DEVELOPER'S MANUAL

EXTENDING FUCNTIONALITIES OF THE ELVIS iLABS ARCHITECHTURE

Author: Adnaan Jiwaji

adnaan@mit.edu

TABLE OF CONTENTS

1. INTRODUCTION

The ELVIS (*Figure 1*) is an electronic workbench that is used to design and test circuits. It offers a variety of features that can be used for circuit analysis and debugging. This includes: Oscilloscope, Digital Multimeter (DMM), Function Generator, Variable Power Supply, Bode Analyzer, Arbitrary Waveform Generator, Dynamic Signal Analyzer (DSA), and Microelectronics Device Characterization capability

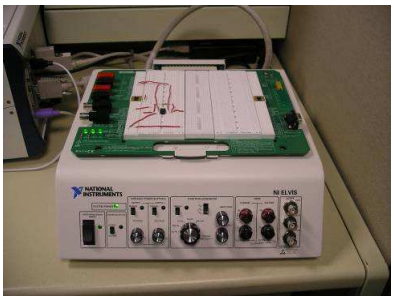


Figure 1: The ELVIS Workbench

.

The iLabs architecture (Figure 2) consists of a Client that a student uses to specify the experiment they want to run. The student's experiment specification is then passed through the Service Broker to the Lab Server that is connected to the ELVIS platform. The ELVIS will have the actual circuit built on the breadboard and the user's experiment will be run on the circuit and sampled output data will be returned to the user.

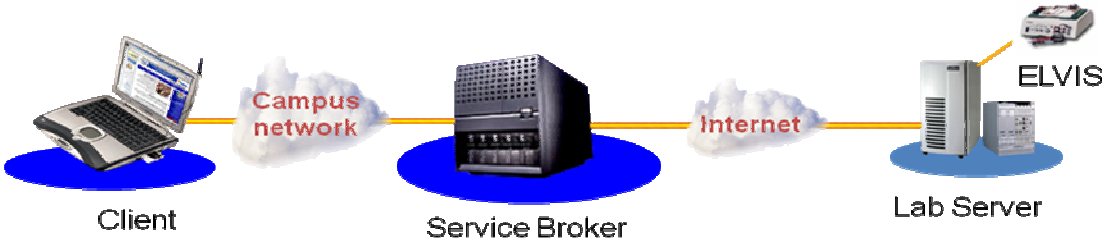


Figure 2: The iLabs architecture

To enable a larger variety of experiments to be run on the ELVIS, the next step is to extend the functionalities available on the ELVIS to the end user. The ELVIS platform offers quite a few capabilities that are still not exposed to the user. These include:

- Digital Multimeter
- Dynamic Signal Analyzer
- Digital Reader
- Digital Writer
- Impedance Analyzer

This approach is possible because National Instruments provides access to the lower level source code to control various functionalities on the ELVIS. They also expose higher level APIs, called Express Virtual Instruments (VIs), which can be easily used by inputting various values into the already programmed interface. To expose each new functionality, the steps to be taken are similar. Changes need to be made to the following areas:

1) The Java User Client has to be programmed to have the UI for the new functionality.
2) The Experiment Engine in the Lab Server that interprets the UI actions on the client to the ELVIS hardware must be modified.
3) The SQL Database Server must be changed to record the data generated by the new functionality.
4) The LabVIEW VI that controls the ELVIS hardware needs to be updated with the appropriate VI for the additional functionality.

This allows a modular approach to making changes to the current iLab architecture. The various steps mentioned above can be executed and tested separately. This also allows use of various technologies for the different steps. The software that interacts directly with the hardware set up on the ELVIS is the LabVIEW graphical programming language. This is what the Express VI for the various ELVIS modules are written in. The Experiment Engine and the Lab Server are written in .NET. This is typical of web service architectures as shown in Figure 1. The system consists of a client, a service broker and a service provider (lab server). The client UI that the user employs to run an experiment is written in Java to allow for cross platform use. The architecture also uses a SQL database to validate user inputs and for record keeping.

2. DESIGN OVERVIEW

In this sections I will describe the structure of various parts of the iLabs software architecture. This is mainly focused towards giving information relevant to developers wishing to develop on the iLabs ELVIS architecture. After the overview in this section, I will describe how to develop each components in the next section. For reference to the various XML files mentions see Section 4 on XML files.

LABSERVER LABVIEW

National Instruments provides the LabView software for accessing drivers that can be used to control the various hardware features on the ELVIS. LabView is a graphical programming language. You can get more information about the language at www.ni.com. The ELVIS iLab code is arranged in the following module structures that are described on the next page.

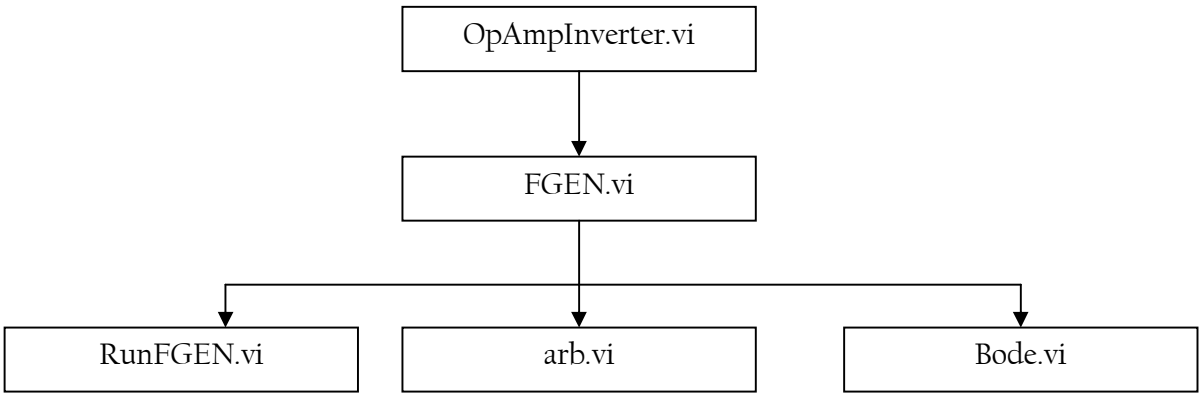


Figure 3: The LabView VI hierarchy

OpAmpInverter.vi

This is main entry point into the Labview code. This is the class called from the compiled DLL from the VB code in the Labserver [Class:PInvoke in OpAmpInverter.vb]. This is just a module that passes user parameters to the underlying FGEN vi.

FGEN.vi

This is the vi that calls the various other hardware features. It calls the function generator and DAQ card (RunFGEN.vi), arbitrary waveform generator (arb.vi) and the bode analyzer (Bode.vi). Only one of the RunFGEN.vi and Bode.vi are run because the bode analyzer uses the function generator to output sine waves of different frequencies to measure the frequency response of the circuit.

RunFGEN.vi

Runs the Function generator to produce the waveform requested by the user and the DAQ card to sample the required analog wave channels.

arb.vi

Runs the arbitrary waveform generator. This generates the required waveform on the DAC0 and DAC1 pins of the ELVIS. The waveform can be a predefined wave like a Square, Triangular, Sine, or Sawtooth wave. The feature also allows other flexible ways of defining a waveform either by a formula or by inputting a file that has the data values for the waveform.

Bode.vi

Runs the Bode Analyzer feature of the ELVIS. This takes in the start and stop frequencies from the user and runs the function generator hardware to sweep the sine waves.

Both the Bode.vi and arb.vi files use Express VIs that are provided in LabView to run the different ELVIS functionalities. More details about this will follow in the development section of the manual.

LABSERVER VISUAL BASIC

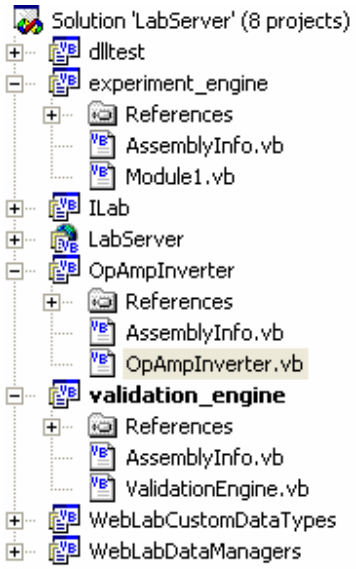


Figure 4: Class structure for the Lab Server VB code

The figure above shows the class structure of the Labserver visual basic code. The main projects are the *experiment_engine*, *OpAmpInverter* and *validation_engine*.

experiment_engine

The execution engine runs in the background at all times while the Lab Server is operational. At a given interval, the execution engine checks the experiment execution queue for new jobs. If one is available, the execution engine de-queues the job The main class file in the experiment engine is the Module1.vb file.

This file has a M*ain()* subroutine that does the following:

1) De-queues submitted jobs from the SQL server. This is done at the following line: `strDBQuery = "SELECT dbo.qm_CheckQueue();"`
2) Once the job is de-queued the method call `ParseExperimentSpec(strExpSpec)is` made to parse the Experiment Specification (*more details in the XML section*) received from the client.

The ParseExperimentSpec method parses the experiment specification (an XML file that contains the experiment parameters chosen by the user). The method stores the values for different instrument parameter in a table called the `functInfoTable.` For example the code below extracts the frequency value specified for the function generator waveform and stores it in the table:

```
'load frequency value
 tempXPath = "/terminal/function/frequency"
 tempNode = xmlTemp.SelectSingleNode(tempXPath)
functInfoTable(instrumentConstant, FUNCT_FREQUENCY) =
Trim(tempNode.InnerXml())
```

Figure 5: Experiment Specification Parser.

Note: `instrumentConstant` and `FUNCT_FREQUENCY` are integer constants.

3) Once all the parameters values are extracted the `runExperiment()` method is called.

This method calls the `RunExperiment()` method in the `Inverter` class defined in the *OpAmpInverter* project with the parameters values stored in the `functInfoTable` table. The `RunExperiment()` method in the `Inverter` class calls the compiled LabView DLL with the specified parameters. The DLL runs the experiment on the ELVIS hardware. Once the experiment is run it returns from the `Inverter` class back to the `runExperiment()` method in the experiment engine with an array of data for graphs that will be displayed to the user. The data points are then put into an XML file called the "Experiment Results" and sent back to the client for display to the user. Finally the execution engine triggers a notification (via the *Notify()* method) to the ServiceBroker saying the data is ready.

OpAmpInverter

The main class file in this project is the *OpAmpInverter.vb* file that defines the `Inverter` class. The `RunExperiment()` method in this class is called from the experiment engine. This method calls the `runExperiment()` method in the PInvoke class that imports the LabView DLL with the parameters passed from the experiment engine. The DLL returns

an interleaved array of the output data back from the LabView code. This array is deinterleaved in the `RunExperiment()` method of the Inverter class, which returns this data back to the `runExperiment()` method of the experiment engine.

validation_engine

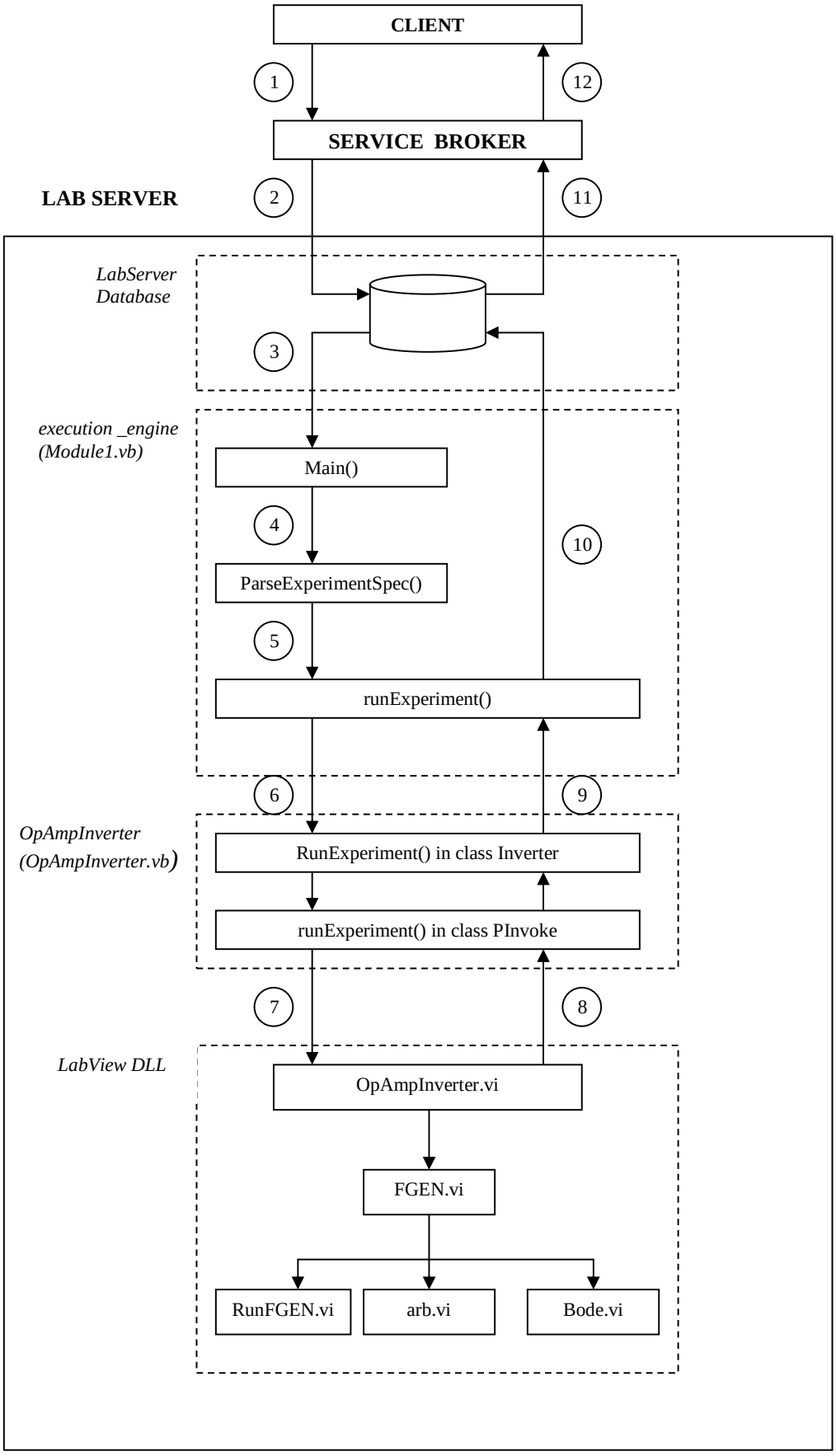
This is the first thing that is called before the job is queued for execution. It checks whether the inputs specified by the user meets the specification set by the designer of the experiment when setting up the assignment. It works the same way as the `ParseExperimentSpec()` method in the execution engine to extract the experiment parameters and checks these values against the values stored in the database.

WebLab Custom Data Types

The most abstract component of the Web Server Layer, this module is composed of a single class which defines a number of custom data types. These types are constructed in order to accommodate certain methods which return collections of data rather than singleton values. Use of these types is limited to interfaces between internal software components within the Web Server Layer.

WebLab Data Managers

The WebLab Data Managers component serves as the primary interface between the Data Persistence Layer of the Lab Server. On the Web Server side, the data managers contain methods defining certain well known interactions with the Data Persistence (Database) Layer and are referenced by the other components within the web server process space.



1) The client sends the "execute" SOAP request to the ServiceBroker with the "Experiment Specification" XML file.
2) A Web service call is made to the appropriate Lab Server and the experiment specification is stored in the Lab Server Database.
3) The experiment engine in its Main() method de-queues the experiment in a FIFO manner and fetches the stored experiment specification.
4) The experiment specification is passed on to the ParseExperimentSpec() method in the execution engine where the XML file is parsed and parameters stored in a table.
5) The runExperiment() method is called. It extracts the parameters from the table stored during parsing.
6) The RunExperiment() method in the Inverter class is called using the parameters extracted this in turn calls the runExperiment() method in the PInvoke class with the same parameters.
7) The PInvoke class imports the compiled LabView DLL and runs the DLL with the parameters when called. The entry point into the LabView code is the OpAmpInverter VI. This runs the experiment on the ELVIS platform.
8) The LabView DLL finishes execution and returns and array of output data to the Inverter class.
9) The Inverter class de-interleaves the output array and returns a 2D array of results to the experiment engine.
10) The experiment engine forms the "Exp Results" XML file and stores it in the database and sends a notification to the Service Broker.
11) & 12) The Service Broker fetches the Exp Result file from DB and forwards it to the client.

Figure 6: The execution cycle showing details of the lab server

LAB SERVER DATABASE

The lab server database has three main tables that you should be aware of:

1) *Setups*: Stores information about each setup including the number of terminals used.

The stored procedure *AddSetup* adds a setup to the list of setups available in the lab server.

2) *SetupTerminalConfig*: Stores information about the terminals present in all the setups. This information include x,y pixel location, terminal type and constraint placed on the terminal.

The stored procedure *AddSetupTerminal* adds a terminal to a setup available in the lab server.

3) *ActiveSetups*: Stores information on what setups are active.

The stored procedure *SetActiveSetup* sets a setup as active.

All these procedure are called by the ASP code of the lab server website.

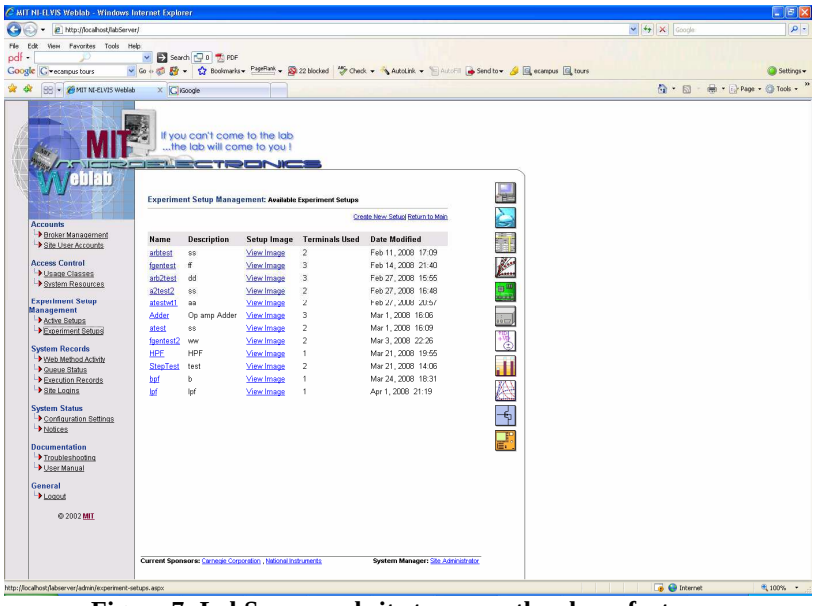


Figure 7: LabServer website to access the above features.

CLIENT (JAVA)

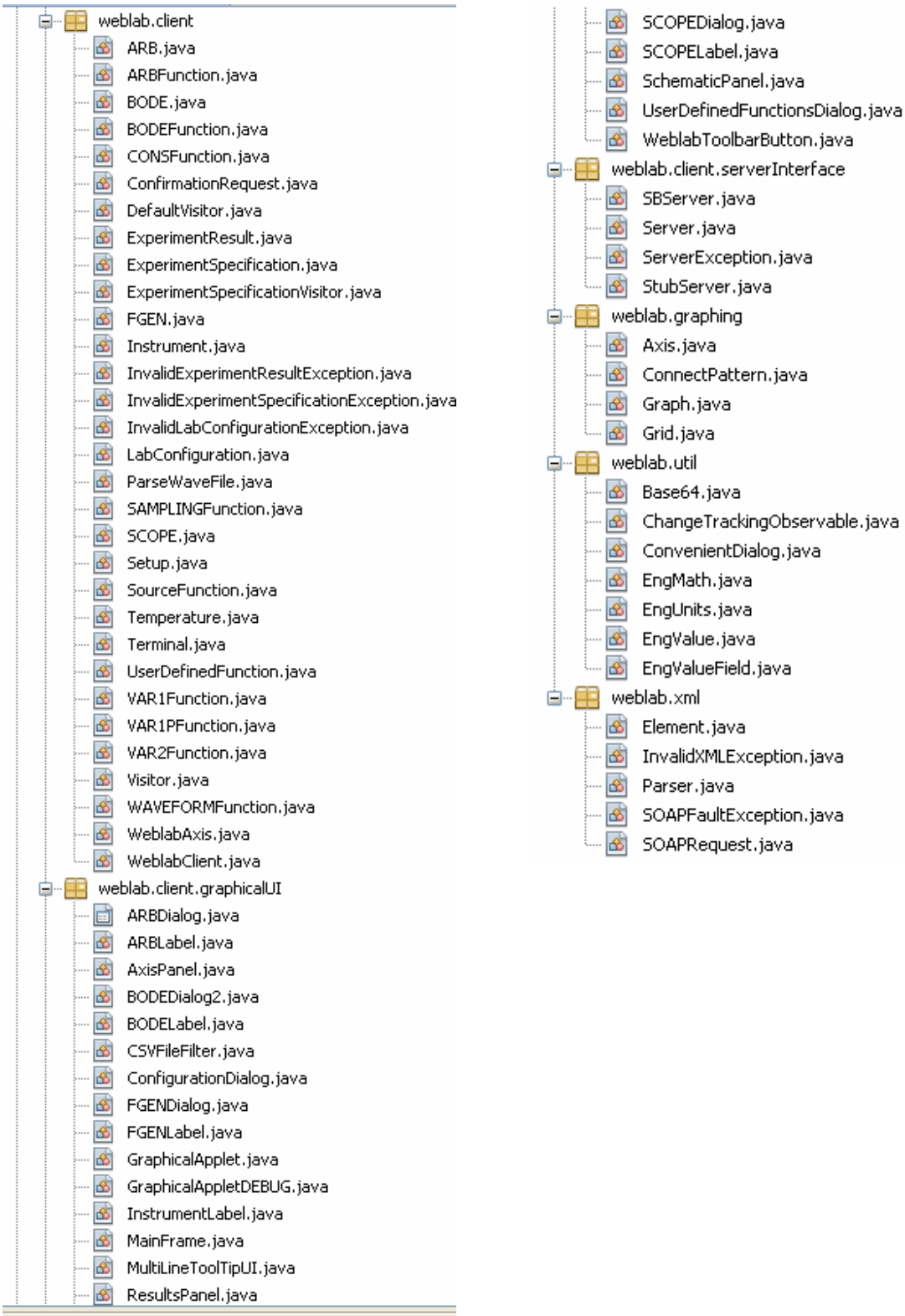


Figure 8: The Client Java code class files

One thing that you should be aware of when understanding the client code is the Visitor design pattern. This pattern is used over and over again throughout the code. Here is an excerpt and diagram from Wikipedia:

"In essence, the visitor allows one to add new virtual functions to a family of classes without modifying the classes themselves; instead, one creates a visitor class that implements all of the appropriate specializations of the virtual function. The visitor takes the instance reference as input, and implements the goal through double dispatch. The idea is to use a structure of element classes, each of which has an accept() method that takes a visitor object as an argument. Visitor is an interface that has a visit() method for each element class. The accept() method of an element class calls back the visit() method for its class. Separate concrete visitor classes can then be written that perform some particular operations.

One of these visit() methods of a concrete visitor can be thought of as methods not of a single class, but rather methods of a pair of classes: the concrete visitor and the particular element class."

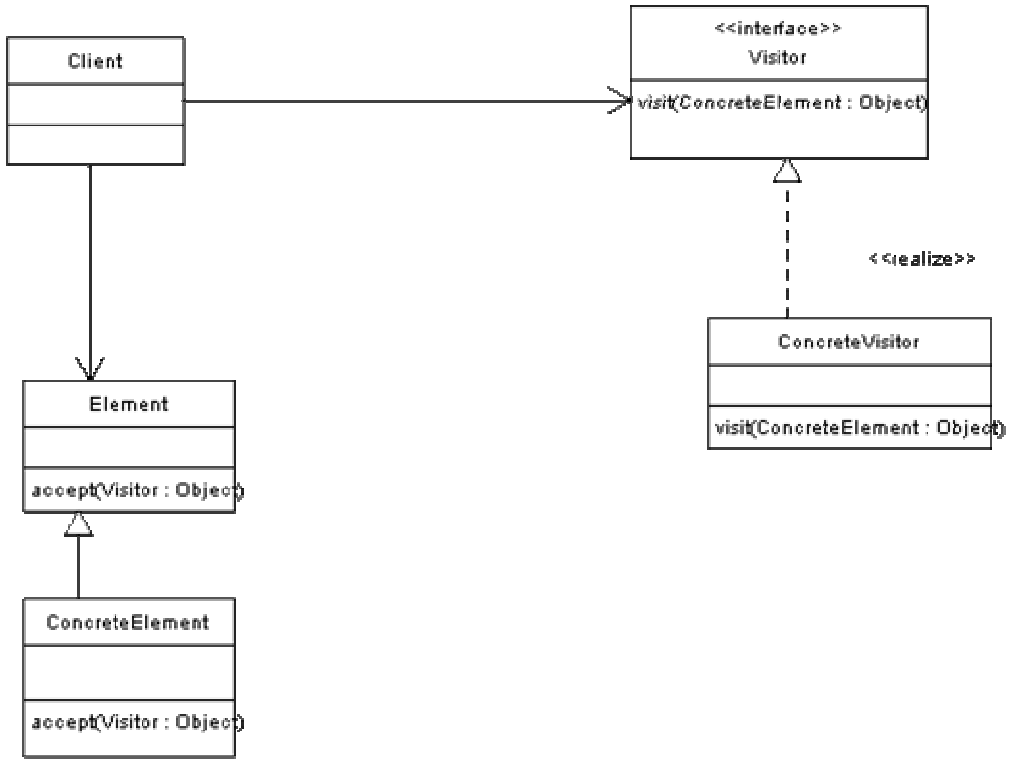


Figure 9: UML of the Visitor Design Pattern

The main flow of information in the Client is as follows:
(italics represent Java class names)

A) When opened the client is initiated through the *GraphicalApplet* class and a Service Broker server (*SBServer*) is associated with the applet. When this happens the loadLabConfiguration() method in *WebLabClient* is called. This method fetches the Lab Configuration XML File (see Section 4) from the lab server through the specified Service Broker through the getLabConfiguration() SOAP call in the *SBServer* class.

B) The Lab Configuration is then parsed by the parseXMLLabConfiguration() method in the *LabConfiguration* class.

From parsing the Lab Configuration a list of *Terminals* are created. A *Terminal* has an instrumentType (for example *Instrument*.FGEN_TYPE or *Instrument*.SCOPE_TYPE) and an instrumentNumber, a label, an xPixelLocation and yPixelLocation to identify where the terminal is located in the setup image.

From the list of Terminals a *Setup* is created which represents the current experiment. Setup has a setupID, a name, a description, an imageURL, and an ordered list of Terminals that are present in the experiment.

C) Once the Lab Configuration has been parsed, the *Setup* is stored in the *ExperimentSpecification* `theSetup` field and the *Instruments* (*FGEN*, *ARB*, *SCOPE*, *BODE*) are created from the *Terminal* information and stored in the `instruments` vector in the same class.

D) Then the *MainFrame* draws the main client elements including the buttons and the menu bars.

F) The *MainFrame* then calls the *SchematicPanel* and the *ResultsPanel*.

G) The *ResultsPanel* class draws the axes for plotting later.

H) The *SchematicPanel* uses setup stored in the `theSetup` in the *ExperimentSpecification* to draw the image of the experiment and the corresponding *InstrumentLabel* for the instruments in the setups.

I) The experiment is ready to be run. When the user clicks on any of the instrument labels the instrument dialog box appears. Each instrument has its own dialog box that users can use to specify parameters of the instruments. For example frequency, amplitude etc for the *FGEN* instrument.

Each *Instrument* has a *SourceFunction* associated with it. This *SourceFunction* (*WAVEFORMFunction* in the case of *FGEN*)is changed when the user specifies values in the dialog box (*FGENDialog* for the case of *FGEN*).

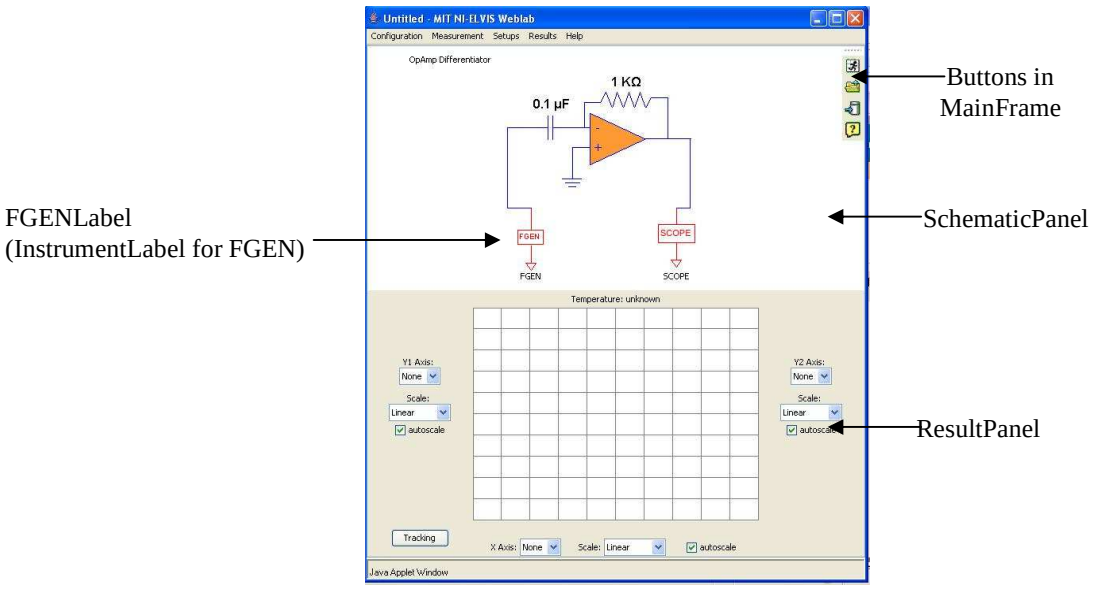


Figure 10: Different parts of the Java Client

J) When the user click on the ‘Run’ button, the Experiment Specification XML document is created by the *ExperimentSpecification* class. This Experiment Specification is sent to the lab server via the execute SOAP call is *SBServer*.

The job is now submitted to the lab server and the events in *Figure 6* take place.

K) When the job finishes executing the RetrieveResult SOAP request is used to get the Experiment Result XML file from the lab server.

L) The parseXMLExperimentResult method in *ExperimentResult* is used to parse the Experiment Result XML file.

M) The data is displayed using the *Axis*, *ConnectPattern*, *Graph* and *Grid* classes in the graphing package.

The best way to visualize the Client code structure is to look at the UML diagrams for different parts of the code. I have included UML diagrams starting on the next page. Please read the captions for the diagrams to understand what is represented.

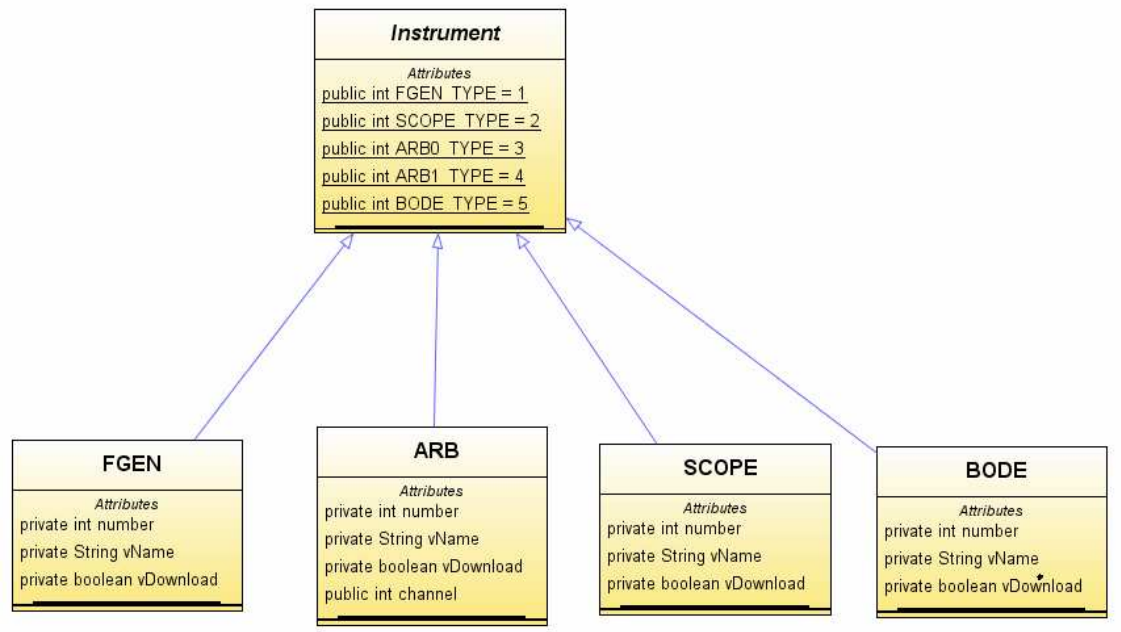


Figure 11: Each feature on the ELVIS is an Instrument. All features inherit from the Instruments class.

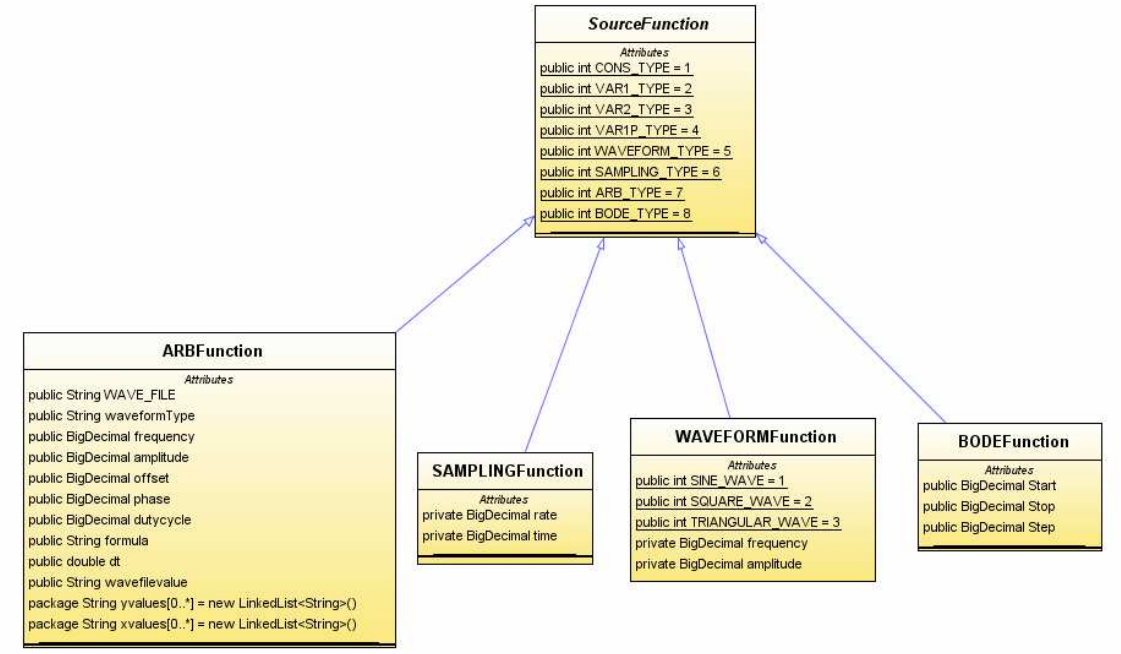


Figure 12: Associated with each instrument is a function. Functions for each instrument are sub classes of the SourceFunction class.

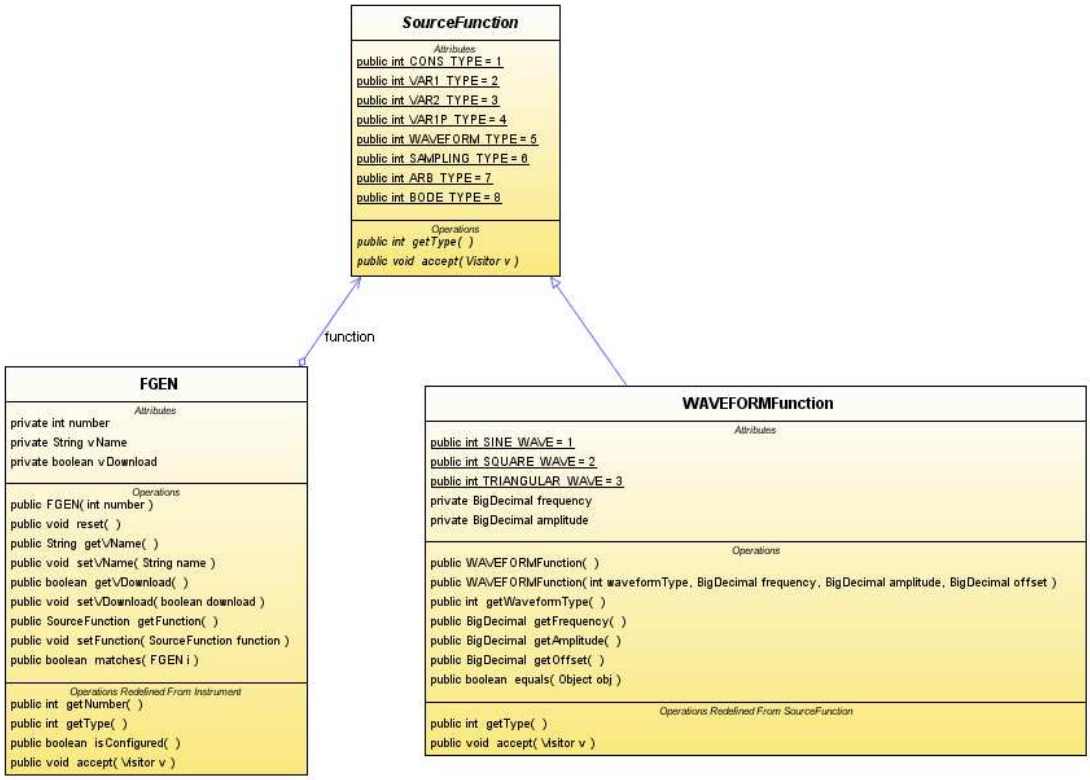


Figure 13: This diagram uses the FGEN(Function Generator) Instrument to show the each instrument type has a SourceFunction associated with it. In the case of the FGEN it is the WAVEFORMFunction.

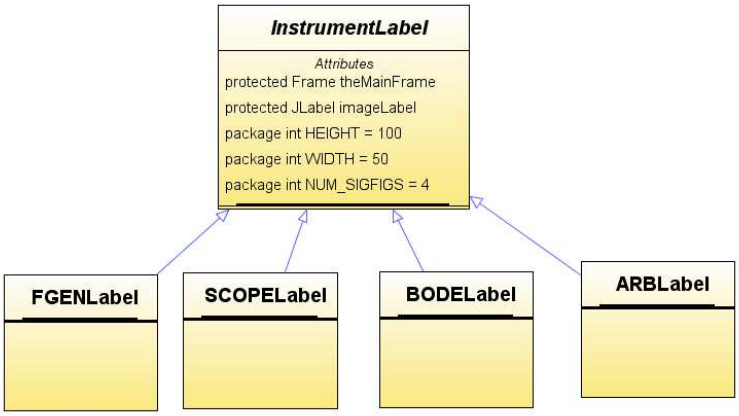


Figure 14: For display purposes each instrument also has a label associated with it. The label has the icon and the name of the instrument that will be displayed to the user.

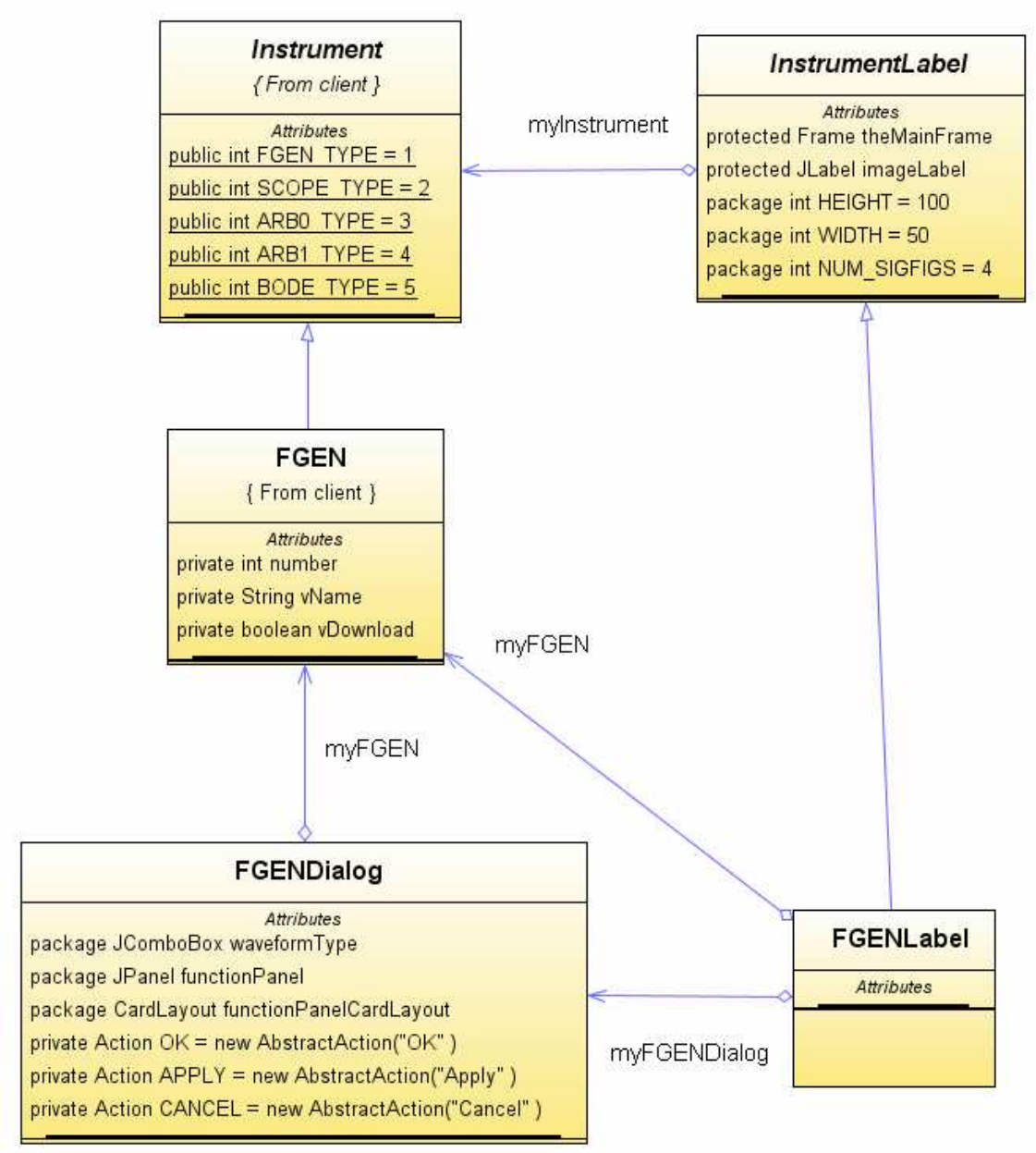


Figure 15: A diagram that uses the FGEN as an example to show that each instrument has a Dialog box associated with it (FGENDialog). The Dialog box is the user interface that is used to specify parameters of the instrument.

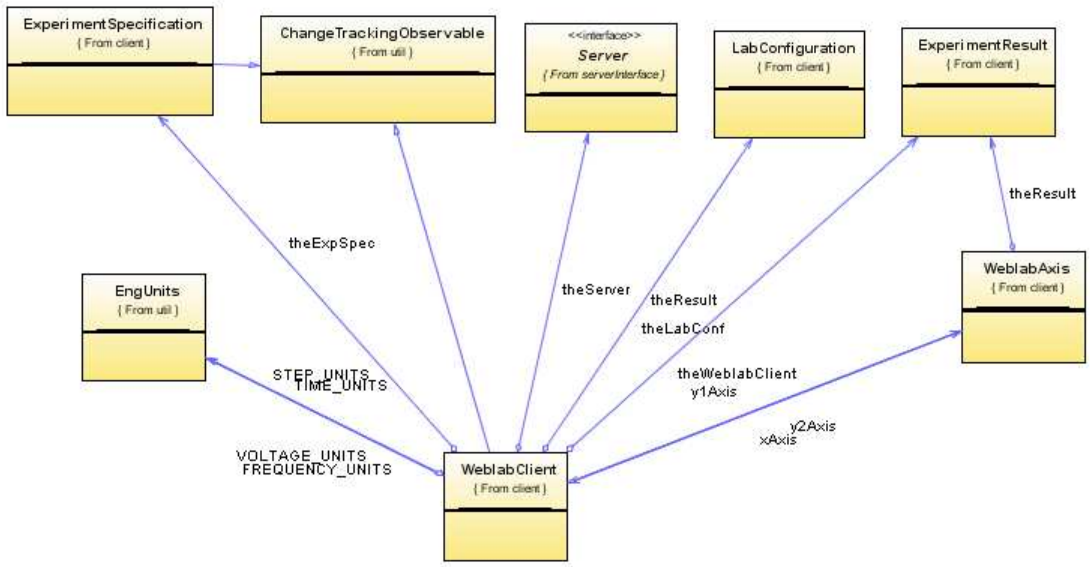


Figure 16: The WebLabClient is the central class of the client. It has a Service Broker server (Server class) associated with it. It also has the LabConfiguration, ExperimentSpecification and ExperimentResult class that define the 3 XML files used. Also has the WebLabAxis fields for plotting.

The best way to understand how to develop the client is to look at the next section as I walk through the changes I made to the client to understand in more detail the various classes represented in the UML diagrams and data flow above.

3. DEVELOPMENT GUIDE

I will walk through all the development steps that I went through in adding the Bode Analyzer (Bode) and the Arbitrary Waveform Generator (Arb) features in each of the software layers. I think this is the best way of showing the development process. Following similar steps will help you as you add more features. I have described the parts in the order that I feel is the best sequence to follow in the development.

LABSERVER LABVIEW

The best place to start in the development process is with the LabView part of lab server because this directly interacts with the hardware and is the easiest place to start debugging the hardware to make sure everything works. The best way to add a new feature is to start by creating a new VI for it. Then you can think about where to place the your newly created VI in the current framework of VIs. Most of the features on the ELVIS have an associated Express VI, and this is usually the best and easiest way to interact with the hardware.

1) MAKING THE VI FOR THE NEW FUNCTIONALITIES

For the arbitrary waveform generator (arb.vi) we wanted to add additional input capabilities to the ELVIS architecture. The ELVIS provides the DAC0 and DAC1 pins for the generating two independent arbitrary waveforms. The capabilities that were supported were for generating arbitrary waveforms were:

1) Square wave
2) Sine wave
3) Sawtooth wave
4) Triangular wave
5) Generating a wave from a formula
6) Generating a wave from a file data file

For generating the Square, Sine, Sawtooth and Triangular waveforms LabView provides VIs that given the wave parameters it simulates the waveforms, and samples them according to the desired sampling rate. The parameters include: Frequency, Amplitude, Phase and Offset. An additional parameter for the Square wave is the duty cycle. There is also an input for the sampling rate and number of samples to be takes from the simulated wave. The variables for the parameters need to be created in the VI.

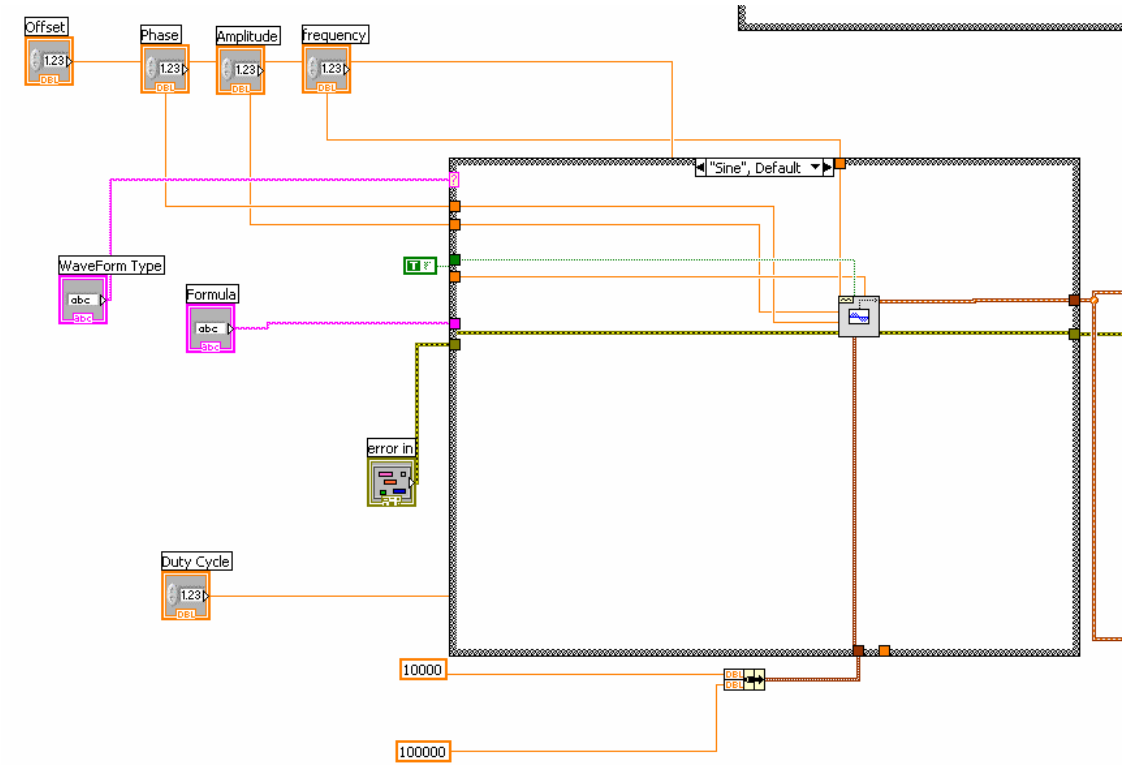


Figure 17: Part of the arb vi showing the sine wave simulator and the input parameters it takes.

For the generating the waveform from a formula there is also a VI that accepts a formula in MatLab syntax and parses the formula to generate the desired waveform.

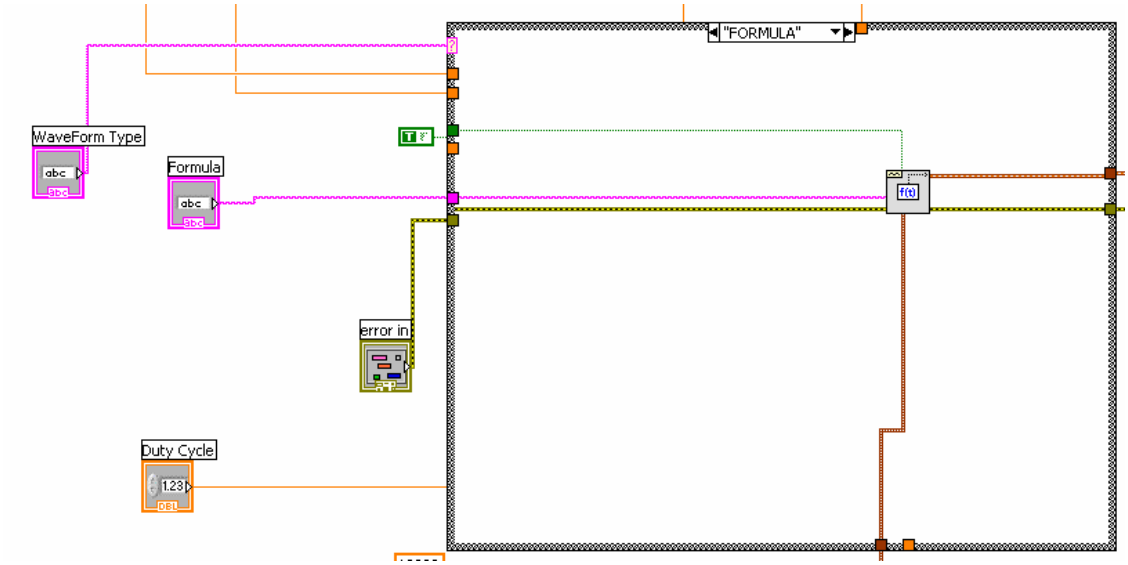


Figure 18: The formula generator VI

For the generating the waveform from file LabView provides the “Build Waveform” VI that takes an array of y values and dt and generates a wave from it.

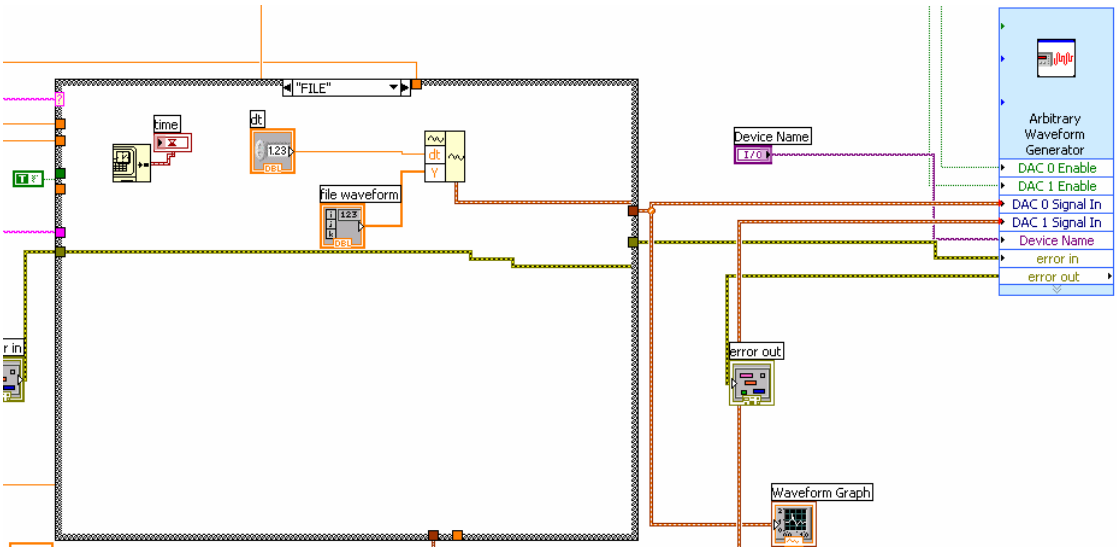


Figure 19:Generating a waveform from an array of y values

All the above options are put into a case statement that uses a string as a selector for which option to use.

For the Bode Analyzer VI (Bode.vi) it was a simpler task to make the VI because the Express VI (shown in *Figure 4*) only requires three parameters: the start frequency, stop frequency and the steps per decade.

2) EXPOSING THE TERMINALS FOR THE VI

After making the VI the next step is to expose the terminals that will be the inputs and output for the VI. To do this you go to the front panel of the VI. Then click on the icon in the top left on the window and choose ‘Show Connector’.

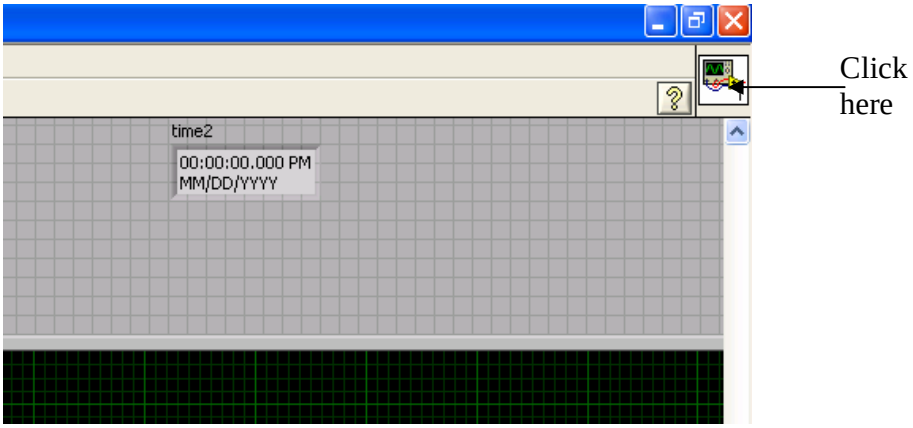


Figure 20: Front panel of the VI

Then you can click on an empty terminal and assign it a variable by clicking on the variable. The colored terminals are assigned terminals and the white ones are free terminals. If you need to add more terminals you can right click and choose ‘Add Terminal’

Figure 21: Connector diagram

3) PLACING THE VI IN THE CURRENT FRAMEWORK

The next step is to add the created VI in the current VI framework. This will depend on how you want the functionalities to be run and if there are any resource conflicts. I placed the ARB VI (arb.vi) in the FGEN.vi file to be run in a new thread. In this way the FGEN.vi file calls both the RUNFGEN.vi file to run the function generator and the arb.vi file to run the arbitrary waveform generator. This way the function generator and the arbitrary waveform generator can be run simultaneously.

For the Bode Analyzer VI (Bode.vi) I placed the VI in a case statement so that only one of RUNFGEN.vi or Bode.vi is run as they both use the function generator hardware.

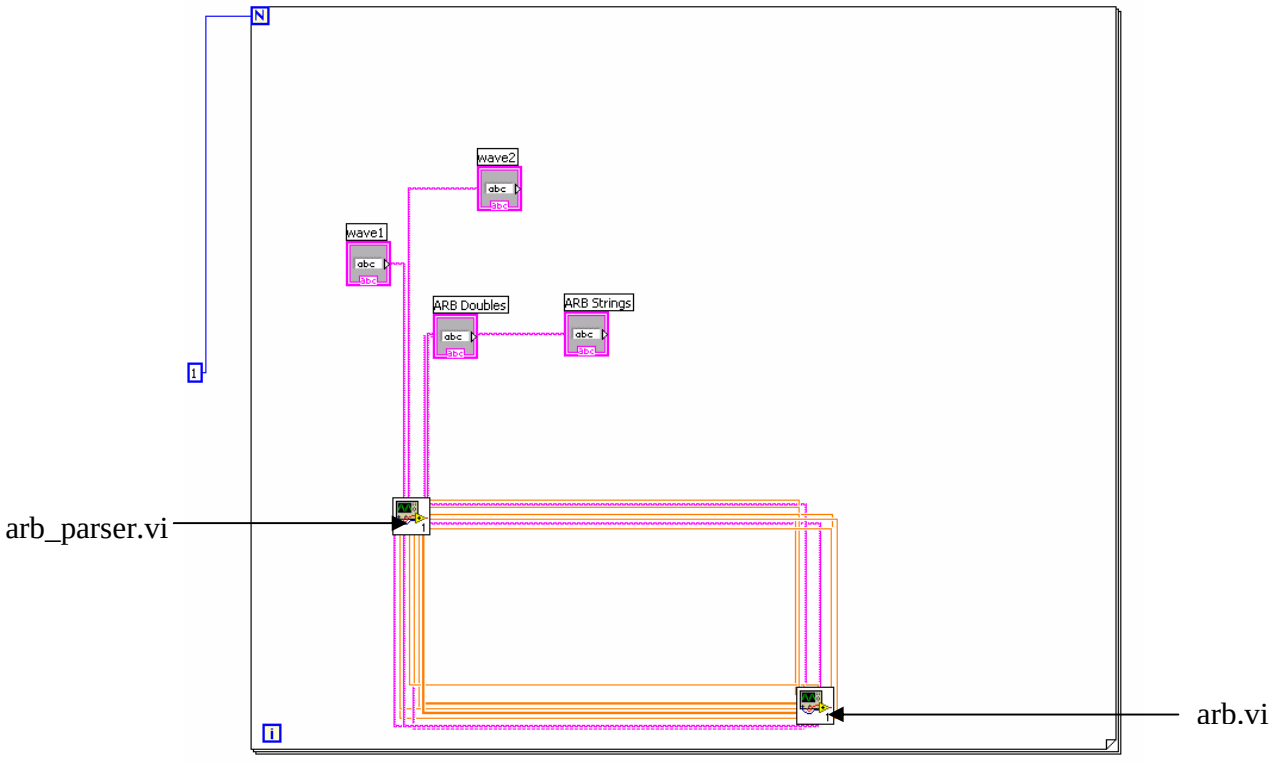


Figure 22: The ARB VI placed in FGEN

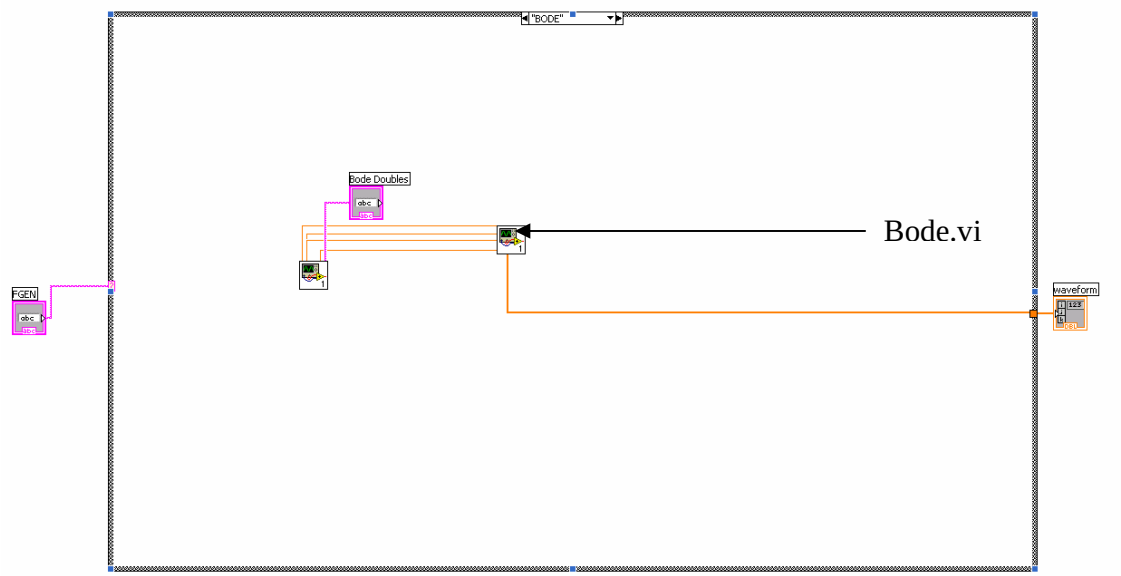


Figure 23: Bode Analyzer VI placed in a case statement

I also added a parser for each functionality as there is limited number of terminals that can be exposed to the lab server. This way for each instrument I only pass one string from the lab server which can be parsed to get the required parameters for running the VI.

4) EXPOSING THE TERMINALS TO OPAMPINVERTER.VI

The next step is to make the variables for the new VIs in the FGEN file and expose the variables in the FGEN.vi and OpAmpInverter.vi file.

5) ADDING A SAMPLING PORT

If you need to capture an additional waveform you can double click on the DAQ assistant VI in RUNFGEN and add an extra analog channel to sample. You will also need to change the interleaved array connected to the DAQ assistant.

6) COMPILIING THE DLL

Once the new terminals have been exposed in the OpAmpInverter.vi which is the entry point file you can compile it into a DLL. In LabView version 8.2, the best way to do this is to add or your VI files to a project. Then you can right click on 'Build Specifications' and choose 'New->DLL'. Add OpAmpInverter.vi into the 'Exported VI' and name sure the variables are in the same order as called from the lab server (*Figure 25*).

7) PLACING THE DLL IN THE LABSERVER

The final step is to place the compiled DLL in the location specified at the lab server in the OpAmpInverter.vb file in the PIvoke class.

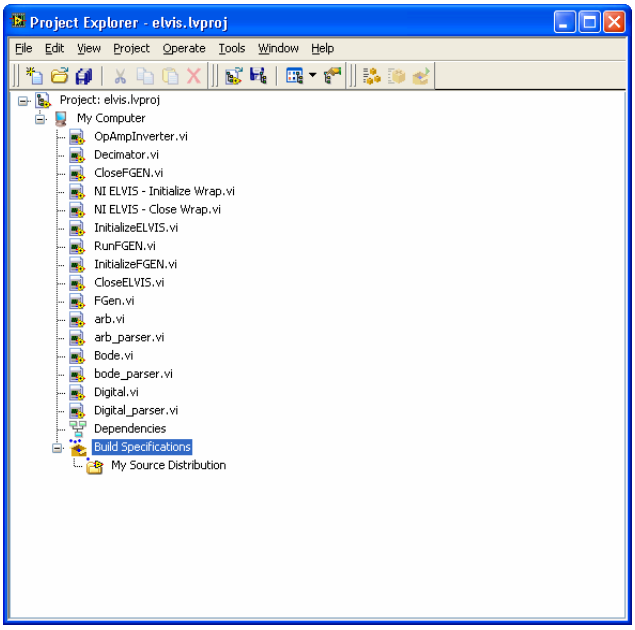


Figure 24: LabView project

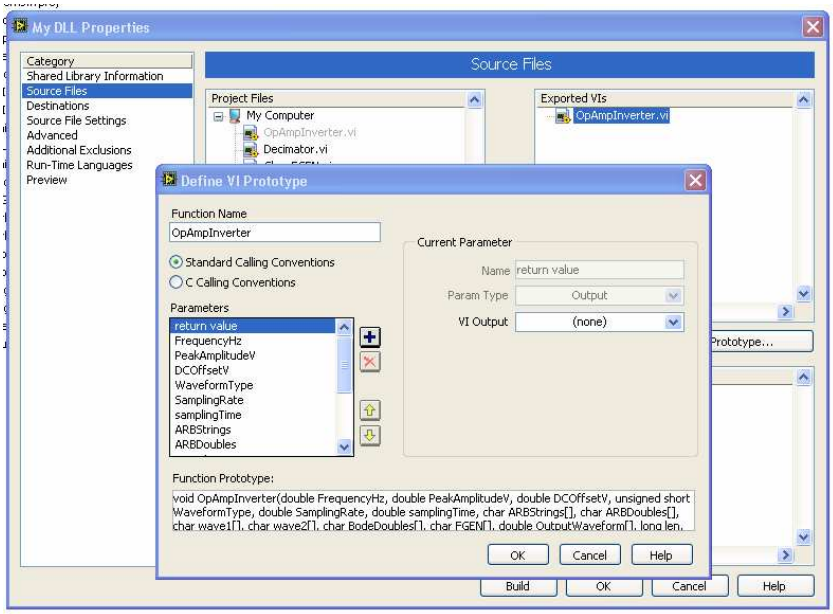


Figure 25: Compiling the DLL

Debugging: Use the “Display Msg” Express VI to put print outs for testing

Figure 26: "Display Msg" Express VI

CLIENT

For all the steps below you can look at existing classes to give you a better idea of what to change. I used the NetBeans IDE for development. To setup it up copy the 'weblab' and the 'img' folder of the client into the 'src' folder of a newly created NetBeans project.

1) The first step in adding a new functionality is creating an instrument for the feature. For the Arbitrary Waveform generator (ARB) and Bode Analyser (BODE) feature a new class was created that extends the Instrument class. You also need to add an instrument identifier number in the Instrument class for the new instrument. You can look at the ARB or FGEN file for examples. Following the same pattern will work.

```
public class ARB extends Instrument {
    private int number;
  private String vName;
  private boolean vDownload;
  private SourceFunction function;
  public int channel;

    /** Creates a new instance of ARB */
    public ARB(int number, int channel) {
        this.channel=channel;
     this.number = number;
    // pre-initialize things that might be null
    this.vName = "";
    this.function = new ARBFunction();
    // initialize everything here
    this.reset();
    }
```

Figure 27: The ARB instrument

2) Then make the function for your instrument by extending the SourceFunction class. The function for your instrument stores the information parameters for your instrument. In my case the ARBFunction is the function for the ARB instrument and stores information like the waveform type selected by the user and the parameters like frequency, amplitude, phase etc associated with the waveform. If the user wants to plot a formula it also stores the wave function. Finally to use a file for generating a wave, there is a parser that parses the file loaded and extracts the dt value and the associated y values that are stored in the `wavefilevalue` field. The BODEFunction similarly has the fields to store the start, stop

frequencies and the steps per decade to be used for the bode analysis. You also need to add a function identifier number in the SourceFunction class for the function.

```
public class ARBFunction extends SourceFunction{



          public  String WAVE_FILE;

        public String waveformType;
        public BigDecimal frequency;
        public BigDecimal amplitude;
        public BigDecimal offset;
        public BigDecimal phase;
        public BigDecimal dutycycle;
        public String formula;
        public  double dt;
        public String wavefilevalue;


    /** Creates a new instance of ARBFunction */
    public ARBFunction() {
        this("SINE", BigDecimal.valueOf(0), BigDecimal.valueOf(0),BigDecimal.valueOf(0), BigDecimal.valueOf(0),
    }
```

Figure 28: The ARBFunction

3) After that next step is to make the instrument label for the feature. This extends from the *InstrumentLabel* class. This class has a chooseImageName() method that specifies the name of the image to be used for the instrument. You will need to draw an image for the instrument and put the image in the 'img' folder and put the name of the image in the method. You can also add a case for your instrument in the updateToolTip() method in the InstrumentLabel class to show a tool tip.

```
public class ARBLabel extends InstrumentLabel {

     private ARB myARB;
  private final ARBDialog myARBDialog;

   /** Creates a new instance of ARBLabel */
    public ARBLabel(Frame theMainFrame, ARB myARB) {


    super(theMainFrame, myARB, "ARB");

    this.myARB = myARB;

    // create a SCOPE dialog that can be opened by clicking on this
    myARBDialog = new ARBDialog(theMainFrame, myARB);

    this.addMouseListener(new MouseAdapter() {
        public void mouseClicked(MouseEvent e) {
          // Point location = new Point(e.getX()+10, e.getY()+10);
          // mySMUDialog.setLocation(getMousePointerAbsolute(location));
          myARBDialog.setVisible(true);
        }
      });
```

Figure 29: ARBLabel

4) Next thing you need to create the dialog user interface that will be used to modify the instrument. This will appear when the label for that instrument is placed. The best way of doing this is to look at FGENDialog class. The user interface can be easily made by copying the code from that class. If you need to make a very different UI I suggest you use the swing GUI builder in NetBeans. The dialog box will have an *OK* and an *Apply* button. When these buttons are pressed you need to handle the action by exporting the values chosen by the user to the source function. Of course the details of this will vary on how the user input was asked for but the basic idea is to gather all the information specified by the user and call the constructor of the source function and assign the function to your instrument. You also need to create the ImportValuesVisitor class in the same file to import pre-existing instrument values to the dialog box. As seen in *Figure 31* in the second last line f.setFunction(…) is called.

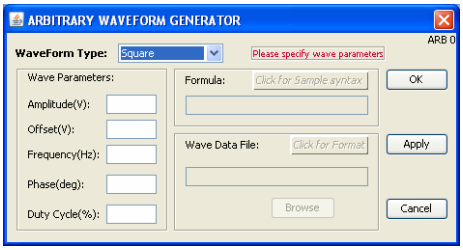


Figure 30: ARBDialog

```
protected void exportValues(FGEN f)
{
  f.setVName("Vin");
  f.setVDownload(true);

  // export source function
  // get the waveformType selected

  int wfType = 0;
  if (waveformType.getSelectedItem().equals("SINE")) {
      wfType = WAVEFORMFunction.SINE_WAVE;
   }else if (waveformType.getSelectedItem().equals("SQUARE")) {
      wfType = WAVEFORMFunction.SQUARE_WAVE;
   }else if (waveformType.getSelectedItem().equals("TRIANGULAR")) {
      wfType = WAVEFORMFunction.TRIANGULAR_WAVE;
   }else {
      //assert false;
      throw new Error("impossible waveformType");
   }

  f.setFunction(new WAVEFORMFunction(wfType, frequency.getValue().toBigDecimal(), amplitude.getValue().toBigDe
  // notify observers that myFGEN has changed
  myFGEN.notifyObservers();
}
```

Figure 31: Export value function for FGEN

5) Add a case for the new instrument in the drawVariableNames(final Graphics g) in the SchematicPanel class to draw instrument name.

6) To handle the parsing of the lab configuration you need to add a case for the instrument in the parseXMLLabConfiguration() method in the LabConfiguration class. This just assigns a type to the instrument so that it is recognized when the instrument in created in the ExperimentSpecification class.

```
            else if (instrumentTypeName.equals("BODE"))
        {
            instrumentType=Instrument.BODE_TYPE;

        }

    else if(instrumentTypeName.equals("ARB0")||instrumentTypeName.equals("ARB1"))
    {
            instrumentType=0;
        if(instrumentTypeName.equals("ARB0"))
        {
            instrumentType=Instrument.ARB0_TYPE;
        }
        else if (instrumentTypeName.equals("ARB1"))
        {
            instrumentType=Instrument.ARB1_TYPE;

        }
```

Figure 32: Adding a case for parsing the lab configuration file.

7) You have to handle the creation of the Experiment Specification document. To do this you need to add the methods visit*Instrument* and visit*InstrumentFunction* to the ExperimentSpecification class. As shown below the visitBODE and visitBODEFunction methods are used to form the Experiment Specification for the bode analyzer. The same XML tags will be used to parse the specification in the lab server. In the visitBODEFunction (*Figure 33*) you can see that I extract the start, stop frequencies and steps from the BodeFunction *f*.

```
public final void visitBODE(BODE f)
{
  // begin terminal
  sb.append("<terminal instrumentType=\"BODE\" instrumentNumber=\"");
  sb.append(f.getNumber());
  sb.append("\">");

  // vname
  if (f.getVDownload())
      sb.append("<vname download=\"true\">");
  else
      sb.append("<vname download=\"false\">");
      sb.append(f.getVName());
      sb.append("</vname>");

  // function
  f.getFunction().accept(this);

  // end terminal
  sb.append("</terminal>");
}
```

Figure 33: visitBODE in ExperimentSpecification

```
    public final void visitBODEFunction(BODEFunction f)
  {
    sb.append("<function type=\"BODE\">");



    sb.append("<startfrequency>");
    sb.append(f.Start);
    sb.append("</startfrequency>");
    sb.append("<stopfrequency>");
    sb.append(f.Stop);
    sb.append("</stopfrequency>");
    sb.append("<step>");
    sb.append(f.Step);
    sb.append("</step>");
    sb.append("</function>");
  }
```

Figure 34: visitBODEFunction in Experiment Specification

8) Finally you need to make sure the visitors for the new instrument and its functions are defined in the Visitor and DefaultVisitor class.

Debugging: Run the *GraphicalAppletDEBUG* file to debug the client independently. You can then use print statements and watch variables using an IDE like NetBeans or Eclipse.

LABSERVER VISUAL BASIC

This development was done using Microsoft Visual Studio 2003. Open the 'LabServer' solution file to view all the projects included in the lab server.

1) The first thing to modify is the way the Experiment Specification is parsed. This is in the `ParseExperimentSpec` method in the Module1.vb file in the experiment_engine project. The way this method does the parsing is by first parsing all the instruments and placing them in a table. This table is called the `termInfoTable`. The table has a field for the `TERM_INSTRUMENT` that specifies the type of instrument and `TERM_FUNCTION` that specifies the name of the function associated with the instrument. Based on the `TERM_INSTRUMENT` an `instrumentConstant` is given to the instrument to identify its type. So this is the first thing you have to change by adding a case for your new instrument. You will also need to define a new constant like `FGEN_FUNCT` for the new instrument. This constant is used to index the table to retrieve the result later. Just give it a unique ID at the beginning of the Module1.vb file.

```
Select Case termInfoTable(loopIdx, TERM_INSTRUMENT)
                Case "FGEN"
                    instrumentConstant = FGEN_FUNCT
                    FGEN_record = loopIdx
                Case "SCOPE"
                    instrumentConstant = SCOPE_FUNCT
                    SCOPE_record = loopIdx
                Case "ARB0"
                    instrumentConstant = ARB_FUNCT
                    ARB_record = loopIdx
                Case "ARB1"
                    instrumentConstant = ARB2_FUNCT
                    ARB2_record = loopIdx
                Case "BODE"
                    instrumentConstant = BODE_FUNCT
                    BODE_record = loopIdx

            End Select
```

Figure 35: Adding a case for the instrument

2) After that the next step is to add a case for reading the function type from the `termInfoTable`. This can be done in the select case shown below by adding a case with the function name you have used. Before you do this however you will need to define indexing constants for the different parameters in a function. These indexes are used to store the parsed values from the Experiment Specification and store them in the `functInfoTable`. These values can then be retrieved later using the same indexes. For example in the figures below the value at `functInfoTable(FGEN_FUNCT, FUNCT_WAVEFORMTYPE)` will be the type of waveform specified by the user.

```
'function information fields
    Const FGEN_FUNCT As Integer = 0

    Const FUNCT_OFFSET As Integer = 0
    Const FUNCT_WAVEFORMTYPE As Integer = 1
    Const FUNCT_FREQUENCY As Integer = 2
    Const FUNCT_AMPLITUDE As Integer = 3
    Const FUNCT_SAMPLINGRATE As Integer = 4
    Const FUNCT_SAMPLINGTIME As Integer = 5
```

Figure 36: Constant for FGEN for indexing the table

```
Select Case termInfoTable(loopIdx, TERM_FUNCTION_TYPE)
                Case "WAVEFORM"
                    'load waveformTYpe value
                    tempXPath = "/terminal/function/waveformType"
             tempNode = xmlTemp.SelectSingleNode(tempXPath)
                 Select Case Trim(tempNode.InnerXml())
                       Case "SINE"
                           functInfoTable(instrumentConstant,
FUNCT_WAVEFORMTYPE) = 0
                       Case "TRIANGULAR"
                           functInfoTable(instrumentConstant,
FUNCT_WAVEFORMTYPE) = 1
                       Case "SQUARE"
                           functInfoTable(instrumentConstant,
FUNCT_WAVEFORMTYPE) = 2
                    End Select
                    Debug.WriteLine("waveformType=" &
Trim(tempNode.InnerXml()))

                    'load frequency value
 tempXPath = "/terminal/function/frequency"
  tempNode = xmlTemp.SelectSingleNode(tempXPath)
  functInfoTable(instrumentConstant, FUNCT_FREQUENCY) =
Trim(tempNode.InnerXml())
```

Figure 37: Case statement for the FGEN function "WAVEFORM"

3) Once the parsing is done the next step is to change the `runExperiment()` method. This where the parsed value stored for different parameters in the table are retrieved to be passed to the LabView DLL. You need initialize variables for the parameters of your new function and assign them to the values stored in the `functInfoTable`.

```
Dim frequency As Double = functInfoTable(FGEN_FUNCT, FUNCT_FREQUENCY)
Dim amplitude As Double = functInfoTable(FGEN_FUNCT, FUNCT_AMPLITUDE)
Dim offset As Double = functInfoTable(FGEN_FUNCT, FUNCT_OFFSET)
Dim waveformType As Double = functInfoTable(FGEN_FUNCT, FUNCT_WAVEFORMTYPE)
```

Figure 38: Retrieving parameter values for the FGEN instrument

4) After that make sure you pass the new variables along to the `RunExperiment()` `method of the OpAmpInverter.vb` class in the correct order.

5) Change the call `PInvoke.runExperiment` in the `RunExperiment()`method to include the new parameters.

6) Change the call to the LabView DLL from the PInvoke class to include the new parameters and make sure they are in the same order as specified in compiling the DLL (*Figure 25*)

```
  Private Class PInvoke


<DllImport("C:\\Inetpub\\wwwroot\\LabServer\\ExperimentSetups\\wrappers\\OpA
mpInverter\\labview\\My DLL\\SharedLib.dll", EntryPoint:="OpAmpInverter",
CallingConvention:=CallingConvention.StdCall)> _
            Public Shared Function runExperiment(ByVal Frequency As Double,
ByVal PeakAmplitude As Double, ByVal DCOffset As Double, ByVal WaveformType
As Short, ByVal SamplingRate As Double, ByVal SamplingTime As Double, ByVal
Arb_Strings As String, ByVal Arb_Doubles As String, ByVal Arb_yvalues As
String, ByVal Arbsec_yvalues As String, ByVal BODE_Doubles As String, ByVal
fgen As String, ByRef waveform As Double, ByVal len As Long, ByRef errorOut
As TD1) As Integer
            End Function)
```

Figure 39: Call to LabView DLL from VB

7) If you add another channel to sample you will need to modify the following code in the `RunExperiment()` method to deinterleave the output array correctly.

```
            Dim j, k As Integer
            For j = 0 To len - 1
                k = j Mod 3
                If k = 0 Then
                    vin(j / 3) = waveform(j)
                ElseIf k = 1 Then
                    vout((j - 1) / 3) = waveform(j)
                ElseIf k = 2 Then
                    arb((j - 2) / 3) = waveform(j)
                End If
```

Figure 40: Deinterleaving the output array

8) Finally when the code returns after executing the DLL to the `runExperiment()` method of Module1.vb, make sure you change the way the Experiment Result XML file is created. The code below shows where the core part of the file is created `strResult()` is an array of strings each entry being a concatenation of all the values of a waveform.

```
strXMLExpResult = strXMLExpResult & "<datavector name='TIME' units='s'>"
& strResult(0) & "</datavector>"
                strXMLExpResult = strXMLExpResult & "<datavector name='"
& triggerchannel & "'units='V'>" & strResult(1) & "</datavector>"
                strXMLExpResult = strXMLExpResult & "<datavector name='"
& secchannel & "'units='V'>" & strResult(3) & "</datavector>"
                strXMLExpResult = strXMLExpResult & "<datavector
name='VOUT' units='V'>" & strResult(2) & "</datavector>"
                strXMLExpResult = strXMLExpResult &
"</experimentResult>"
```

Figure 41: Experiment Result XML file is created

9) You will also need to change the validation engine. The validation engine also parses the Experiment Specification in a similar way as the experiment engine. Hence you can make the same changes as you did in the experiment engine to the `parseXMLSpec()`method of the validation engine. Then you have to change the `experimentValidator()` method. This basically loops through each of the function in the Specification and makes sure that the parameters are within the requirement specified by the developer of the experiment.

```
Case "FGEN"
                        Select Case UCase(termInfoTable(loopIdx,
TERM_FUNCTION_TYPE))
                            Case "WAVEFORM"
                                'validate waveformType, frequency, amplitude
and offset values against
                                ' values stored in the database
                                'FREQUENCY
                                If functInfoTable(FGEN_FUNCT, FUNCT_FREQUENCY)
= "" Or Not IsNumeric(functInfoTable(FGEN_FUNCT, FUNCT_FREQUENCY)) Then
                                    Return "Error - A numeric frequency value
for FGEN must be supplied."
                                    Exit Function
                                End If
```

Figure 42: Validation check for FGEN

Debugging: You can start the experiment engine in debug mode by right clicking on the *experiment_engine* project and going to Debug->Start new instance. In debug mode you can place breakpoints and watch variables.

LABSERVER ASP WEBSITE AND DATABASE

Now you are done changing the code for the whole architecture. Now you need to change the Lab Server administration pages so that your new functionality is available to be seen by however is making the labs. This can be done by modifying *experiment-setups.aspx* page. Things that should be changed here include adding the new instrument to the drop down lists of available instrument, adding constraint fields for your new instrument and adding the new instrument and its constraints to the lab server database that provides permanent storage.

The code below shows where you can add new features and their new constraints. Once you put an HTML tag for it here you will need to change the "Create Terminal" action to incorporate the new constraints.

```
    <td>
        <font class="regular"><b>Instrument:</b>
            <asp:DropDownList ID="ddlNewTermInstrument" Runat="Server">
                <asp:ListItem Value="FGEN">FGEN</asp:ListItem>
                <asp:ListItem Value="SCOPE">SCOPE</asp:ListItem>
                <asp:ListItem Value="ARBO">ARBO</asp:ListItem>
                <asp:ListItem Value="ARB1">ARB1</asp:ListItem>
                <asp:ListItem Value="BODE">BODE</asp:ListItem>
            </asp:DropDownList>
        </font>
    </td>
</tr>
<tr>
    <td>
        <font class="regular"><b>Horizontal Location (pixels):</b>
            <asp:TextBox ID="txtNewTermXLoc" Columns="3" MaxLength="5" Runat="Server" />
        </font>
    </td>
    <td>
        <font class="regular"><b>Vertical Location (pixels):</b>
            <asp:TextBox ID="txtNewTermYLoc" Columns="3" MaxLength="5" Runat="Server" />
        </font>
    </td>
</tr>
<tr>
    <td>
        <font class="regular">Maximum Voltage Amplitude (+/- V)
            <asp:TextBox ID="txtNewTermMaxAmp" Columns="10" MaxLength="12" Runat="Server" />
        </font>
    </td>
```

Figure 43: LabServer administration code

When the lab creator presses "Create Terminal" the following database call is made:

```
strResult = rpmObject.AddSetupTerminal(CInt(e.CommandArgument),
txtNewTermName.Text, CInt(txtNewTermXLoc.Text),……..
```

This call will need to be change together with the SQL `AddSetupTerminal` and the `SetupTerminalConfig` table to add more columns for your new constraints. You will also need to change the 'Check Constraint' of `SetupTerminalConfig` to accept your new instrument.

4. XML FILES

Lab Configuration

This document is created by the lab server. It contains information about which experiments that can be performed on the lab server at a particular time. This information is drawn from the lab server's database. Information contained in this document should be adequate for the client to display information on any experiment that is built on the ELVIS board. This ability will enable us recycle client code for different experiments. The nature of this document captures the model that was described for experiments above.

Experiment Specification XML Document

This document is generated by the client in order to specify the nature of the experiment to be conducted on the lab server. For this version of the ELVIS weblab, this document should contain the following information:

• The ID of the experiment which the user has submitted. This is the same ID that was assigned by the lab server in the Lab Configuration XML document.

• A list of all the component profiles that are used by the user in this experiment. For each component profile, the document should contain a list of all the terminals, each of which should contain a list of other terminals that it is connected to. This cascade of information will help recreate the circuit as it was connected by the user for validation.

• A list of the ELVIS instruments that are used by the user in this experiment. For each instrument, the document should not only contain a list of terminals and their connections, but also information on how the instrument has been configured by the user.

Experiment Result

This document is generated by the lab server upon the successful completion of an experiment. The document contains entries of data that correspond to each of the parameters that the user wants to measure for the given experiment. For a simple experiment that contains an input waveform, an output waveform and time values, the document would contain an array of double values for each of the three parameters. The clients graphing API would then use this information to recreate these waveforms for the user.

LAB CONFIGURATION

```
<?xml version="1.0" encoding="utf-8" standalone="no" ?>
<!DOCTYPE labConfiguration SYSTEM
"http://localhost/LabServer/xml/labConfiguration.dtd">
<labConfiguration lab="MIT ELVIS Weblab" specversion="0.1">
        <setup id="5">
                <name>OpAmp Differentiator Circuit</name>
                <description>Aren't opAmpss just swell?</description>

        <imageURL>http://localhost/labServer/setupImages/opAmpDifferentiator.gif</im
ageURL>
                <terminal instrumentType="FGEN" instrumentNumber="1">
                        <label>Input Waveform</label>
                        <pixelLocation>
                                <x>121</x>
                                <y>94</y>
                        </pixelLocation>
                </terminal>
                <terminal instrumentType="SCOPE" instrumentNumber="2">
                        <label>Oscilloscope</label>
                        <pixelLocation>
                                <x>195</x>
                                <y>156</y>
                        </pixelLocation>
                </terminal>
        </setup>
</labConfiguration>
```

EXPERIMENT SPECIFICATION

```
<?xml version="1.0" encoding="utf-8" standalone="no" ?>
<!DOCTYPE experimentSpecification SYSTEM
"http://localhost/labServer/xml/experimentSpecification.dtd">
<experimentSpecification lab="MIT NI-ELVIS Weblab" specversion="0.1">
        <setupID>1</setupID>
                <vname download="true">VIN</vname>
                <iname download="true">IIN</iname>
                <mode>V</mode>
                <function type="WAVEFORM">
                        <waveformType>SINE</waveformType>
                        <frequency>100</frequency>
                        <amplitude>0.5</amplitude>
                        <offset>0.1</offset>
                </function>
        </terminal>
        <terminal instrumentType="SCOPE" instrumentNumber="2">
                <vname download="true">VOUT</vname>
                <iname download="true">IOUT</iname>
                <mode>V</mode>
                <function type="SAMPLING">
                        <samplingRate>100</samplingRate>
                        <samplingTime>0.01</samplingTime>
                </function>
        </terminal>
        <userDefinedFunction>
                <name download="true">SQRTID</name>
                <units>A</units>
                <body>SQRT(VIN)</body>
        </userDefinedFunction>
</experimentSpecification>
```

EXPERIMENT RESULTS

```
<?xml version="1.0" encoding="utf-8" standalone="no" ?>
<!DOCTYPE experimentResult SYSTEM
"http://localhost/labServer/xml/experimentResult.dtd">
<experimentResult lab="MIT NI-ELVIS Weblab" specversion="0.1">
        <datavector name="VIN" units="V">1,2,3,4,5</datavector>
        <datavector name="VOUT" units="I">6,7,8,9,0</datavector>
</experimentResult>
```

5. REFERENCES AND FURTHER READING

A . Service Broker to Lab Server API
http://icampus.mit.edu/iLabs/Architecture/downloads/protectedfiles/Service%20Broker%20to%20Lab%20Server%20API.doc

B. Client to Service Broker API (Release 6.0)
http://icampus.mit.edu/iLabs/Architecture/downloads/protectedfiles/Client%20to%20Service%20Broker%20API%206.0.doc

C. MicroElectronics WebLab Lab Server
http://icampus.mit.edu/iLabs/Architecture/downloads/protectedfiles/Microelectronics%20WebLab%20Lab%20Server%20Description.doc

D. Service Broker Experiment Storage API (Release 6.0)
http://icampus.mit.edu/iLabs/Architecture/downloads/protectedfiles/Service%20Broker%20Experiment%20Storage%20API%206.0.doc

E. Gikandi's Thesis: ELVIS Version 1 (Available upon request)

F. Bryant's Thesis: ELVIS Version 2 (Available upon request)

G. Adnaan's Thesis: ELVIS Version 3 (Available upon request: adnaan@mit.edu)