Dominic Létourneau
Jean-Marc Valin
François Michaud
Carle Côté

FlowDesigner: the free data-flow oriented development environment

FlowDesigner is a free (GPL/LGPL) data-flow oriented development environment. It can be used to build complex applications by combining small, reusable building blocks. In some way, it has similarities with Simulink and LabView, although it is not designed to be a *clone* of any of them. FlowDesigner features a GUI that allows rapid application development and includes a visual debugger. It is written in C++ and features a plug-in mechanism that allows new blocks/toolboxes(sets of related blocks) to be easily added. FlowDesigner was designed with the following goals in mind: ease of use, speed, flexibility, expandability and modularity. Since it is not an *interpreted language*, it can be quite fast. FlowDesigner followed the same approach as for the C++ language which can be summarized by : *you don't pay for the features you don't use*. Although this development environment can be seen as a rapid prototyping tool, it can also be used for building real-time applications, such as Digital Signal Processing (DSP) and Artificial Intelligence (AI) applications.

This article focuses on basic concepts useful to build small FlowDesigner applications that use the Fuzzy Logic and the Artificial Neural Network (ANN) toolboxes. Some more advanced features, like building your own FlowDesigner blocks, data types and operators, are also discussed.

Terminology

This section defines the concepts and terms used by FlowDesigner. To help understanding those definitions, comparisons with Matlab and the C language shows how FlowDesigner can be related to each other.

Dominic Létourneau has a Bachelor's degree in Computer Engineering and a Master's degree in Electrical Engineering from the Université de Sherbrooke, Québec, Canada. Since 2001, he is a research engineer at the LABORIUS, the Research Laboratory on Mobile Robotics and Intelligent Systems of the Université de Sherbrooke. He has been working on FlowDesigner along with Jean-Marc Valin since 1999. FlowDesigner is being supported, enhanced and used at LABORIUS, with the help of Carle Côté, François Michaud and many others. The LABORIUS team have research interests that cover combination of systems and intelligent capabilities to increase the usability of mobile robots in the real world. This work is supported by the Canada Research Chair program.

Contact with the authors: *flowdesigner-devel@lists.sourceforge.net*

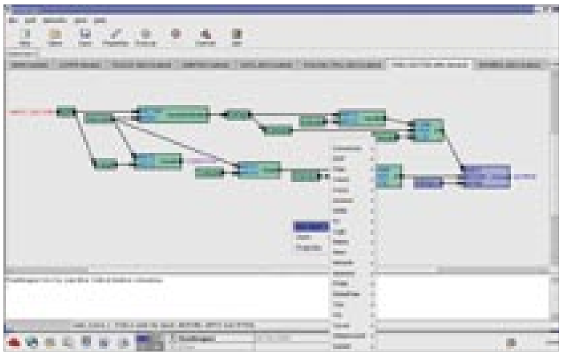

Figure 1. *FlowDesigner 0.8.2*

Nodes or Blocks

The basic processing unit in FlowDesigner is a *Node* (also called *Block)*. A *Node* is in all ways similar to a C or Matlab function as it takes some input data, performs some operations and outputs data.

Node terminals (inputs/outputs)

Each *Node* have inputs and outputs used to create interconnections with other *Nodes*. The inputs are equivalent to arguments in a Matlab/C function, which are variables of specific types needed by the function process. The same analogy stands for outputs, but in a less restrictive form than C where only one value can be returned. *Node* inputs and outputs are sometimes referred to as *Terminals*.

Node parameters

Node parameters are equivalent to C/Matlab constant variables. They are specified at *build-time* and stay constant throughout the run. *Node parameters* scope is local to the *Node* that contains them, meaning that they cannot be shared between *Nodes* at run-time. Like in C/Matlab, using *Node parameters* can improve run-time performance over the use of typical *Node* inputs, but lacks flexibility. Initialization of *Node parameters* is typically done by editing parameters fields contained in each *Node*, setting their types and values manually. It is also possible to define a *Node parameters* to be a `SUBNET_PARAM`, which indicates that this parameter will be initialized at *build-time* by the *Sub-network* containing the *Node* (see *Sub-networks* section).

Datatypes and Operators

Unlike Matlab that mainly supports the complex-double-matrix type, FlowDesigner (like C and C++) has support for many different types. The basic FlowDesigner types are: *Bool*, *Int*, *Float*, *Stream*, *String*, *Vector* and *Matrix*. There are also toolbox-specific types, like *FFNet* (Artificial Neural Networks), *VQ* (Vector Quantizer), *GMM* (Gaussian Mixture Model), etc.

There are many binary operators that are defined in the FlowDesigner operator tables, but the most useful are : *add*, *sub*, *mul*, *div*, *concat*, *greater*, *smaller* and *equal*. Operators are implemented as functions taking two input objects and returning the new resulting object after the operator is applied. Operators are defined for each object pair with different or identical types. The appropriate function to compute the result of each operator is determined at run time, looking for the data type of the operands first then calling the appropriate function. If the operator is not available for the operands data types, a run time exception is thrown. Applying operators on more than two objects is possible by cascading the binary operators.

Links

Links represents inputs/outputs connections between blocks. A *Link* can only be created between an input and an output that are data type compatible. This means that some *Nodes* expect a certain type of data as input and will generate a runtime exception (which will abort execution) if the wrong datatype is used (e.g. a `Load` *Node* expects a *Stream* as input and nothing else). Some *Node*s, like the `NOP` (no-op) *Node*, can take any type as input. Some *Nodes* have more complex behavior, like the `Add` *Node* that can add two *floats*, two *Vectors* of the same dimension, but cannot add a *Bool* and a *Vector*. There is no real correspondence between FlowDesigner *Links* and C or Matlab constructs.

Sub-networks (composite nodes)

A *sub-network* (or *subnet*) is a collection of connected *Nodes* that can be used as if they were a single *Node* (also called *composite Node*). Inputs and outputs of the sub-networks must be identified by giving names to the appropriate inputs/outputs of the connected *Nodes* collection. Most FlowDesigner subnets are saved into .n files, which are almost the exact equivalent of Matlab's .m files. There is no real C equivalent because C is a compiled language, although it could be seen as a C function calling another C function.

Sub-networks parameters

Like *Nodes*, *Sub-networks* can have parameters. In fact, *Sub-networks parameters* are *Nodes parameters* that are identified (`SUBNET_PARAM`) to be defined at *build-time* by the *Sub-network* containing them. They behave exactly as *Node parameters* and can be either defined manually or be defined in another *Sub-network* containing them.

Network

Network is the name given to the main *Sub-network* of a project.

Internal mechanisms of FlowDesigner

In order to understand data processing with FlowDesigner, this section explains the main mechanisms involved.

Pull and self-scheduling mechanisms

With data-flow *Networks*, two interaction mechanisms are typically implemented: push and pull. Pushing is when an interaction between processing elements is initiated by the data sender (producer); pulling occurs when an interaction is initiated by the data receiver (consumer). Push connections are appropriate for communication triggered by asynchronous events, while pull connections instructs the source element to send data only when the destination element is ready to process. FlowDesigner was originally designed for image and audio signal processing (DSP), having to deal with synchronous data processing. That explains why FlowDesigner uses pull mode architecture. The pull mechanism also provides the simplicity of designing processing elements that do not have to be aware of the others and where everything is self-scheduled. Self-scheduling happens when *Nodes* are asked to output their results: each output *Node* (sink *Nodes*) calls their input *Nodes* to compute recursively in order to be able to obtain the input data required for calculation. This kind of computation does not require to have a specific scheduler that tells when a *Node* has to process its input data. This simple implicit scheduling mechanism makes it possible to build *Sub-networks* from smaller functional *Nodes* without running into efficiency problems caused by scheduling overhead.

Sub-networks and Iterators

Every FlowDesigner program contains a *Network* called `MAIN`, which is equivalent to the `main()` function in a C program. However, you can add *Sub-networks*, equivalent of sub-routines from any *Network* or *Sub-network* that can contain several *Nodes* connected together. Doing so, you simplify the programming and you can reuse those *Networks* as *Sub-networks* in a higher level *Network*. It is very important to name the newly created *Network* a different name than `MAIN` for obvious reasons. Those *Networks* must absolutely have *named* inputs and outputs in order to be used in higher level *Networks*. To add *Sub-networks* into a *Network* of higher level, right-click on the background and select the *Sub-network* you want to add from the menu (New Node->Subnet).

Another useful type of *Network* you can create is the *Iterator*. An *Iterator* is a control structure that performs a loop. It stops looping when a certain control condition is met. The condition is a boolean value the *Iterator* gets from a *Node*. To define the *Iterator's* condition, left click on a *Node* output while holding the CONTROL modifier. When an *Iterator* is inserted into a higher level *Network*, it performs locally *X* iterations when its outputs are requested. Using *Iterators* enables the user to create feedback loops with the *Feedback Node* inser-

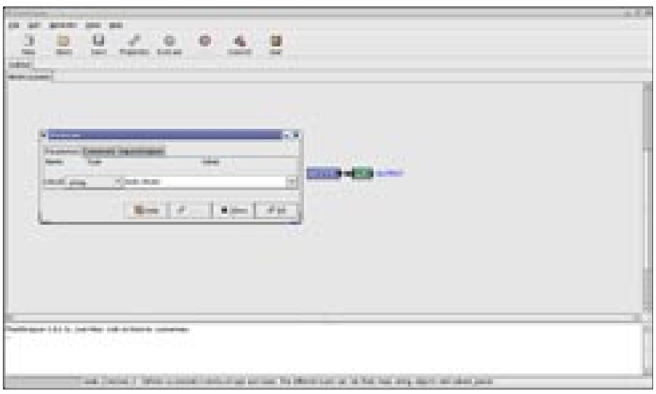

Figure 2. *Hello World Network*

ted into your *Iterator*. The *Feedback Node* allows to use values that are calculated *N* iteration in the past that are stored automatically in the output buffers of the *Nodes* preceding the *Feedback Node*.

Buffering

FlowDesigner's buffering mechanism allows *Nodes* to compute their outputs only once per iteration for better efficiency. During a given iteration, if *Node* A has calculated its outputs which are requested by *Node* B, *Node* A just returns the results stored in its output buffers, without propagating the request recursively to its input *Nodes*. Buffer size is managed by the system, enabling *Nodes* to request outputs over the *N* previous iterations, enabling the creation of feedback *Nodes*.

Automatic type checking and type conversion

Automatic type checking and type conversion are provided by the data-flow library. When linking *Nodes* together with the GUI, users are automatically notified when a link between two *Nodes* is invalid, which prevents errors and misuses of a *Node*. Type checking is also performed at run time. When a *Node* is expecting a particular type and does not receive this type as an input, it tries to convert the value in the desired type and if the conversion is not possible, throws an exception.

Automatic object creation and destruction

FlowDesigner uses reference counting pointers (or smart pointers), also referred as *ObjectRef* in the documentation and C++ code, to transport the data-type objects between *Nodes*. *ObjectRef* can be used like standard *Object** pointers, but will also count the number of reference to any *Object* derived class and delete the object when unused. This can be seen as a very simple garbage collector. Objects creation and destruction are handled by the system, to avoid dealing with memory allocation. Objects can also be allocated in *memory pools*, which enables the data-flow library to reuse the already allocated memory for certain object types.

Dynamically loaded toolboxes

FlowDesigner toolboxes are loaded dynamically when starting the application. The default path is scanned recursively (*/usr/lib/flowdesigner/toolbox/*) for toolbox definitions (.def files) and for toolbox libraries (.tlb files). The user can also set environment variable named FLOWDESIGNER_PATH, for user-defined toolbox directories, which is useful when having a system wide installation of FlowDesigner and local installation of some user-defined toolboxes.

Figure 3. *FOR loop iterator Sub-network*

On the Net

- FlowDesigner home page
  http://flowdesigner.sourceforge.net/
- LABORIUS : Research Laboratory on Mobile Robotics and Intelligent Systems
  http://www.gel.usherbrooke.ca/laborius/
- RobotFlow toolkit for FlowDesigner
  http://robotflow.sourceforge.net/
- Mobile and Autonomous Robotics Integration Environment (MARIE)
  http://marie.sourceforge.net/
- Octave toolbox
  http://www.octave.org/
- GNOME2 and GTK2 developer's site
  http://developer.gnome.org/

Developing applications with FlowDesigner

Creating your first FlowDesigner Networks

The first example is the classical *Hello World*. Figure 2 shows a Constant *Node* with its *Node parameter* value defined as a *String* initialized to "*Hello World*". This *Node* is connected to a *Print Node* which outputs its input value in a text console when pulled. The output terminal of the *Print Node* need to be named to indicate that this output terminal need to be pulled by the self-scheduling mechanism of FlowDesigner. Running this *Network* produces a single "Hello World" printed on the console and exits.

The Figure 3 show how to create a simple *FOR* loop which prints iteration count in console at each iteration. Listing 1 shows an equivalent program written in C.

Figure 3 shows how a loop is implemented in FlowDesigner. To create the loop, *Sub-network* type *Iterator* is required. Iteration condition is also needed to evaluate the exit condition of the iteration control loop. In this case, CONDITION terminal is pulled to evaluate that the iteration count is less than 5 at each iteration. If the condition is true, OUTPUT terminal is pulled and will print the iteration count in a text console. Otherwise, FOR_LOOP *Sub-network* iteration ends and returns the OUTPUT value to the MAIN *Network*.

Inserting graphical probes to help you debug your application

In order to understand how FlowDesigner handles FOR_LOOP *Iterator Sub-networks*, it might be interesting to see at run time how it behaves. Figure 4 shows a modified FOR_LOOP *Sub-*

Listing 1. *Equivalent to Figure 3 program written in C*

```
int main(int argc, char **argv)
{
   for(int i = 0; i < 5; i++)
      printf("<Int %i >\n", i);
   return 0;
}
```

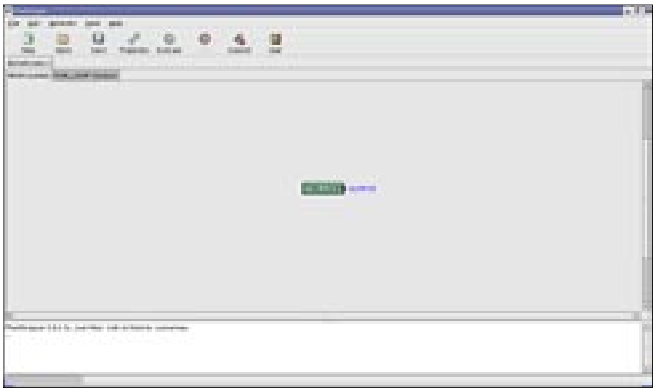

Figure 4. *FOR loop iterator Sub-network with* `TextProbe`

network that uses a TextProbe to output the `CONDITION` value at run-time. At run-time, a probe window is displayed (CONDITION window in Figure 4) allowing to show the value of the data that flows in the probe. Using *Forward*, *Stop* and *Execute* buttons in the probe window, it is possible to do a step-by-step debugging strategy by tracking data at precise iteration count of FlowDesigner's process.

Execution of FlowDesigner networks from the console

An application called *batchflow* is provided with FlowDesigner. It allows to run *Networks* created with the FlowDesigner GUI in console mode. The only requirement for the *Networks* to run is that the user needs to avoid using graphical probes into them. The FlowDesigner *Networks* can also be treated as shell scripts and are executed directly. The first line of each *Network* file contains `#!/usr/bin/env batchflow` to tell the shell which application to run to parse the rest of the file, containing the XML description of the *Network*. An application called *gflow* is also provided with FlowDesigner to allow the user to run *Networks* without the integrated development interface but with graphical probes.

Using available FlowDesigner toolboxes and external applications

Using the Fuzzy Logic toolbox

Figure 5 shows how the Fuzzy Logic toolbox can be used to control a fan. This example is provided with the FlowDesigner Fuzzy Logic toolbox. Extensive use of the *Concat* operator is performed in order to create three fuzzy sets : *HUMITIDY, TEMPERATURE* and *SPEED*. Each *FuzzySet* contains three fuzzy trapezoidal membership functions. The top right of the Figure 5 shows an example of parameters that are used to set the coordinates of one trapezoidal function. The fuzzy controller is created with the *GenericModel Node*, taking the *HUMIDITY* and *TEMPERATURE* sets as the antecedent sets and SPEED as the consequent set. The last *FuzzyRule* out of nine rule possible is displayed in Figure 5, which tells the *GenericModel Node* the following rules : *IF TEMPERATURE is HIGH and HUMIDITY is HIGH THEN the SPEED is HIGH*. The *FuzzyRules* and *FuzzySets* names must match for the *GenericModel* to be created properly. *FuzzySets* used for the antecedent part of the rules and *FuzzySets* used for the consequent part of the rules must be grouped together with the *Concat* operator. The *GenericModel Node* implements the Mamdani model and uses the center of area (COA) method for defuzzification. The Fuzzy Logic toolbox is not complete yet, but already contains all the basic *Nodes* and data types to create systems with unlimited number of membership functions, sets and rules. Complete fuzzy systems can be created and saved to disk for use with user-defined setup and FlowDesigner *Networks*.

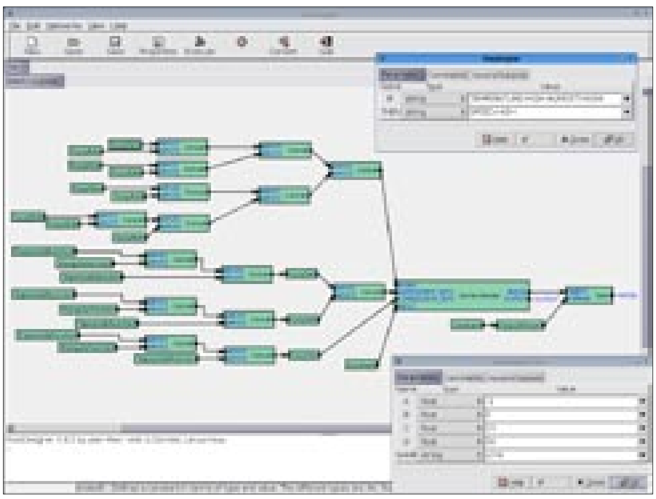

Figure 5. *Fuzzy Logic toolbox*

Using the Artificial Neural Network toolbox

The Artificial Neural Network (ANN) toolbox provides easy to use *Nodes* to train the ANN and to use already trained ANNs to process new input data. Figure 6 shows an example taken from the RobotFlow toolbox available at *http://robotflow.sourceforge.net/demo.html*. This demo contains FlowDesigner *Networks* that are used to recognize alphanumeric printed text extracted from color images using the FlowDesigner ANN toolbox. Before training the ANN, the user must classify image templates to make input and output sets, represented as input and output *Vectors* in FlowDesigner. The idea with this demo is to provide a simple algorithm that extracts characters from color components. Characters are extracted from the image with a simple color segmentation algorithm, with the black color representing text and orange representing the background color. For each template image, a pair of input and output *Vec-*

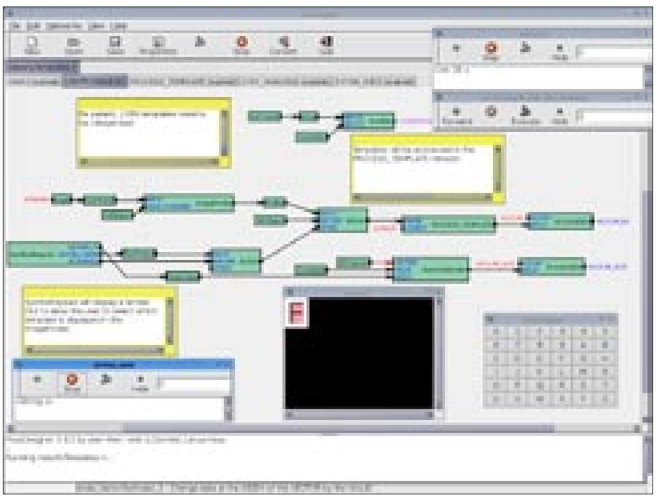

Figure 6. *Input and output sets required to train the Artificial Neural Network*

tors is created. Each extracted character is scaled and transformed into a *Vector* to create the input *Vector.* The user selects, with the *SymbolKeypad* GUI, the appropriate character that corresponds to the template image. An output *Vector* associated to the input *Vector* is then created. Once each character is classified, the training set containing input/output *Vector* pairs will be saved to disk and used to train the ANN subsequently.

Figure 7 shows the FlowDesigner training *Network* that is used and all the parameters given to the *NNetTrainDBD* Node. The *NNetTrainDBD Node* needs the previously saved input and output sets and an initialized ANN. The initialized ANN is composed of proper layer(s) configuration and is created with the *NNetInit Node.* The topology of the ANN is specified as a *Node parameter* and each ANN layer is composed of an arbitrary number of neurons and corresponding weights randomly initialized. The *NNetTrainDBD* uses the delta-bar-delta training algorithm that adapts the learning rate automatically. The *NNetTrainDBD Node parameters* GUI shows that the training will be done in 2000 epochs. Finally, once the ANN is trained and the user is satisfied with the results, the weights and the ANN configuration are saved to disk. Multiple ANN configurations and training algorithms are available in the ANN toolbox. The user is encouraged to try the demonstration to better understand how to use them.

Using FlowDesigner, RobotFlow and MARIE for robotic applications development

RobotFlow and MARIE are two projects that are currently using FlowDesigner. RobotFlow is a mobile robotics toolkit based on the FlowDesigner project. The visual programming interface provided with FlowDesigner helps visualize and understand what is really happening in the robot's control loops, sensors, actuators, using graphical probes and debugging in real-time. MARIE, which stands for *Mobile and Autonomous Robotics Integration Environment*, is a robotic development and integration environment focused on software reusability and exploitation of already available APIs and middlewares frequently used in robotics. One of MARIE's goal is to expand stand-alone applications scope, like FlowDesigner/RobotFlow, by adding the possibility to create interactions between them and by adding the possibility to distribute them on multiple

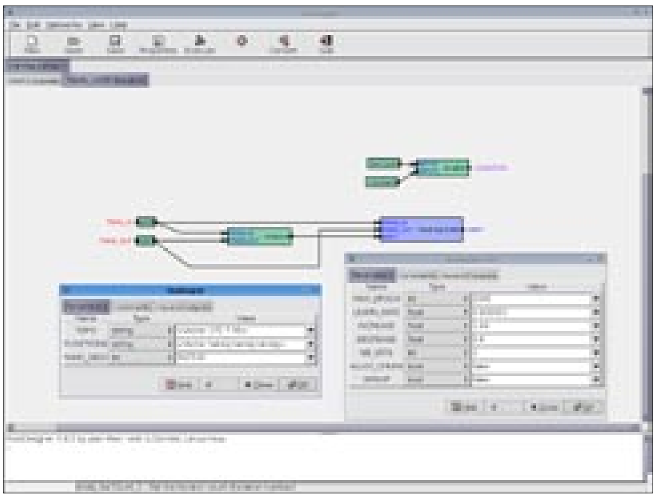

Figure 7. *Training the Artificial Neural Network with FlowDesigner*

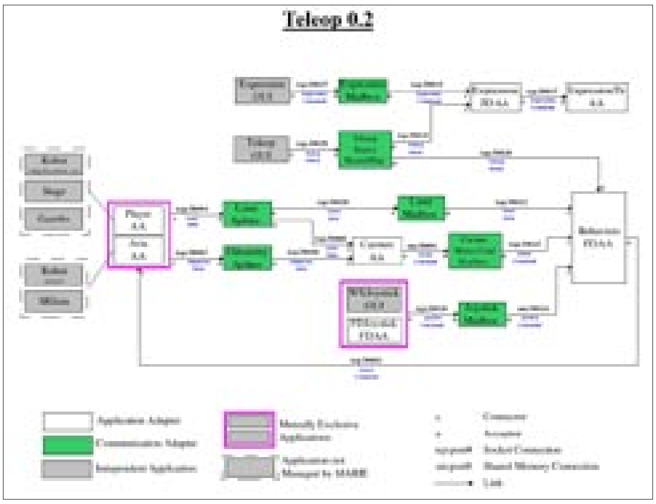

Figure 8. *Semi-autonomous teleoperated robot using MARIE and FlowDesigner*

processing *Nodes.*

Figure 8 shows a semi-autonomous teleoperation project created with MARIE. In this project, FlowDesigner/RobotFlow have been used to create control logic and glue logic required to interconnect and control all applications involved in the project. The components *Behaviors FDAA*, *Expression FDAA* and *FDJoystick FDAA* are three FlowDesigner independent *Networks* running in separated process managed by MARIE. Developing control and glue logic in FlowDesigner accelerated the overall development by using already available functionalities, by having a graphical representation support for implementation, and by having access to debug tools for investigating undesirable robot's behaviors at run-time.

Expanding FlowDesigner

Building your own nodes

In FlowDesigner, all *Nodes* are implemented in C++ as a class that derives, directly or indirectly, from a base class called `Node` (note that most *Nodes* derive from `BufferedNode`). Creating new *Nodes* does not require knowledge of FlowDesigner's internal processes and design, but only the procedure to define inputs, outputs, parameters, and the desired processing function for calculation by the *Node.*

Listing 2 shows a simple `MyNode` *Node* that adds two values and transfers the result in its output. Most of the new *Nodes* will derive from either the `Node` abstract class or the `BufferedNode` abstract class. You should use public inheritance when deriving your new class. In all cases, you need to define a constructor for your new `Node` class. The parameters for this constructors are: `string nodeName, const ParameterSet &params`), which are used to initialize the base class. Also, if you derive from `BufferedNode`, you need to define the `virtual void calculate(int output_id, int count, Buffer &out)` function. The arguments are the ID of the output requested (`output_id`), the iteration ID (`count`) and the output buffer for the requested output (`out`). The calculate function is expected to assign an object to `out[count]`. If you derive directly from the `Node` class, you need to override the `ObjectRef getOutput(int output_id, int count)` function. The meaning of `output_id` and `count` is the same as for the `BufferedNode` equivalent, and

Listing 2. *FlowDesigner user-defined C++ Node*

```
#include "BufferedNode.h"
#include "Buffer.h"
#include "operators.h"

//Forward declaration
class MyNode;

DECLARE_NODE(MyNode)
/*Node
 *
 * @name MyNode
 * @category Operator
 * @description Adds two input values and returns the result
 *
 * @input_name INPUT1
 * @input_description First value
 * @input_type any
 *
 * @input_name INPUT2
 * @input_description Second value
 * @input_type any
 *
 * @output_name OUTPUT
 * @output_description Result of the addition
 * @output_type any
 *
END*/

class MyNode : public BufferedNode {

   int input1ID;
   int input2ID;
   int outputID;

public:

   MyNode(string nodeName, ParameterSet params)
   : BufferedNode(nodeName, params)
   {
      input1ID = addInput("INPUT1");
      input2ID = addInput("INPUT2");
      outputID = addOutput("OUTPUT");
   }

   void calculate(int output_id, int count, Buffer &out)
   {
      ObjectRef inputValue = getInput(input1ID, count);
      ObjectRef input2Value = getInput(input2ID, count);
      //output the result in the out Buffer
      out[count] = inputValue + input2Value;
   }
};
```

the result should be returned as an *ObjectRef* (smart `Object` pointer).

Listing 3. *Defining a new FlowDesigner data type*

```
#include "Object.h"
class MyType : public Object
{
  private:
   //private variables / functions

  public:
  MyType()
  {
   //implementation of the default constructor
  }
  MyType(const MyType &cpy)
  {
   //implementation of the copy constructor
  }
  virtual void printOn(ostream &out = cout) const
  {
   //implementation of printOn(...)
  }
  MyType& operator+ (const MyType &obj)
  {
   //Implementation of the operator +, useful for
                      FlowDesigner
   //add operator used in Listing 3.
  }

  virtual void readFrom(istream &in);
  {
  //implementation of readFrom(...)
  }
};
DECLARE_TYPE(MyType);
```

Comments at the beginning of the source code (starting with `@`) define in which category the *Node* belongs, the description of the *Node*, and the definition of all inputs, outputs and parameters. The C++ code must match the textual description for the *Node* to work properly. The `DECLARE_NODE(MyNode)` macro is used to register the *Node* in a dictionary when the toolbox is dynamically loaded. Once ready for use, the definitions of the *Nodes* (C++ comments) are then parsed by a PERL script (*info2def.pl*), provided with FlowDesigner, to produce an XML description of each *Node* for each toolboxes. The FlowDesigner GUI is both using the definition of available *Nodes* and the internal *Node* dictionary to display the usable *Nodes* in the *Nodes* selection menu.

Defining your data types and operators

Using standardized data types and operators reduces complexity of the C++ *Nodes*, improves code readability and helps uniformize *Nodes*. User-defined data types and operators can easily be added in new toolkits.

Listing 3 shows how to create your own *MyType* data type. In order to be used in new *Nodes*, new types must derive from the `Object` base class. That is the only absolute requirement. However, if you want the new type to be integrated with FlowDesigner, there are several things you can do:

Listing 4. *Defining a new FlowDesigner operator*

```
#include "operators.h"
#include "net_types.h"
ObjectRef addMyType(ObjectRef op1, ObjectRef op2) {
   //Smart pointers to MyType objects
   RCPtr<MyType> op1Value = op1;
   RCPtr<MyType> op2Value = op2;
   //return the result of the addition
   return ObjectRef(new MyType((*op1Value) + (*op2Value)));
}
REGISTER_DOUBLE_VTABLE(addVtable,addMyType,MyType,MyType);
```

- Implement the `void printOn(ostream &out)` const function. This function writes the object to the out stream in the FlowDesigner format.
- Implement the `void readFrom (istream &in)` function. This function reads the object from the in stream in the FlowDesigner format.
- Add the macro `DECLARE_TYPE(MyType)` to the C++ file where the object is implemented. This adds the new *MyType* object type to the FlowDesigner type dictionary.

Listing 4 shows how to define a new *add* operator for our new *MyType* type. The `REGISTER_DOUBLE_VTABLE` macro is useful to register the add operator in the `addVtable (add table)` with the input types *MyType* as the first operand and second operand.

Conclusions

FlowDesigner is still in its early stage and is a work in progress. Nevertheless, it is already usable for a lot of applications. The easiest way to use FlowDesigner is by using its graphical user interface (GUI) and connecting existing *Nodes* together to form the data-flow *Network*. Also, the user can build its own *Nodes* and toolboxes without knowing all the underlying principles and classes used by the data-flow processing engine. Future FlowDesigner improvements will include:

- Better documentation and more examples,
- Support for both Linux and Windows,
- Octave toolbox,
- GUI improvements,
- Better support for importing and exporting *Networks,*
- More visualization *Probes.*

FlowDesigner and the related projects are developed by LABORIUS, the Research Laboratory on Mobile Robotics and Intelligent Systems, Québec, Canada. Any suggestions or contributions are welcomed to improve FlowDesigner. Do not hesitate to contact the authors if you need more informations.