This can happen when a crossbar has the desired output port engaged or when a buffer is full. In these cases the `sendStop` method is called on the input of the component. This in turn calls the `onStopUp` method on the sender component. Normally this method calls `sendStop` on the input, thus, propagating the `code` signal backwards. The `onStopUp` method can be overridden in case the propagation should not happen. For example, a buffer can still process incoming messages though its output is blocked by a `stop` signal.

D.4 Creating a new Traffic Pattern

There are many different types of traffic patterns already implemented in SICOSYS, nevertheless it might be interesting to add new kinds in order to test networks and routers under new conditions. Thus, the development of a simple traffic pattern and how it is coded into SICOSYS is explained bellow.

D.4.1 The New Traffic Pattern class

Each traffic pattern is implemented by a single class derived from `PRZTraffic-Pattern`. During the simulation the method `injectMessage` is executed any time a new message should be created. By overriding this method the new traffic pattern can be executed. If the traffic pattern needs some parameters to be set, the class must have the appropriate attributes to store them.

The example that will be illustrated sends messages from the source node to a specific neighbor. The parameters that it will take are the relative coordinates of the neighbor `dx`, `dy` and `dz`. These will be kept in the private part of the class and will be set by a public method.

D.4.2 The `injectMessage` method

For a simple traffic pattern like this, most of the code can be coded inline in the class definition. The only method that needs more than a line is the `injectMessage` method. This method takes a single parameter, the position of the source node, and should always return `true`. The `PRZPosition` is a vector class that represents a position within the network by holding the three coordinates that define it.

```
Boolean TUTTrafficPatternJump :: injectMessage(const
   PRZPosition& source)
{
   /* Create the message with source node */
   PRZMessage* msg = generateBasicMessage(source);

   /* Calculate target node */
   unsigned x, y, z;
   x = (source.valueForCoordinate(PRZPosition::X) + getX
      () + dx ) % getX();
   y = (source.valueForCoordinate(PRZPosition::Y) + getY
      () + dy ) % getY();
   z = (source.valueForCoordinate(PRZPosition::Z) + getZ
      () + dz ) % getZ();
   PRZPosition destiny = PRZPosition(x,y,z);
   /* Set target node in message */
   msg->setDestiny(destiny);
   /* Send message */
   getSimulation().getNetwork()->sendMessage(msg);

   return true;
}
```

In general, the `injectMessage` method has to:

- Create a message object, that will eventually be sent to the network.
- Calculate the target node from the source node and other external data, in this case, the parameters. Nodes should not be sending messages to themselves.
- Set the calculated target node in the message.
- Send the message to the network.

D.4.3 Traffic Pattern Creation

Once the traffic pattern class is defined, SICOSYS must be able to create instances of it. This is done by modifying the `createTrafficPattern` method of the `TrafficPattern` class. This method gets the traffic pattern tags from the `Simula.sgm` file and creates the appropriate traffic pattern object. It has a huge `if-else` chain to select the traffic pattern.

In order to create an instance of the traffic pattern class described above, the following code is required:

```
   else if( type == TUT_TAG_JUMP )
   {
      /* Create the traffic pattern object */
      TUTTrafficPatternJump* newPattern = new
         TUTTrafficPatternJump(simul);
      /* Read tag parameters */
      PRZString param;
      int x = 0, y = 0, z = 0;
      if( tag.getAttributeValueWithName(PRZ_TAG_PARAM1,
         param) )
         x = param.asInteger();
      if( tag.getAttributeValueWithName(PRZ_TAG_PARAM2,
         param) )
         y = param.asInteger();
      if( red3D && tag.getAttributeValueWithName(
         PRZ_TAG_PARAM3, param) )
         z = param.asInteger();

      /* Show error if parameters aren’t valid */
      if( !x && !y && !z )
      {
         PRZString err;
         err.sprintf( ERR_PRZTRAFI_001, (char*)
            PRZ_TAG_PARAM1 );
         EXIT_PROGRAM(err);
      }
      /* Set paraemter values in traffic pattern class */
      newPattern -> setJump(x,y,z);
      /* Set traffic pattern object */
      pattern=newPattern;
   }
```

Observe that the `TUT_TAG_JUMP` tag string must be declared in the `PRZConst.hpp` file and that it can not be used by to different traffic patterns. The instantiation of a traffic pattern normally involves four steps:

- Creation of the new traffic pattern object.
- Read tag parameters.
- Check validity of the parameters
- Set the read parameters and other information in the traffic pattern object.

D.5 Creating a new Network

The networks that SICOSYS simulates are based on a 2D or 3D arrangement of nodes. The way the connections between the nodes are made, leaves a great degree of freedom to implement many regular networks. In the following example, a variation on the concept of midimew network is implemented into SICOSYS, the square midimew network. An example with 64 nodes is shown in figure D.3.

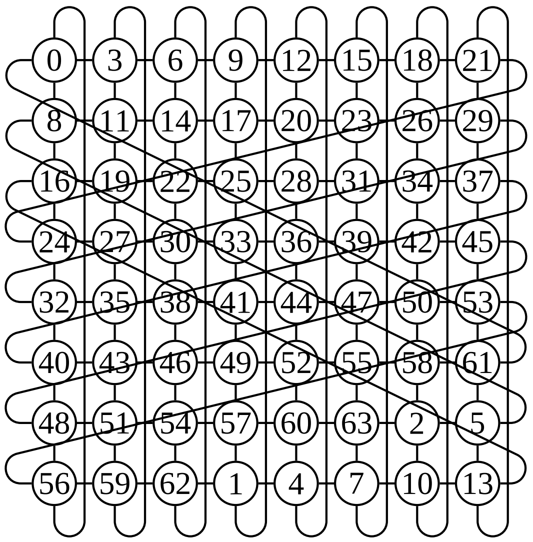

Figure D.3: 64-node square midimew

D.5.1 The New Network Class

As shown in the introduction, new network classes are to be derived from `PRZNetwork`. The new network class must implement the special aspects of the network type. A list of methods that must be overridden and what should they do is shown below:

`newFrom` Instantiate the class from the tag.

Constructor Initialize statistics.

`initialize` Create and connect the routers of the network.

`initializeConnectionsFor` Connect a given router to its neighbors.

`routingRecord` Calculate the routing record, given source and target nodes.

`distance` Calculate topological distance between two nodes.

`getDiameter` Calculate the diameter of the network.

`asString` Give description of the component.

Some of these methods are fairly simple and need not further explanation. Anyway, the methods involved with the creation process and those concerning the routing will be covered in detail in the following subsection. However, the complete code can be read in `inc/PRZSquareMidimewNetwork.hpp` and `src-/PRZSquareMidimewNetwork.cpp`.

D.5.2 The Creation Process

All networks are created from their sgml description in `Network.sgm`. The `PRZNetworkBuilder` class reads the file and calls the `newFrom` method of the appropriate class, passing it the tag read from the file. This can be done without an instance of the class because this method is `static`. In fact this method does the network object creation after reading and checking the tag passed by the builder. A close view of the `PRZNetworkBuider` shows that its method `parseComponentDefinition` has a big `if-else` chain. It compares the tag string against the tag names of the all network classes SICOSYS has already implemented. When adding a new network class, the new tag string must be declared in the `PRZConst.hpp` file and it can not be used by another network class.

After the creation, there are some parameters passed to the object, and then the `initialize` method is executed. The `initialize` method does some parameter checks before it is ready for creating the routers for each node of the network. Then with the aid of `initializeConnectionsFor` the routers are connected together. This method is called once for each router. It calculates all the neighbors and the port numbers associated with each routing direction.

Then it connects all the outputs of the router with the corresponding inputs of its neighbors. A little care must be taken in this part to avoid missing or duplicate connections.

D.5.3 Routing Through The Network

Once the router arrangement is constructed and connected, the network class is used to calculate the routing records of the messages. This is done by the `routingRecord` method. For other networks implemented in SICOSYS, there are direct ways of calculating the routing records, but for the `PRZSquareMidimewNetwork` there is no such direct method. Thus, a routing table is constructed by a brute force approach during initialization and the `routingRecord` just looks it up.

D.6 Creating a new Component

In order to compose the behavior of a router, SICOSYS has a selection of components that can be connected to each other. This section will explain how new components can be added to SICOSYS and how to implement their functionality. In SICOSYS, the structure and the behavior of a component are coded separately in two different classes. In fact, a component class can be associated to an arbitrary number of behavior classes.

D.6.1 The Component's Structure

SICOSYS' components are all derived from the `PRZRunnableComponent` class. It encapsulates the functionality needed by the simulator to execute the component. The component class stands for the structure of the component and is also responsible for the creation process. In the following example, the construction of a simple component will be presented. The component is the simple routing block that is implemented in SICOSYS. It has one input and one output, and it admits the following parameters in its description tag:

control the type of routing function to use.

headerDelay the number of cycles it takes to process the header of a packet.

dataDelay the number of cycles it takes to process data flits.

The definition of the `PRZRouting` class must have attributes to hold the parameters shown above. Anyhow, the base class already provides support for *headerDelay* and *dataDelay*, so only a member for the *control* is needed.

In order to properly read the tag, interpret its parameters and instantiate the class, the `PRZRouterBuilder` class calls the `newFrom` method when it needs to create a routing. This method is `static` and can be called when there is no object created of its class. The `newFrom` has to create a new object of the class `PRZRouting`, set all the parameters read from the tag and return it to the `PRZRouterBuilder` class.

Once the `PRZRouterBuilder` class has the instance of the `PRZRouting` class, it initializes it. This involves the creation of the flow class, that encapsulates the functionality of the component. When the component is initialized, the method `buildFlowControl` is called. This method must be overridden by the component class in order to create the flow class correctly. In the example there is a wide choice of flow classes and, depending on the value of the *control* parameter, one of them will be created.

D.6.2 The Flow Control Class

SICOSYS has many flow control classes for the `PRZRouting` class. Each provide a different protocol towards the crossbar or handles different packet structures. Control flow classes are like the brains of the component, they are able of reading parameters from the component, receive and send packets through the components inputs and outputs, etc.

Though all the flow control functionality of the component might be implemented in one class, it is sometimes worth considering designing a class that holds the common features of a component. Then, more specialized classes can be derived from this one to complete the behavior of a particular component. This is the case of the `PRZRouting` component in which `PRZRoutingFlow` provides basic methods for `PRZRoutingBurbleFlow`, implementing a bubble routing, or `PRZRoutingCVFlow`, for virtual channel routing.

This example will explain the `PRZRoutingBurbleFlow` class, it implements the DOR bubble routing. This routing reads packets from the input port and calculates to which, of the router ports, they should be routed to. If the packet is to be routed onto another dimension, the routing checks that the buffer lying in the new dimension has space for, at least, two more packets. This is the bubble condition. The packets have one header flit, specifying the number of hops, for each dimension. Therefore, before routing the packet to another dimension, the routing cuts the first header flit.

The methods that the `PRZRoutingFlow` class provides include some utility functions, that are called by the derived classes, and some virtual methods, that are called by the simulator. Among the first group there are:

`cutCurrentHeader` tells the caller if it should remove the current header flit of the packet. This is done by finding if the packet has reached the end of the current dimension.

`changeDirection` informs the caller if a flit should be routed to a new dimension. This is done by comparing the routing port required by the packet and the dimension on which the routing is located.

`getDirection` tells the caller on which dimension lies the component.

The virtual methods overridden by the `PRZRoutingFlow` class include two that are involved with an event queue that is used to implement the delays on the component's activity. These are:

`outputWriting` finds out if there is an event that should be processed and calls `dispatchEvent` to process it if necessary.

`dispatchEvent` process events. Events in routing components are very simple, they specify at what time should the message be sent through the output. The `dispatchEvent` method sends the message only if the `controlAlgorithm` method returns `true`, if not it generates a new event that should be processed on the next cycle. Performing, in this way, a retry mechanism. The `controlAlgorithm` method should be overridden by the derived class to properly control the sending of messages.

In addition to these methods there are `onReadyDown` and `onStopDown` that manage the flow control signals.

Mainly, the derived `PRZRoutingBurbleFlow` class needs only to override three methods, these are:

`onReadyUp` the simulator calls this method whenever there is a flit that can be read. When called, this method reads the new flit and, if it is not a header, it sends it immediately to the output. If the flit is a header, it calls `updateMessageInfo`, cuts the header if necessary, generates an event to be processed when the header delay has elapsed and tells the previous component to stop sending flits.

`updateMessageInfo` reads the message and calculates to which port it should be routed. Because the routing information is distributed between the header flits, this is done in several steps. The method return `true` if the packet must change dimension.

`controlAlgoritm` this method is called by the `dispatchEvent` method to know if it should send a message. It checks that the bubble condition is true before allowing the message to be sent.