The FORCE

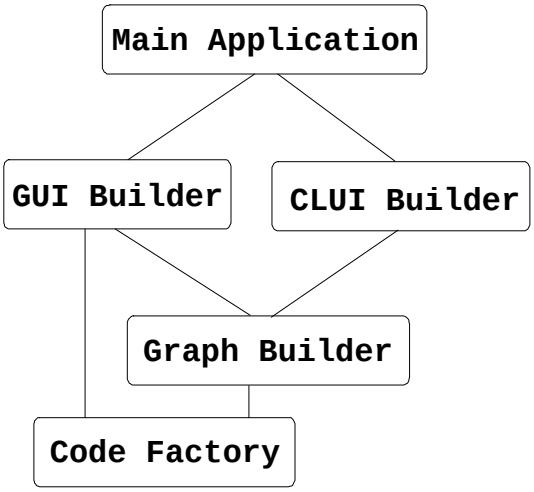
The Flow Of Regulated C Expressions

Top-Level Design

Author: Michael Pellauer

Package Overview

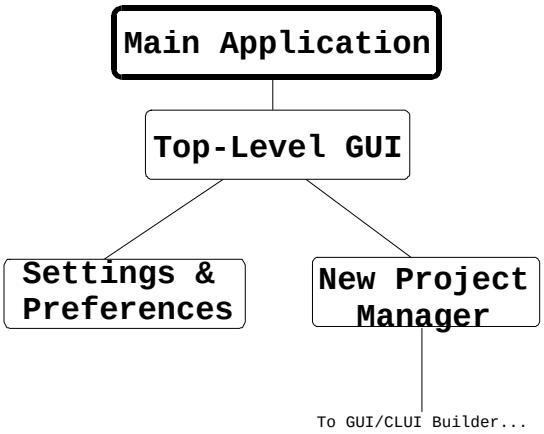
The FORCE is composed of five major packages. First is the **Main Application**, which consists of top-level GUI such as menus, and is responsible for project management. Next is the **GUI Builder**, which encapsulates all creation and layout of GUI elements. Third is the **CLUI Builder**, a wrapper around Command-Line projects. Most important is the **Graph Builder**, which handles actual graph construction and management. To avoid redundant code information in memory, all actual code information is managed by a **Code Factory** package, which uses the "nifty counter" technique to keep redundancies to a minimum.



Top-Level Package View

Main Application Components

The Main Application package is devoted to tying together the other components of the program, rather than containing functional components itself, hence the number of components is relatively low.



Main Application Components

The **Top-Level GUI** component encapsulates the menus and any other appropriate elements, such as a toolbar. NOTE: This design assumes that all GUI elements will be implemented with the GTK toolkit.

```
class FORCETopGUI {
public:
	//Callbacks for Menu Functions
	static void FileNewCB(GtkWidget* widget, gpointer data);
	static void FileOpenCB(GtkWidget* widget, gpointer data);
	static void FileCloseCB(GtkWidget* widget, gpointer data);
	…
protected:
	//Links to other components
	FORCESettings *settings_;
	FORCEProjectManager *projman_;
	//The Menus Themselves
	GtkWidget *menu1;
	…
};
```

It must also include a component which is used to change overall **Settings and Preferences**. These include color highlighting of different node types, and advanced options such as using real C names of functions.

```
class FORCESettings{
public:
	//Callbacks for GUI Elements
	static void highlightCB(GtkWidget* widget, gpointer data);
	static void nameToggleCB(GtkWidget* widget, gpointer data);
	…
protected:
	//The actual options;
	boolean highlighting;
	…
	//The Dialog Box Components Themselves
	GtkWidget *label;
	…
};
```

Finally, it must contain a **New Project Manager** which helps the user create new projects. It encapsulates a dialog box where the user is presented with the option to name the project, select the directory its stored in, and select special libraries for inclusion. Depending on whether the users selects a GUI-based project or not, the CLUI-builder or GUI-builder packages are launched.

```
class FORCEProjectManager{
public:
	//Callbacks for GUI Elements
	static void FileSelectCB(GtkWidget* widget, gpointer data);
	static void LibraryCB(GtkWidget* widget, gpointer data);
	static void NameCB(GtkWidget* widget, gpointer data);
	…
protected:
	//Links to other components
	FORCECLUIBuilder *clui_;
	FORCEGUIBuilder *gui_;
	//The Dialog Box Components Themselves
	GtkWidget *label;
	…
};
```

CLUI Builder Components

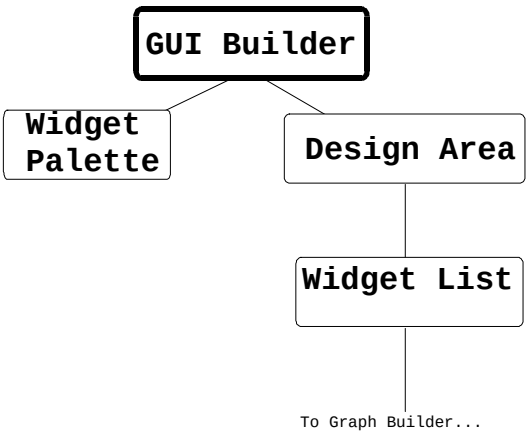
The CLUI Builder package represents an abstraction around command-line programs. As such, it doesn't need to do much work. It simply handles code generation appropriate to set up a main() function, then tells the Graph-Builder to begin generation.

```
class FORCECLUIBuilder {
public:
	FORCECLUIBuilder(); //Set up stuff for programming

	void GenerateCode(FileHandle *f); //First generate stuff for main()
	                                  // then tell Code Builder to generate
	                                 // its stuff. Then put in ending stuff.
	…
protected:
	//Link to the Graph Builder
	FORCEGraphBuilder *graph;
};
```

GUI Builder Components

The GUI Builder package includes every component related to the creation and layout of graphical widgets, and must handle generation of appropriate C code to set up the UI.



GUI Builder Components

The **Widget Palette** component encapsulates a tool bar of toggle buttons that allows the users to select different widgets. When the user clicks in the Design Area, the appropriate widget is created.

```
class FORCEWidgetPalette{
public:
	//Callbacks for the toggle buttons
	static void SelectToolCB(GtkWidget* widget, gpointer data);
	static void ButtonCB(GtkWidget* widget, gpointer data);
	static void ToggleButtonCB(GtkWidget* widget, gpointer data);
	static void CheckBoxCB(GtkWidget* widget, gpointer data);
	static void LabelCB(GtkWidget* widget, gpointer data);
	…
	//Accessor for enumerated type
	FORCETool getCurrent { return current_; }
protected:
	FORCETool current_; //enumerated type of tools.
	//Links to other components
	FORCEGUIBuilder *gui_;
	//The Toggle Buttons Themselves
	GtkWidget *selectButton, *buttonButton;
	…
};
```

The GUI Builder also includes a **Design Area** component, which represents an area for the user to lay widgets out in, and is responsible for handling mouse interactions. When the user clicks in the design area, it responds appropriately to the current tool. For instance, if the current tool is the “button”, then it creates a new button where the user clicks.

```
class FORCEDesignArea{
public:
	//See if the user clicked on a widget
	FORCEWidget *determineClicked(int x, int y);

	//Create widget based on current tools.
	void createWidget(int x, int y);
```

```
        …
protected:
        //Links to other components
        FORCEGUIBuilder *gui_;
        FORCEToolPalette *tools_;
        FORCEWidgetList *widgets_;
        //The Drawing Area
        GtkWidget *drawing_area_;
        …
};
```

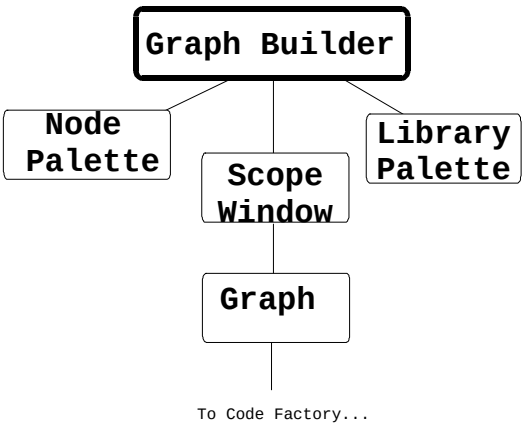
The Design Area utilizes another component, the **Widget List**. This is responsible for keeping track of every widget that the user creates, including their location and options such as labels. It must also keep track of the corresponding C code associated with creating a widget, for code generation time. Each node of the list will have a link to a Graph component (described below) which represents flow graphs the user has associated with that widget. Because it would be inefficient to retain multiple copies of the data in memory, this should be managed by pointers which are managed by a special Code Factory package (described below).

```
class FORCEWidgetList{
public:
        //Generate Code for widget
        void generateCode(FileHandle *h);

        //accessors
        int getX() {return x;}
        int getY() {return y;}
        …
protected:
        //Linked list link
        FORCEWidgetList *next_;
        //Links to actual widget data
        FORCEWidgetImplementation *widgets_;
        //layout stuff
        int x, int y;
        //Link to code from factory
        FORCECode *code_;
        …
};
```

Graph Builder Components

The Graph Builder is the most complex package. First it must handle all aspects of graph creation, layout, and connections. To accomplish this it has a **Node Palette** component, a **Library Palette** component, and a **Scope Window**.



Graph Builder Components

The **Node Palette** is used to select which type of node (see The FORCE User Specifications document) they want to create. It is a floating toolbar that the user can move at will. It consists of toggle buttons corresponding to each node type, when the user selects a node, it becomes the current node type.

```
class FORCENodePalette{
public:
	//Callbacks for the toggle buttons
	static void SelectToolCB(GtkWidget* widget, gpointer data);
	static void ValueNodeCB(GtkWidget* widget, gpointer data);
	static void VariableNodeCB(GtkWidget* widget, gpointer data);
	static void FunctionNodeCB(GtkWidget* widget, gpointer data);
	static void CommentCB(GtkWidget* widget, gpointer data);
	…
	//Accessor for enumerated type
	FORCENode getCurrent { return current_; }
protected:
	FORCENode current_; //enumerated type of nodes.
	//Links to other components
	FORCEGraphBuilder *builder_;
	//The Palette Buttons Themselves
	GtkWidget *valueButton, *selectButton;
	…
};
```

The **Library Palette** is a floating palette that allows the user to select from lists of predefined functions to include in their graphs. It is sorted by different categories, and includes user-defined functions that the user has created with the subgraph tool (See The FORCE Specifications). The user selects one function. When a new function node is created it defaults to the selected function. The palette displays information about the function, its parameters, etc.

```
class FORCELibraryPalette{
public:
	//Callbacks for the toggle buttons
	static void SelectToolCB(GtkWidget* widget, gpointer data);
	static void ValueNodeCB(GtkWidget* widget, gpointer data);
	static void VariableNodeCB(GtkWidget* widget, gpointer data);
	static void FunctionNodeCB(GtkWidget* widget, gpointer data);
```

```
        static void CommentCB(GtkWidget* widget, gpointer data);
        …
        //Accessor for enumerated type
        FORCETool getCurrent { return current_; }
protected:
        FORCETool current_; //enumerated type of tools.
        //Links to other components
        FORCEGUIBuilder *gui_;
        //The List widgets that do the displaying.
        GtkWidget *category_list_, *function_list_;
        //The labels to display help information.
        GtkWidget *help_area;
        …
};
```

The **Scope Window** represents the current function scope the user is working on. It primarily consists of a graph design area where the user actually constructs the graph. The Scope Window is responsible for handling all user interaction, as well as actual creation and layout of nodes, as represented by the **Graph** component (see below). Secondarily, it utilizes a **Variable Manager** helper class, which is displays all variables available in the current scope, and a **Parameter Manager**, which allows the user to add input parameters and return values. If the user is inserting C Code, then it must be able to switch to a text editing mode, with the help of a Text Area component.

```
class FORCEScopeWindow{
public:
        //Determines which node user clicked on
        determineClicked(int x, int y);

        //Passes on to graph.
        generateCode(FileHandle *f);
        …
protected:
        //external links
        FORCEGraphBuilder *graph_;

        //”helper” components.
        FORCEDesignArea *design_;
        FORCEParameterManager *parm_man_;
        FORCEVariableManager *variable_man_;
        FORCETextArea *text_editor_;
        //The actual graph
        FORCEGraph *graph_;
        //The design area
        GtkWidget *design_;
        …
};
```

Finally, the Graph Builder package includes a component that represents the **Graph** itself. This component is the most complex of any. It must include different classes to represent value nodes, variable nodes, function nodes, conditional nodes, and while loops. Each node must include links to other nodes for each input and output plug, as well as for nodes that are attached by flow-of-control connections (see The FORCE Specification).

```
class FORCEGraph { //Mostly Abstract Super Class
public:
        //Actually generates the code appropriate to the node
        generateCode(FileHandle *f);
        …
```

```
        //accessors
        int getX() {return x;}
        int getY() {return y;}
        …
protected:
        //Link to next object in flow of control
        FORCEGraph *next_;
        //Links to input
        FORCEGraph *input[MAX_INPUT];
        //Links to output
        FORCEGraph *output[MAX_OUTPUT];
        //Position info
        int x, int y;
        //Link to actual code from factory
        FORCECode *code_;
        …
};
```

Code Factory Components

The Code Factory Package is designed to get around the problem of redundant code. If the user creates a program with fifty calls to strcmp, we do not want to keep 50 identical copies of the data on strcmp code generation in memory. To prevent this, widgets and nodes obtain their code from the Code Factory, which loads the information from a database on physical storage employing the “nifty counter” technique. Thus the first time a particular function is requested it loads it from disk, increments a counter, and returns a reference to a FORCECode object. Every subsequent request is ensured to return a reference to the same object, and the object is deleted if there are no more nodes using it.

```
class FORCECodeFactory { //Mostly Abstract Super Class
public:
        //Returns code info for the specified widget type for GUI.
        FORCECode *getButtonCode();
        FORCECode *getToggleButtonCode();
        FORCECode *getLabelCode();
        FORCECode *getTextFieldCode();
        …
        //Returns code info for the specified function.
        FORCECode *getFunctionCode(FORCEFunctionID f);
        …
protected:
        //Nifty Counters for each widget type
        int buttonCount, labelCount,TextFieldCount;
        …
        //Nifty Counter Array by function ID.
        int funcs[MAX_FUNCTION_ID];
        //On-disk-database
        FileHandle *database;
};
```

A Word on Execution

As the specification stands, The FORCE is not designed to execute code visually by following routes. Such functionality could be added by altering the Code Generation Factory to return scripted instructions rather than actual C code. An additional Executable package could run the instructions, keeping track of the state of each variable and branching appropriately.

However, this model cannot deal with “Code Nodes” (see The FORCE Specification) where a more advanced user has entered actual C code. This would require code parsing which is beyond the scope of this project. Additionally, running would require the display and control of multiple windows, say where a command-line program is running, or a GUI-based program is in its main loop. Retrieving the interaction with the user through these windows and back to The FORCE, and responding in an appropriate way is difficult to say the least. The FORCE would have to generate and compile code that was much more complex then what it is currently designed for, and this code would bear little resemblance to what the user intended, defeating the goal of The FORCE as a C teaching tool. Finally, this interaction code would irreversibly wed the program to The FORCE, making it completely unportable.

By programming with The FORCE the user is already demonstrating a minimum knowledge of flow-of-control. While dynamic running of code with visual response in the graph could sometimes be a valuable teaching tool, the cost of adding this feature is simply too high.

Schedule

3/5 Language Specification Finalized
3/12 Top-Level Design Finalized
3/17 Graph Builder GUI Prototype
3/24 GUI Builder GUI Prototype
4/5 Detailed Interface Design Finalized
4/12 Basic Node Creation, Connection
4/19 Basic Code Generation
4/21 Basic GUI Generation
4/23 Function Nodes, Subgraphs
4/26 Branches, Loops
4/29 Full Code Generation
5/1 Tentative Integration, Full GUI Generation
5/8 Full Integration
5/15 Done!