ATLAS Level-1 Calorimeter Trigger
Software Note 011

01 February 2002
Version Draft 0.99

Overview of the Simulation Framework

L1Calo Group [1]

1 Introduction

This note is intended to form an user guide to developing software with the software simulation framework. More detailed reference documentation also exists [1]. This covers most of the material given here plus more information and detail not needed by most users of the framework. The reference material was generated from the in-line comments using Doxygen [2].

The guide starts with a general discussion of the architecture of the software, then goes into more details about the most important classes in the modeling class library. A simple example simulation is then described, with some discussion of implementation details, and finally there are brief notes on the some more detailed and exotic features, which are probably not needed for most circumstances.

2 Motivation

The intention of the simulation is to provide a functional description of the Level-1 Calorimeter trigger system at the level of data that can be recorded by the standard DAQ or in test memories throughout the system. When data is recorded at various points along the real-time path, the simulation can be used to verify the correct operation of the algorithms used. This procedure can also be used when known test data is being injected at one point of the the system.

The need for such a system comes from the complexity of the trigger electronics and the various test-setups that will be used in assessing the hardware. The necessary mapping from input to output could be done by simpler means for the final system, but during the slice-test and beyond, there are likely to be a large variety of different configurations. This software provides a way to model the modules and connections (as well as their settings) directly and can be integrated with the configuration database to automatically pick up the current situation. However, it does not go into the detail needed for a full hardware simulation, which would be too slow to run on a large integrated system.

[1] Please send any comments and corrections to Stephen Hillier.

3 Simulation Architecture

The architecture is based on a very simplified VHDL-like model [3]. In VHDL, a system is built up from Entities which perform the processing tasks. These Entities are connected by any number of Ports, which drive data from one Entity to the next. In a similar fashion, in the simulation software a basic processing unit is called a ProcessElement, and these can be connected together via DataPorts.

One of the major simplifications of the architecture compared to VHDL is a very basic time-structure. Since the software is intended to simulate the progress of real-time and readout data through a correctly timed-in level-1 system, there is no attempt to simulate anything below the sub-tick level. This means that the simulation can progress tick-by-tick through each stage of the system, with the downstream stages being processed first and the flow of control then proceeds upstream. This simplification also leads to the restriction that no feedback-like element can be simulated – the simulation can only flow in one direction, and any attempt to use a 'later' result upstream will cause a simulation error.

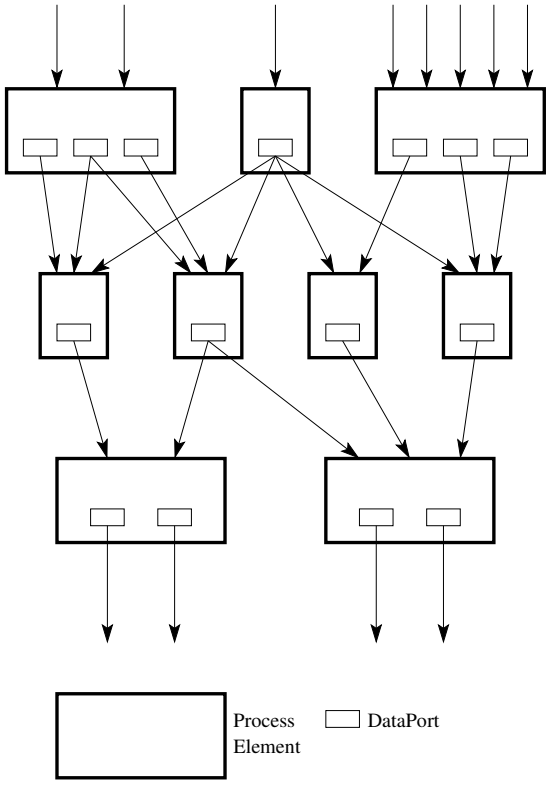
Another simplification is the treatment of DataPorts as all being contained in a single ProcessElement. Any ProcessElement can contain more than one DataPort. The contained DataPorts form the output of the ProcessElement. Inputs to a ProcessElement are formed by reference to the outputs of other ProcessElements. Any output can go to more than one input. The situation is illustrated in figure 3.

Other than providing the mechanism to process the flow of data, the other main function of the framework is to provide a mechanism for connecting inputs and outputs together. There are a few methods provided to do this, and they all work on the principle that each input and output is associated with a name and optionally a number. Typical examples might be an algorithm with inputs names such as 'EMTower' 1, 'HadTower' 2 and an output named 'NumberOfHits'.

The final major structural feature of the modeling classes is the ability to wrap a set of ProcessElements into a compound entity. This provides a way to build up complex elements from simpler ones, and then use them as if they were just another simple element providing outputs and requiring inputs – all the internal complexity is hidden in the class definition. This functionality is provided by the ProcessContainer class.

4 Class Library

The most fundamental part of the framework consists of seven classes. The other classes in the model library are common utility classes, and not part of the basic modeling scheme. One of the seven classes is the controlling Simulation class,

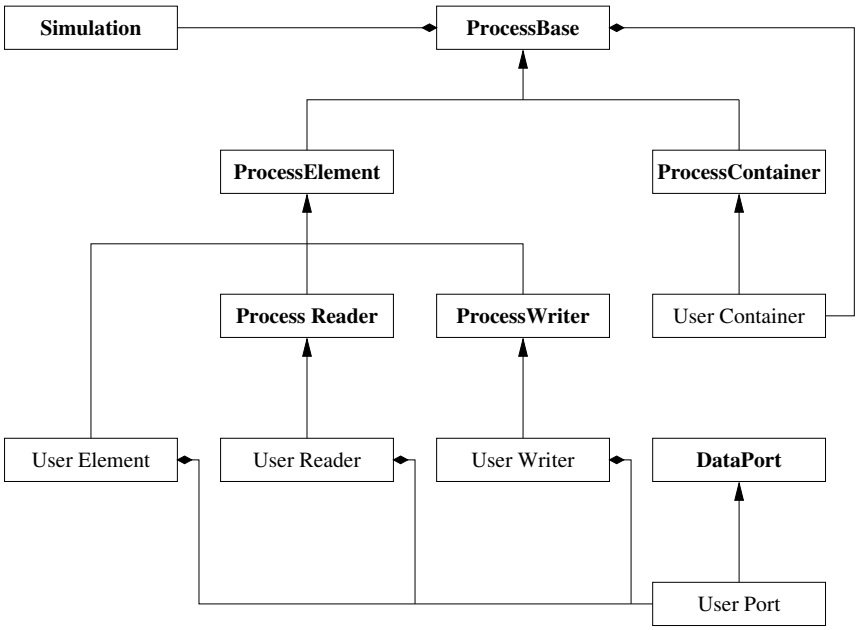


Compositional Structure of a Simulation.

which knows about all the elements of the simulation, and is used to initiate processing. The user must create just one of these per simulation. The other six are all abstract classes which provide the architectural mechanisms. In order to produce a meaningful simulation, concrete classes must be derived from these and instantiated to build up the model.

The main relationships between the seven fundamental classes are illustrated in figure 4. All components of a simulation, whether containers, or any type of processor, derive from ProcessBase. This is the main inheritance tree. Separate from this are the DataPorts, which are only connected to Process classes by containment. Note that ProcessContainers do not directly contain DataPorts, but just other ProcessBase derived classes (which might themselves contain DataPorts). Finally, the simulation must know about all of the ProcessBase derived objects to be used in the simulation.

Descriptions of the classes follow, although more detailed descriptions can be found in reference documentation [1].



Framework classes in bold
User defined classes in normal

Modeling Library Class Structure.

4.1 Simulation

This class is used to control the step-by-step functioning of a simulation. It is to some extent aware of all the elements included in its simulation, and all components that have to interact must be registered with the same Simulation object. One of the most important hidden functions is to remember which objects actually require to be processed to produce the required output. It is by requesting these objects to update that the simulation is invoked.

The most important methods to the user are operator+= and newEvent. The operator+= method registers a simulation component as a member of this Simulation. All components in a simulation have to be added to the simulation in this way for them to work in cooperation with the other components. The reason for having this association is that two completely seperate simulations can be run be the same program without confusion. If this is not done, not only will the simulation not run, but even trying to connect two component together will fail. The newEvent simply requests that the next event should be processed. This initiates the processing of a single simulation cycle (one tick), and returns the result – OK, end of data, or fail.

4.2 ProcessBase

This is the base class for all simulation components and its main task is to provide the connection mechanisms between elements. It provides both shared implementation for some features (mostly the inter-connection behaviour), and virtual member functions that must be implemented by the derived classes. Some useful information methods are also provided for debugging.

There are a few methods that the user needs to know about. Firstly those related to connecting input and output ports as seen by the steering program. These are connectInput, which requests the named input port of an object to be connected to a DataPort, and getOutput which asks for a pointer to a named output. There are actually two arguments to identify the port in both cases. The first is a name, and the second a number for cases where there are more than one input/output of the same name in that object. The number defaults to zero if not provided for ease of use with unique ports.

Methods which the user needs to know about for writing their own ProcessBase derived classes are registerOutput and inputValid. These methods provide the derived classes with a way to describe their inputs and outputs. Outputs are contained by an object, so a derived constructor can use registerOutput to simply advertise their location and names. The inputValid method is a virtual method which has to be over-ridden by derived classes to check if an input request is valid, and if so record internally where that input is coming from.

4.3 ProcessElement

This is the most fundamental element of the simulation. It provides the base class for any individual process that takes one or more inputs and provides one or more outputs. ProcessElement adds the code necessary to perform a processing stage on each step of the simulation. The usage of ProcessElement to the programmer is to derive a class from it which contains output DataPorts, and references to input DataPorts, and provide four methods, as described below.

Firstly, the user class should have a constructor and destructor. In the constructor, the output ports are created if necessary, and registered as outputs using registerOutput. Secondly, the ProcessBase method inputValid must be overridden to be able to connect up inputs. This method will be used to request an input connection. The class should check that the name and type of the input are correct, and if so, set the internal reference to that input to be equal to the DataPort passed in the arguments.

Finally, the actually processing stage is provided by overriding the virtual method process in ProcessElement. This method should look at the input val-

ues from the input DataPorts (provided that they have all been connected) and so derive the output values to set to output DataPorts. Success or otherwise is reported back in the return code.

4.4 ProcessContainer

In order to be able to build up a complex block of functionality from smaller blocks, there needs to be a way of grouping elements hierarchically. This is the base class for any object doing this grouping, or containing. It cannot do any processing itself, just contain any number of smaller blocks and connect them together. It must however behave like other elements in the way it handles inputs and outputs, which means that it must inherit these from ProcessBase, and it adds extra functionality related to containing simulation objects. Note that contained objects can be either ProcessElements or other ProcessContainers.

A typical user class derived from ProcessContainer would contain one or more other ProcessBase derived objects. These would be created and destroyed (if necessary) in the constructor and destructor. The constructor here however is slightly more complex than for a ProcessElement. There are actually four tasks is may have to do.

- Create the internal objects
- Register these using registerInternal
- Make necessary connections between internals
- Register outputs using registerOutput

The only new method here is registerInternal, which is provided by ProcessContainer for the containment mechanism. A corresponding getInternal method exists for probing into a container if needed.

The only other method a ProcessContainer has to provide is an override for inputValid. The reason for this is to appear, from the outside, similar to a ProcessElement. The implementation of inputValid for containers is however a little different. The reason for this is that a container has no input ports of its own, but rather it has to forward the connection request to the relevant internal. Note that the forwarded request can use a different name (and number) for the internal connection request.

4.5 ProcessReader

This is a base class for simulator elements that need to read input from a file. It adds a simple mechanism for connecting the simulation to test vector input files. The actual reading is done by a user written derived class, but this base class provides useful functionality for control over opening of files, and a flag for what type of input to expect. A typical ProcessReader will probably have no input ports, but will provide one or more output ports, although there is no restriction on either, as it still has all the capabilities of a ProcessElement.

One new method provided here is inputFromFile. This is used to connect the object to an input file, also telling it which file format to expect, if more than one exists. In order to actually read a file, a pure virtual method readInput is provided. This is overridden by a user class of this type to read in the next chuck of data in order to drive the outputs.

4.6 ProcessWriter

This is very similar to a ProcessReader, only clearly its intention is to write output results, rather than read input. Typically, it will have no outputs, but would be able to be connected to several inputs. Again, like a Reader, there is no restriction on input and outputs as it is a fully functional ProcessElement. One slight added complication is the fact that more than one file with a different format can be written at the same time, by specifying different files for each valid file format.

The two new methods of interest are outputToFile, which is used as a request from the main program to connect the output to a named file, and writeOutput, a pure virtual method which must be provided by the user derived class to write the data in the format required.

4.7 DataPort, IntPort and BoolPort

All objects which are to be used as either inputs or outputs to an element of a simulation must be derived from DataPort. It exists to provide a mechanism of relating one simulation element to another, and updating the necessary values of the port when needed. In order to do this, all DataPorts must be associated with an owner (which will be a ProcessElement), and the identification of that owner is the main purpose of this class. Derived classes provide the actual data that will be transferred between elements.

There is very little a user needs to know about the internal workings of DataPort class, only that all ports must be derived from it. However, the model provides a couple of useful DataPort derived classes, IntPort and BoolPort. These

will probably be sufficient for most interconnections between elements, although more specialized data formats may be needed in some cases. IntPort provides a port which can have any number of integer values, and BoolPort provides the same for boolean values.

Essentially, IntPort is just an integer array wrapped in a class which also derives from DataPort. The methods getWord and setWord are provided for access to the integer values, and these also provide naive array bounds checking. The constructor and accessors are defined in such a way that they can be used without indices if a port providing a single integer value is required. The BoolPort class is very similar, with the names of the accessors being getBit and setBit.

4.8 Other Modeling Library classes

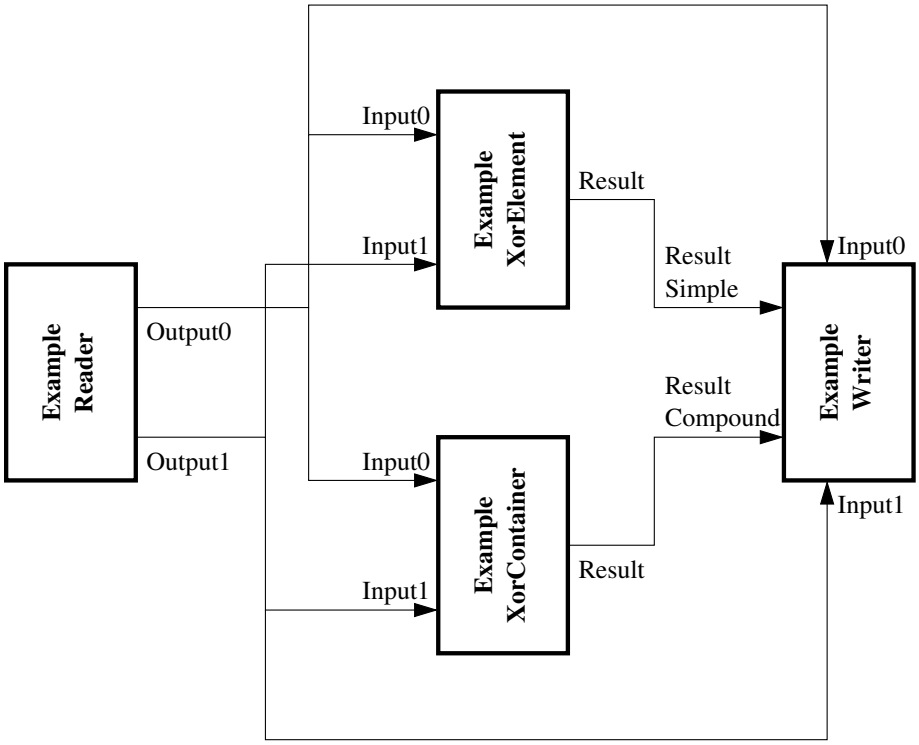
There are a few other general purpose utility classes provided in the library. Four of these are simple and common processes. There are three basic gates provided – GateAndN, GateOrN and GateNot. As the names suggest, these provide ProcessElements which perform simple logical 'and', 'or' and 'not' operations. For 'and' and 'or', provision is allowed for an indefinite number of inputs. ShiftRegInt provides a shift register for integer values. The final class is HexFormat, which is a very simple output formatter for values in padded hexadecimal, with the length of padding being set by the user.

5 Case Study

In order to illustrate the use of the modeling library, a simple example has been written and also documented in the reference manual [1]. The system that is being modelled is described here and notes on how this translates into the classes in the example are given.

5.1 Testing two versions of an Xor Gate

Two different ways to model an exclusive-or gate have been written. One using a simple ProcessElement, and the other by building up an xor from five component 'and', 'or' and 'not' gates. These are called ExampleXorElement and ExampleXorContainer respectively. Note that the second compound-type gate would probably be a rather unwieldy and inefficient way to model the 'xor' gate, but it is done this way to illustrate the construction of a ProcessContainer. It does however make apparent the fact that both classes have the same interface, both at the programming level and at the simulation level – ie they appear to have the same inputs and outputs and exhibit the same behaviour.



Test circuit for the two gates.

In order to test these two gates, we need to provide test-vectors, which is done by reading a file through a ProcessReader called ExampleReader. The results are output via a ProcessWriter called ExampleWriter. The whole simulation is controlled by the steering class called ExampleMain. Thus an example of user derived versions of all the main modeling classes is given, plus the usage of BoolPorts to connect them is shown.

The simple test 'circuit' is shown in figure 5.1. The connections and input and output names are given for each of the ports. The ExampleWriter receives both the inputs and the two gates' outputs in order to write out a truth table for both.

5.2 ExampleMain

The contents of the main steering program can be almost read off from figure 5.1. Firstly, the Simulation object is created:

```
Simulation exampleSim;
```

Then the four objects that make up the simulation are created, in lines similar to:

```
ExampleXorElement simpleGate;
```

Next, the four objects are added to the Simulation, for example:

```
exampleSim += compoundGate;
```

The next step is to connect the components together. This is generally a two step process. First find the output port of one object, then request it to be connected to its destination, eg:

```
DataPort* testA = testValues.getOutput( "Output", 0 );
simpleGate.connectInput( "Input", testA, 0 );
```

Note that a single output (eg Output0 from the ExampleReader) can be connected into several destinations. Once all eight connections have been made, all that remains is to attach the input and output files:

```
testValues.inputFromFile( "exercise.dat" );
resultLogger.outputToFile( "results.dat" );
```

before stepping through the simulation:

```
while ( exampleSim.newEvent() == procOk ) {
  // do something if required between steps
}
```

This steps through each set of test-vectors one by one until they are exhausted.

5.3 ExampleReader

The header file for the reader looks like this:

```
#include "ProcessReader.h"

class BoolPort;

class ExampleReader : public ProcessReader
{
public:
   ExampleReader( );
   virtual ~ExampleReader( );
protected:
   virtual ProcStatus readInput( ifstream& file, const FormatType type );
private:
// Outputs (owned)
   BoolPort* m_outA;
   BoolPort* m_outB;
};
```

It is quite a simple class with a constructor, destructor and one method which is invoked to read the file, and set the outputs. There are two outputs, which although logically contained in the class, and with tied lifetimes, are actually declared as pointers. This is to allow forward declaration of BoolPort, rather than inclusion of the header file. This is done to reduce header dependencies. It however does mean that the constructor and destructor have to explicitly create and delete the ports. The constructor also has to register the outputs, which it does with the

name ‘Output’ and numbers 0 and 1. The constructor and destructor are really quite trivial.

The final part of the implementation is the file reading. Obviously this will be very data dependent, but should follow some general rules. The main purpose is to drive the output ports to the values interpreted from the file. When the file is complete, the method should return procEnd to indicate ‘end of data’. This will propagate through the simulation and cause it to return ‘end of data’ to the calling program.

5.4 ExampleWriter

The header file for the writer looks like this:

```
#include "ProcessWriter.h"

class BoolPort;

class ExampleWriter : public ProcessWriter
{
public:
   ExampleWriter( );
   virtual ~ExampleWriter( );
protected:
   virtual ProcStatus process( );
   virtual bool inputValid( const string& name, DataPort* in, const int idNum );
   virtual void writeOutput( ofstream& file, const FormatType type );
private:
// Inputs
   BoolPort* m_inputA;
   BoolPort* m_inputB;
   BoolPort* m_resultSimple;
   BoolPort* m_resultCompound;
// Other members
   bool m_firstWrite;
};
```

The writer has a few more methods than the reader. The four member ports are now inputs rather than outputs, so do not have to be created in the constructor – in fact the constructor and destructor are pretty much trivial. However, in order to be connected, the class has to override the inputValid method to check connection requests to each of its four inputs. This is actually the most complex method – it checks that any request contains the right type of port (a BoolPort in each case), then the the name (and optional number if needed) are correct, and if so, the relevant port member is set to point at the new input port.

Processing is actually done in two stages. There is the usual process step for a ProcessElement. For most writers, there is no further processing to be done, but it is possible that an intermediate processing step might also be a writer, so it is allowed to do processing and setting of output ports here. In this case, and as

a general recommendation for writers, all that is done is to check that the inputs are valid. If not, an error will be generated by returning procFail. The other part of the processing is done in the method writeOutput – this is where the new data actually is written. In this case a simple truth table for the two gates is printed to the output file.

5.5 ExampleXorElement

This is a typical ProcessElement, with inputs and outputs and a process step. There are two inputs, and one output port, which has to be created and deleted in the constructor and destructor. It is registered under the name 'Output'. The class declaration looks like this:

```
#include "ProcessElement.h"

class DataPort;
class BoolPort;

class ExampleXorElement : public ProcessElement
{
public:
   ExampleXorElement( );
   virtual ~ExampleXorElement( );
protected:
   virtual ProcStatus process( );
   virtual bool inputValid( const string& name, DataPort* in, const int idNum );
private:
// Outputs (owned)
   BoolPort* m_outXorResult;
// Inputs (not owned)
   BoolPort* m_inputA;
   BoolPort* m_inputB;
};
```

The only two methods are the usual inputValid, which follows the same principle as the one in ExampleWriter, and the all important process step. The inputValid method allows two connections named 'Input0' and 'Input1'. The process method checks that the inputs are set, and if so uses their values to derive a result which is set as the value of the output port. This is of course a very simple calculation for an 'xor' gate. Thus the actual code for this class is very simple:

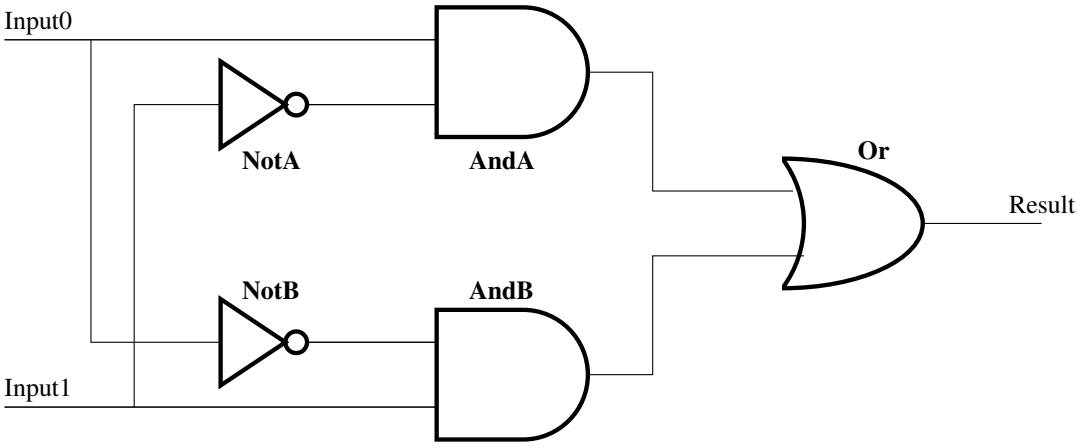
```
 m_outXorResult->setBit( m_inputA->getBit() ^ m_inputB->getBit() );
```

5.6 ExampleXorContainer

This is a rather more complex way to perform an 'xor' function, using constituent gates, but it forms a nice illustration of a typical container. The class declaration looks like this:

```
#include "ProcessContainer.h"

class GateNot;
class GateOrN;
class GateAndN;
```



Xor implemented by two 'and' gates, two 'not' gates and an 'or'.

```
class ExampleXorContainer : public ProcessContainer
{
public:
   ExampleXorContainer( );
   virtual ~ExampleXorContainer( );
protected:
   virtual bool inputValid( const string& name, DataPort* in, const int idNum );
private:
// Internals (owned)
   GateNot*  m_notA;
   GateNot*  m_notB;
   GateAndN* m_andA;
   GateAndN* m_andB;
   GateOrN*  m_or;
};
```

There are very few methods – the constructor, destructor and the inputValid method, but these are themselves a little more complex than usual. In order to see what is going on, the implementation of the 'xor' in terms of simpler gates is shown in figure 5.6. Instead of input and output ports, the class now contains five ProcessElements which represent the five gates. These of course have to be constructed and destroyed in the container constructor and destructor. The constructor also has several more jobs to do.

The first extra responsibility of the constructor is to register its internal components after their creation – this is done by called registerInternal on each object:

```
registerInternal( "NotA", m_notA );
```

A suitable name is given, and this is used by the getInternal method if a user needs to probe into a container. Then the internal connections must be made. This is very similar to the way connections are made in the steering program. Finally, the outputs have to be advertised using registerOutput. There is a subtlety here

though compared to a ProcessElement. Because the container doesn't own any output ports itself, it has to get the output ports from its most downstream internal elements – in this case the final 'or' gate. The complete constructor code looks like this:

```
ExampleXorContainer::ExampleXorContainer()
{
   // Create internals and register them
   m_notA = new GateNot;
   m_notB = new GateNot;
   m_andA = new GateAndN;
   m_andB = new GateAndN;
   m_or   = new GateOrN;

   registerInternal( "NotA", m_notA );
   registerInternal( "NotB", m_notB );
   registerInternal( "AndA", m_andA );
   registerInternal( "AndB", m_andB );
   registerInternal( "Or"  , m_or   );

   // Make internal connections
   m_andA->connectInput( "AndInput", m_notA->getOutput( "NotResult" ), 0 );
   m_andB->connectInput( "AndInput", m_notB->getOutput( "NotResult" ), 0 );

   m_or->connectInput( "OrInput", m_andA->getOutput( "AndResult" ), 0 );
   m_or->connectInput( "OrInput", m_andB->getOutput( "AndResult" ), 1 );

   // Register Output - the output of the Or gate
   registerOutput( "Result", m_or->getOutput( "OrResult" ) );
}
```

For similar reasons, the inputValid method is a little different for containers. Instead of checking and setting its own input port references, it forwards connection requests onwards to the upstream elements of itself, using the connectInput method on the internals. In fact, as is the case here, it may have to forward the request to more than one internal – each input gets connected to an 'and' gate and a 'not' gate. However, having written the inputValid method, there is no more to be done – containers do not have a process method to override. Here is what the inputValid method looks like:

```
bool
ExampleXorContainer::inputValid( const string& name, DataPort* in, const int idNum )
{
   bool retVal = false;

   // Forward conection requests to the downstream internal components
   if ( name == "Input" )
   {
      // Check which gates to send input to through number of connection
      if ( idNum == 0 )
      {
         retVal = m_notA->connectInput( "NotInput", in );
         // If first connection works, try second too
         if ( retVal )
            retVal = m_andB->connectInput( "AndInput", in, 1 );
      }
      // And now for second input
      if ( idNum == 1 )
      {
```

```
            retVal = m_notB->connectInput( "NotInput", in );
            if ( retVal )
               retVal = m_andA->connectInput( "AndInput", in, 1 );
         }

      }

      return retVal;
   }
```

5.7 Running the example

From the standard cvs download of the simulation package, using 'make example' should build the executable example.exe in the bin directory. If this is run in the example directory, a new file should be written called 'results.dat'. This should contain a correct truth table for both the simple gate, and the compound one.

6 Additional esoteric features

The methods described so far should provide a basis for most models, but a few additional capabilities are built into the library for unusual circumstances. A brief description of these is given here, but more details can be found in the reference documentation [1]. Most users will probably not have to read or understand this section in order to build a useful simulation.

6.1 More on runtime simulation control

The actual process by which a simulation step occurs is that one or more output ports are identified as having an interest in updating their value. In order to do this, the port requests its owner (the ProcessElement that contains it) to process. This it does by first asking its inputs to update, then performing its calculation. This request for updating clearly has to ripple upstream until it all inputs relevant to the required outputs are updated. Note that anything not needed to produce the required output will not be processed (and can get out of step if needed at a later time). The situation is illustrated in figure 6.1.

How does the simulation know which outputs are required? By default, the assumption is that all ProcessWriters that are attached to a file are required to be processed. The user can however tune this behaviour by setting other ProcessBase derived objects to be required, or even switch of outputs that aren't needed. This is done by the Simulation methods registerUpdateInterest and removeUpdateInterest. Care is needed to make sure that if an update interest is registered, then it is also removed before the Simulation object is deleted.

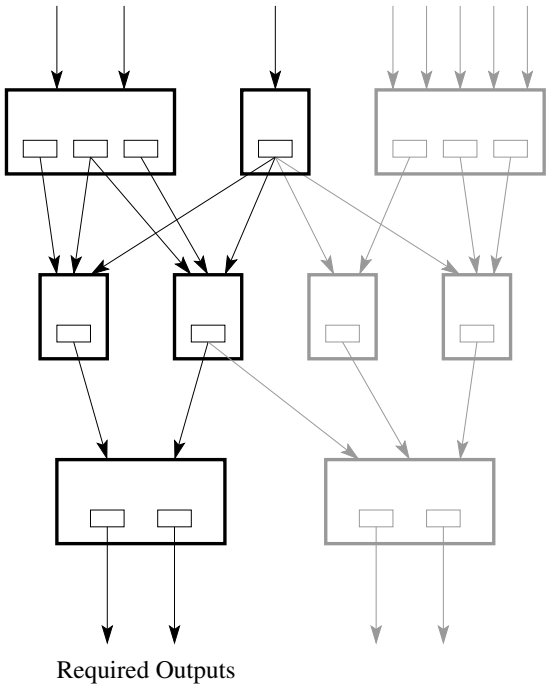


Figure showing updating processes – those in grey are not updated as not needed for the output shown.

6.2 Process update frequency

It may be in some instances that a process is only required to update infrequently, and not every tick. For example, if a set of test-vectors is being tested at various thresholds, then the thresholds only need to update at a constant integral number of ticks. There are various ways of achieving this, but there is a method built into the modeling library to cope with this situation.

The method setUpdateFrequency of a ProcessElement changes the update behaviour on each tick. It can be set to any positive integer, but the default is one. It sets the frequency of updating as a multiple of ticks. By default, process is called every tick, but if the frequency is set to, say, 10, then updates will only occur on the 1st, 11th, 21st etc event. This behaviour is achieved through a private method in ProcessElement called processIfNeeded. Note that it relies on the event number as maintained by the Simulation object, in particular, it requires that the event number be zero at the start of the simulation to set the phase of the updating cycle. This may be a problem if the same Simulation object is used several times over, and the event number will have been incremented by a previous set of simulation cycles. In order to cope with this problem, the Simulation class has a reset

method, which sets the event counter back to zero.

6.3 Modifying processing behaviour

The default scheme for each processing step in a simulation cycle is to call the virtual method process after all the inputs have been updated. This however can be modified if required by overriding another virtual method called processStages. By default it just calls process, but there may be new classes of simulation objects that require more than this one stage version. It would not be useful to do this for just one object, as it can just be done in the process method. However, for defining a new class of objects, it could be useful, and in fact this mechanism is used in implementing the ProcessReader and ProcessWriter classes, which have two stages – the process, then the read/write.

References

[1] Simulation Framework Reference Manual
`http://www.ep.ph.bham.ac.uk/user/hillier/level1/simulation/simref.pdf`
`http://www.ep.ph.bham.ac.uk/user/hillier/level1/simulation/html`

[2] Doxygen home page
`http://www.doxygen.org/index.html`

[3] VHDL International home page
`http://www.eda.org/vhdl_intl/www_vhdl_org_index.html`