

Figure 5.5: ezAMI software architecture diagram

### 5.3.1 Architecture of the Software

The software design is following the object oriented design (OOD) rule. Each function module is an object. The communication between objects is either through class inheritance or the signal-slot mechanism that is built in the Qt interface design environment (IDE). Figure 5.5 shows the flow diagram of the ezAMI architecture. Each box represents an function object. The arrow lines between those objects denote the interactions. The center blue circle represents the top graphic user interface (GUI). All other rectangle boxes are the sub systems interacting with the main GUI. The whole software has the following function blocks.

- Main interface
- Simulator
- Compiler
- Excitation generation
- Plotting
- Schematic drawing
- Project tree view
- Code formatting
- AMI model Generation

Detailed description of each function object is presented in the following.

#### Main GUI interface

The main GUI interface is implemented in `MainWindow.cpp` . It is the top level object which interacts with each sub-object as illustrated in figure 5.5. During initialization, an instance of each sub-object will be created. Table 5.1 summarizes the functionality of the subroutines.

Most of those functions are the handling functions when a menu or button is clicked in the software main interface. Communication between the objects

is also handled in some of the functions, e.g. `updateProjectTreeFromCCompiler()` which handles any signal sent by the compiler object. More detailed discussion on this topic will be presented in the following object introduction sections.

## Simulator

Simulator object is implemented in the `simulator.cpp`. According to figure 5.5, the simulator receives the excitation waveform from plotting object and then prepare for simulation input. This is done in the function `receiveInputWave()` and `prepareInputWave()`. Once the input waveform is ready, then the `run()` function is called, in which a DLL is loaded and three AMI functions: `AMI_Init()`, `AMI_GetWave()`, and `AMI_Close()` are called consecutively. At the end of compilation, the `prepareOutput()` is called and the `outputReady()` signal is emitted. The `setDllPath()` function is called whenever a DLL is selected and is linked to the model in the AMI model dialog.

## Compiler and AMI model generation

The compiler object is implemented in the `compiler.cpp`. The AMI model generation object is implemented in the `generatedlldialog.cpp`. Both objects are interacting closely with each other. Once all the development is completed, the next step is compiling. A dialog window is popped up when the build action is clicked from the software main interface.

There are three import functions: `compile()`, `generateDll()`, and `generateAmiFile()` in compiler object. The first function builds the DLL for simulation. The second function builds the IBIS-AMI DLL for simulation in circuit simulator. The last function generates the `.ami` file that is associated with the DLL when simulated in circuit simulator. There are two signals: `updateProjectArch()` and `sendBuildInfo()`. The first one is emitted when the project architecture is changed by the compilation. The main interface updates project management window accordingly. The second sends the rules

Table 5.1: Main interface functions

Functions	Comments
MainWindow()	Object constructor
~MainWindow()	Object destructor
on_actionAMI_Generation_triggered()	Generate AMI dialog
on_actionRun_2_triggered()	Start simulation
on_actionAMI_Generation_triggered()	Start AMI model generation
on_actionBuild_2_triggered()	Start compiler build
on_actionSave_All_triggered()	Save project and code
on_actionSave_triggered()	Save just the code
on_actionExcitation_triggered()	Draw excitation schematic
on_actionAMI_triggered()	Draw all schematic
on_actionPlot_triggered()	Draw plot schematic
on_doubleClicked()	Action when schematic is double clicked
isInRegion()	Check which schematic is double clicked
on_actionOpen_triggered()	Open new project
on_projectTreeView_doubleClicked()	Open file from project management window
onCustomContextMenu()	Handle right click in project management window
setupContextMenu()	Set up right click menu
setProjectInfo()	Initiate project architecture when new project created
on_CustomContextMenu_triggered()	Handle a click in right-click menu
saveProjectFile()	Save project info into file
saveCodeFile()	Save code into file
on_actionClean_2_triggered()	Clear project management window when clear project menu is clicked
on_actionLVFFN_triggered()	Setup LVFFN example in ezAMI
updateProjectTreeFromCompiler()	Update any change when compilation is done
addModelFilesInDirectory()	Parse files in the directory selected and add them to project model
updateModelByChild()	Update project model when branch change
parseAmiFunctions()	Fill three AMI template functions into code area
copyPath()	Copy whole LVFFN project file into the new directory user selected

to the AMI model generation object for AMI model and `.ami` file generation.

The AMI model generation object is essentially a dialog window which

allows users to select the platform on which the AMI model is going to be used and to specify parameters in the .ami file. Besides, any building errors and messages generated by the compiler will be displayed to the users. This object calls function in the compiler object to conduct the actual building and compilation.

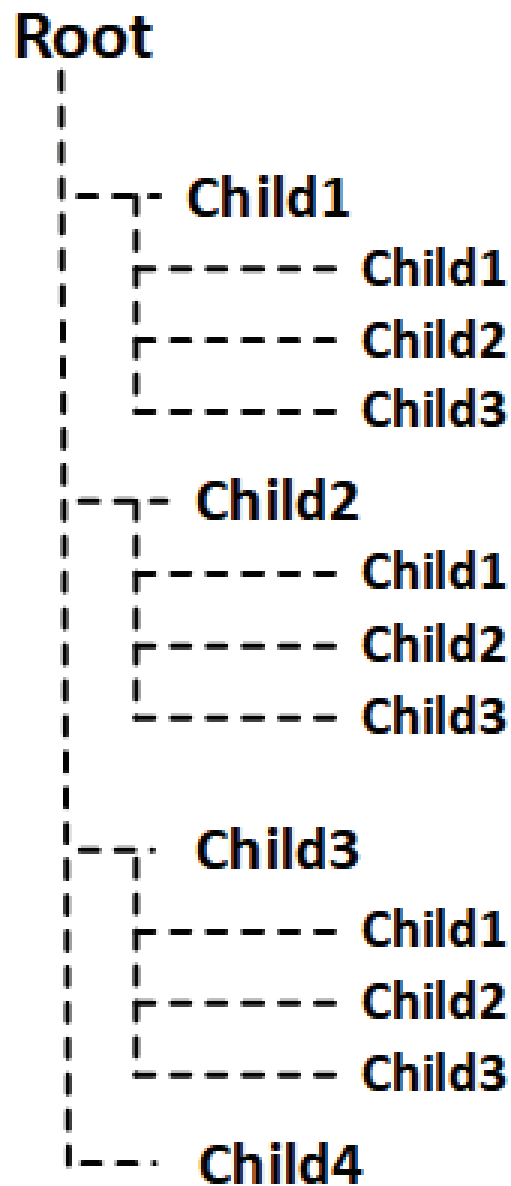


Figure 5.6: Tree model diagram

## Plotting

The plotting object is another big object which is implemented in the `plotting.cpp`. This object draws the waveform generated from the excitation object and simulator object. The `setupCor()` function creates an empty coordinate system from scratch. `XaxisSetup()` updates the X axis with respect to a variety of time scales.

The `coordinateSetup()` is a function handling communications between the plotting object and the excitation generation object. It receives information such as type, sample per bit, total number of bits, and amplitude. Then it calls the `updatePlotPoints()` and `updateCoor()`.

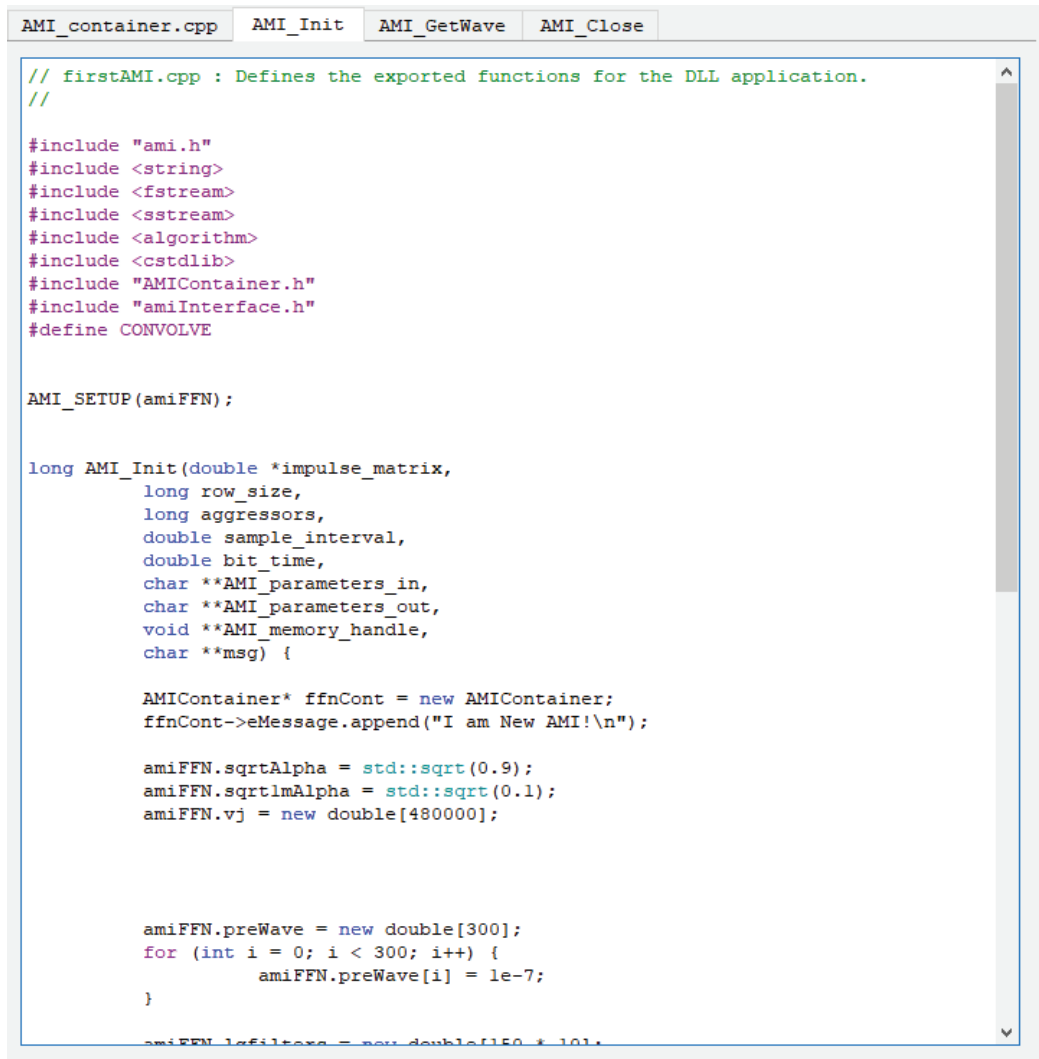
The `updateCoor()` function clears old plotting information and adds new plot into the coordinate system in terms of the updated plotting information from either the excitation generation or simulator object. The `updatePlotPoints()` actually does plotting by drawing all the individual points. The last function, `addSimulatedWave()`, receives simulated waveform from the simulator object and conducts necessary processing, and subsequently calls the `updateCoor()` function.

## Excitation generation

The excitation generation object and schematic object are implemented in the `excitationdialog.cpp`. The excitation generation object generates two level modulated and four level modulated PRBS bit sequences. The dialog window is activated by a double-click on the excitation icon. Users can specify the data rate, samples-per-bit, total number of bits, amplitude, and offset in the dialog. With the input from user, the sample waveform is plotted in real time at the bottom half of the dialog window. User specifies the excitation type which is either PAM-2 or PAM-4. The specifications in the dialog window is routed to the simulator and plotting object when the "Ok" button is clicked.

There are two important functions implemented in the `excitationdialog.cpp`. They are `updateHash()` and `getSamples()`. the user specified information

in the dialog window is stored in a hash table. The `updateHash()` function updates the hash table whenever there is a change in the dialog window. The `getSamples()` function updates the excitation waveform information for plotting.



The image shows a code editor window with four tabs: `AMI_container.cpp`, `AMI_Init`, `AMI_GetWave`, and `AMI_Close`. The `AMI_Init` tab is active, displaying the following C++ code:

```
// firstAMI.cpp : Defines the exported functions for the DLL application.
//

#include "ami.h"
#include <string>
#include <fstream>
#include <sstream>
#include <algorithm>
#include <cstdlib>
#include "AMIContainer.h"
#include "amiInterface.h"
#define CONVOLVE

AMI_SETUP(amiFFN);

long AMI_Init(double *impulse_matrix,
             long row_size,
             long aggressors,
             double sample_interval,
             double bit_time,
             char **AMI_parameters_in,
             char **AMI_parameters_out,
             void **AMI_memory_handle,
             char **msg) {

    AMIContainer* ffncont = new AMIContainer;
    ffncont->eMessage.append("I am New AMI!\n");

    amiFFN.sqrtAlpha = std::sqrt(0.9);
    amiFFN.sqrtlmAlpha = std::sqrt(0.1);
    amiFFN.vj = new double[480000];

    amiFFN.preWave = new double[300];
    for (int i = 0; i < 300; i++) {
        amiFFN.preWave[i] = 1e-7;
    }

    amiFFN.lofilters = new double[150 * 101];
```

Figure 5.7: Code region

## Schematic

There is a schematic window in the main interface. It locates at the lower left corner of the main window. The schematic symbols are handled by the `svgload.cpp`, and `sceneclick.cpp`. Both classes are derived from the

`QGraphicView` and `QGraphicsScene` class. Most of the functions are overloaded functions from the parent classes. Both objects handle loading SVG icons and double-clicking events.

## Project Tree View

Project tree view object is a project hierarchy maintainer. The object is implemented in the `projecttreemodel.cpp` and `projecttreeitem.cpp`. The structure of the project is a tree-like structure as shown in figure 5.6. There is a root node under which multiple children inherit. The child could also have children. Each node preserves partial project information which is retrieved and used by other objects.

There are three level descendants for the ezAMI software. A project root node serves as the ancestor. At the second level, there are five siblings. They are the project name node, source code node, executable node, AMI model node, and resource node. The project name node contains the project name and the local directory storing the project. The source code node contains all the `.cpp` and `.h` files. The executable node saves the DLL for the simulator project. The AMI model node contains the AMI DLL and the `.ami` files after the AMI model generation is completed. The resource node saves all the `.txt` files that are used during simulation.

The `projecttreeitem.cpp` implements the `projectTreeItem` object which maintains a single tree node. It stores pointers to its parent and children. The data associated with this object is stored in a vector. User can add/remove child to the node using the function `appendChild()`, `removeChild()`, and `removeAllChild()`, respectively. Data can be retrieved using the `data()` function. Parent/child node can be accessed through the `child()` and `parentItem()` function. The node information such as the number of child, data elements, and the position of itself in its parent node can be read out with the `childCount()`, `columnCount()`, and `row()` function.

The `projecttreemodel.cpp` is an object maintains the project architecture. This is an important object to other objects in the software. For



instance, the compiler will have to retrieve the information stored in the source code node before compilation. The main interface reads the information in this object to update the graphics display in the project management window. The object is inherited from the `QAbstractItemModel`. All the functions implemented in this object are overloaded functions except for the `openModelData`, `getProjectRoot()`, `removeRow()`, and `setupModelData()` function. The documentation of the overloaded functions can be found in the documentation of the `QAbstractItemModel` object on the Qt maintenance website.

The `openModel()` function basically reads in the `.ezproj` project file, populates the contents, and creates a project tree model. The `getProjectRoot()` function retrieves the ancestor of the tree model. The `removeRow()` removes the child given the parent node and the position.

## Code formatting

ezAMI software is essentially an integrated development environment (IDE). The code formatting object is to make the coding in the code space more user friendly. This object is implemented in the `codeformathighlight.cpp`. The object specifies font color for key words in the coding region, e.g. variable type, macros, control key words etc. Figure 5.7 shows the result of this object. When the project tree model object loads source code into coding space, the source code is first parsed by the code formatting object and is rendered with respect to the keywords as shown in figure 5.7.

The `textToProcess()` function reads all the text contents as a string from the code space. Then it parse the text into a variety of categories such as macros, comments, and regular code. The `setCommentFlag()` function sets a flag specifically for comments in between `"/**"` and `"*/"`. There is a special case `"/**/"` which is rare but there is possibility of existence. The `setCommentFlag()` function is designed to address such scenario. The `setTextColor()` conducts the evaluation per the input word and returns the font color for the text editor to render.

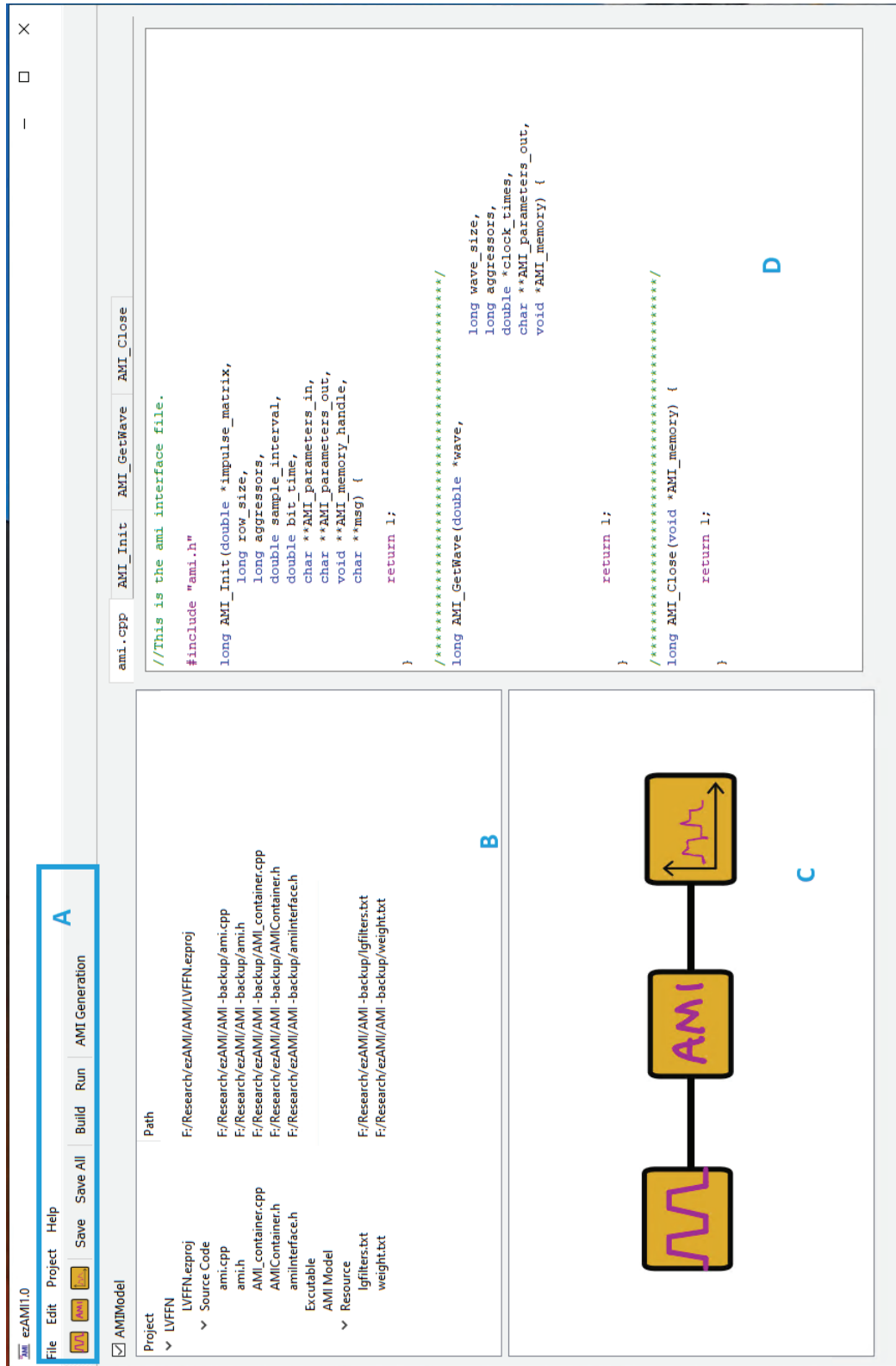


Figure 5.8: ezAMI main interface

### 5.3.2 Software interface

In this section, the software main window is introduced. The main interface has four main regions: The menu action region, the project management region, the schematic region, and the coding region. Figure 5.8 shows the main window in which label A, B, C, and D denote the four regions mentioned above respectively.

#### Menu and actions

The menu actions are placed at the top of the software main interface. Figure 5.9 shows all the expanded menu. Figure 5.9(a) shows the expanded **File** menu. **New** is a sub-menu which has two actions: **Project** and **File**. When **Project** or **File** is clicked, a new project or a new file creation dialog is promoted. The **Open** action promotes the open project dialog. It allows user to open a `.ezproj` project file from a directory. **Example** is a sub-menu which includes only one action: **LVFFN**. The **LVFFN** is a built-in example project in which my PhD work on LVFFN is implemented. In Appendix B, a detailed tutorial is included for interested user to learn how to use this software.

The **Save** and **Save All** are actions saving codes in coding region and the project hierarchy in files. The **Save** action only saves code in source code files, while the **Save All** saves both code and project hierarchy. The **close** action closes the whole software. It is recommended that user should save all the information before closing the project.

In the **Edit** menu (figure 5.9(b)), there are three actions: **Copy**, **Cut**, and **Paste**. The implementation of these three actions has not been completed in the current 1.0 release, but will be completed in the future release.

The **Project** menu (figure 5.9(c)) has six actions implemented. The **Copy Project** action copies the whole project directory into the new location. The **Close Project** will only clear up the project management region and leave others as is.

The **Build** action calls compiler to compile all the source code and gener-

ates a DLL which will be used in the simulation later on. The **Clean** action deletes all the compiler generated files and restores the directory to the initial state. The **Run** action calls the simulator to conduct the simulation using the generated DLL with the excitations generated by the excitation object. The **AMI Generation** action will generate IBIA-AMI DLL and .ami file, which can be simulated in circuit simulation software like ADS.

The **Help** menu contains all the actions related to software logistics such as software tutorial, documentation, and license information. The ezAMI 1.0 has only implemented the software introduction and license action which is the **About** action (figure 5.9(d)).

Figure 5.9(e) shows the convenient menu bar in which the actions used frequently is added. There are three icons in the red frame. They are new actions for schematic drawing. The first icon with square wave in the icon is an action for adding the excitation generation symbol and plotting symbol into the schematic region (labeled C in figure 5.8). The second diagram with "AMI" in the icon draws excitation, AMI model and plotting symbol. The third icon only places the plotting symbol into the schematic region.

### Project management region

The project management region is labeled B in figure 5.8. Once a new project is created or an existing project is opened, the project hierarchy is displayed in this region. The diagram displayed is a tree-like structure in which one row represents one node. Each row has two columns. The left column contains the name and the right one contains the location on local computer. The project structure has a root node which is the name and the location of the project. There are four children nodes that are the source code, executable, AMI model, and resource node. These four nodes have their own child/children which is/are the specific file/files under the category. User can add/remove files through right-clicking on the node. The file contents are displayed in the code region when the node is double-clicked.

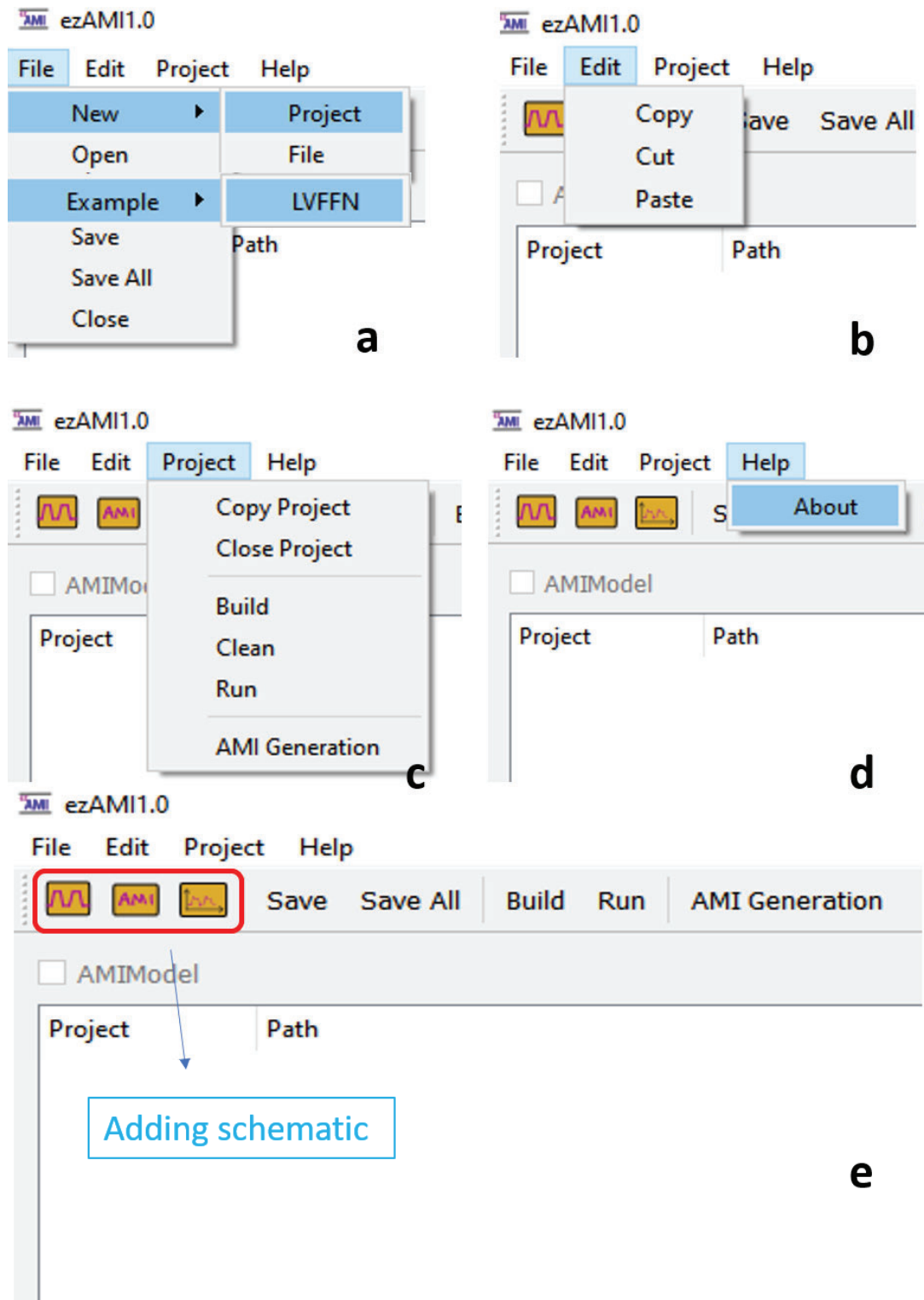


Figure 5.9: Menu actions

### Schematic region

Schematic region is labeled C in figure 5.8. As shown, there are three symbols which represents the excitation, AMI model, and plotting. The connection in between represents the data flow from the excitation to the plotting through the AMI model. The excitation is processed in the AMI model. The result is plotted in the plotting object. Figure 5.10 shows the excitation generation dialog and AMI model dialog when the specific symbol is double-clicked. The excitation generation dialog generates PRBS excitation for PAM-2 and PAM-4. Users can select which of them to be generated using the checkbox. Users can specify data rate, samples per unit interval, magnitude, offset etc. All those information will be used for excitation waveform generation. The model association dialog allows user to specify the DLL for simulation. The DLL model is selected by choosing the DLL file in a directory through browsing. Once the DLL model is selected, the model is associated with the AMI symbol in the schematic window. Once the simulation is launched, the simulator will load the DLL and call the functions in this DLL for excitation waveform processing.

### Coding region

The coding region is labeled D in figure 5.8. It is for users to write their code which will be compiled into the DLL. The coding region is a text editor with keyword rendering. The code loaded will be pre-processed to highlight the keywords such as variable declaration, control syntax, macros, and so on so forth. The coding region has four windows. They are Your\_Code, AMI\_Init, AMI\_GetWave, and AMI\_Close. Your\_Code interface displays the code loaded by double-clicking the file in project management region. AMI\_Init, AMI\_GetWave, and AMI\_Close are interfaces having the `AMI_Init()`, `AMI_GetWave()`, `AMI_Close()` template functions displayed. Users can modify them as desired .

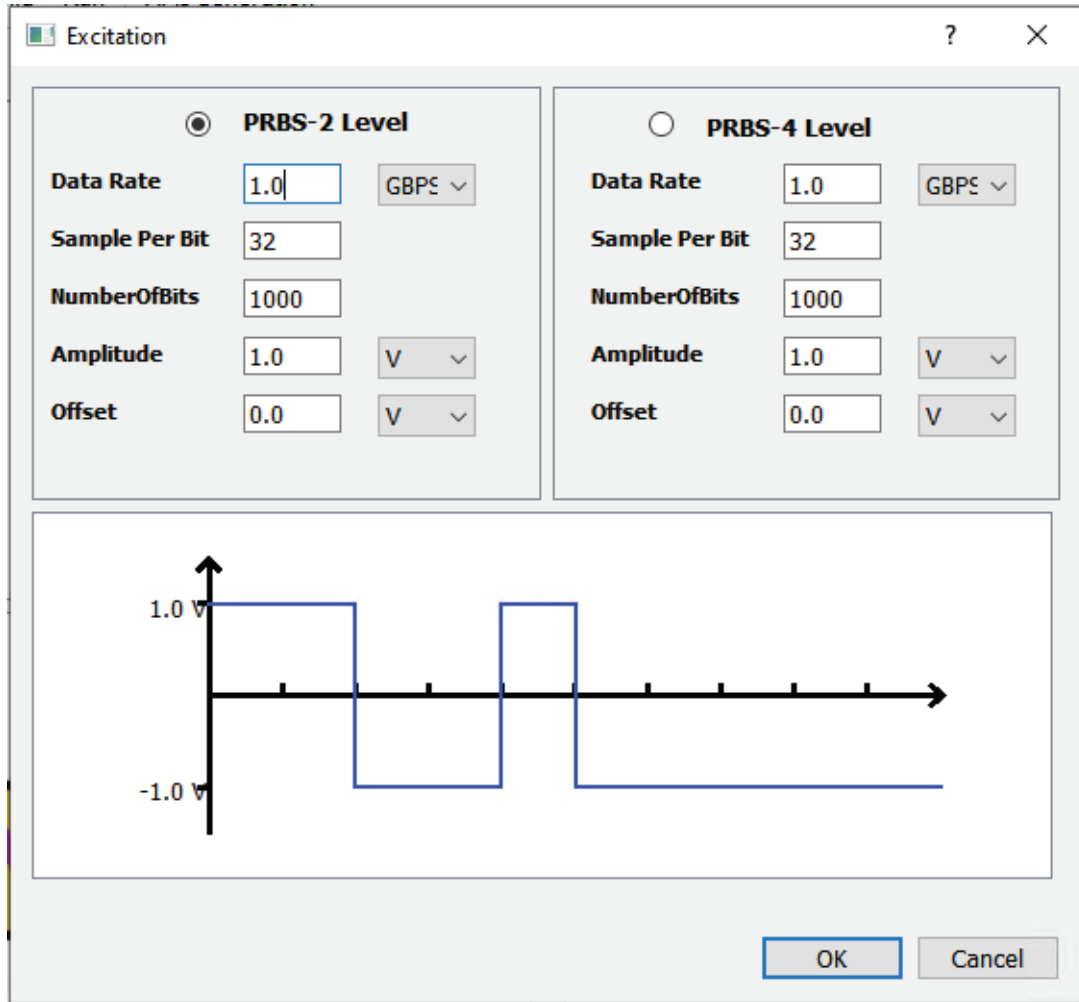


Figure 5.10: Excitation generation dialog

## 5.4 Conclusion

The behavioral model generated from machine learning cannot be directly simulated in circuit simulator software, which significantly limits its applications in practice. We successfully implemented the LVFFN model in IBIS-AMI, an industrial standard. The model was verified using ADS from Keysight. Eye diagram analysis was conducted in ADS as well. The eye diagrams from LVFFN obtained with ADS appear very similar to the one generated with the reference model.

To facilitate IBIS-AMI model generation for machine learning models, an