Google Summer of Code 2018

gr-modtool overhaul

Swapnil Negi

March 20, 2018

1 Introduction

Presently gr-modtool is not Py3k compatible. But now most software developers are moving towards python 3 because of its high intuitiveness, chained error handling and various other intriguing features. So, it is a necessity to make the tool Py3k compatible.
In the current scenario, applications are expected to be modular, customizable, and easily extensible. All these can be accomplished by building a strong plug-in architecture. The code base of gr-modtool as of now is static, i.e., some portion of the tool is repetitive. So, to improve the usability and understanding of the tool, it is highly desirable to make it more functional.
In the proposal, the focus is mainly on the pockets of Py3k idiosyncrasies, the patches where code needs to be functional for better usability and the methodology for building modtool as a pluggable command line tool.

1.1 Primary features of project

1. Version independent compatibility with python 2 and python 3.
2. Rewrite the tool as a plug-in architecture
3. Refine the codebase to make the code more functional and restructuring the present codeblocks
4. Write an actual UI for making the tool much more interactive (if possible)

2 Proposed Workflow

Initially, I'll work upon making the entire modtool python 3 compatible. Then, I will work upon initialising plugin architecture, then rewriting the entire tool upgrading the current utilities and user experience (if needed). After that, I will start with the main task of making the code more functional and rewriting the same as plug-in architecture with present functionalities like add, remove, disable, etc. as plugins.

2.1 Python version independent compatibility

Although gr-modtool automates the boring, monotonous work involved in writing the boilerplate code, makefile editing, etc. for cpp or python 2 developers, it is the need of the hour to make it python 3 compatible.
So, the major patches of python 3 incompatibility that I observed are:

2.1.1 Handling Exceptions

In python 3, there is a change in methodology for handling exceptions as it gets quite confusing when raising multiple kinds of exceptions in python 2.
So the present code throws a SyntaxError for several try-except statements.
For example, in build_utils.py, the present code:

```
try:
    if os.environ['do_makefile'] == '0':
        do_makefile = False
    else:
        do_makefile = True
except KeyError, e:
    do_makefile = False
```

changes to

```
try:
    if os.environ['do_makefile'] == '0':
        do_makefile = False
    else:
        do_makefile = True
except KeyError as e:
    do_makefile = False
```

2.1.2 Raising Exceptions

In python 3, there is a change in methodology for raising exceptions since exceptions are classes and they need to be instantiated before raising. So, the present code raises SyntaxError while raising exceptions.
For example in build_utils.py, the present code:

```
mo = re.search (r'\.([a-z]+)\.t$', template_name)
if not mo:
    raise ValueError, "Incorrectly formed template_name '%s'" %
        (template_name,)
return mo.group (1)
```

changes to

```
mo = re.search (r'\.([a-z]+)\.t$', template_name)
if not mo:
    raise ValueError("Incorrectly formed template_name '%s'" %
        (template_name,))
return mo.group (1)
```

2.1.3 Import Statement

For modules that have been renamed we can use try-except or can import the from __future__ python module.
There are several other variations like difference in print statement, metaclasses, integer incompatibilities, etc. which will be incorporated if required.

2.2 Functional Code

Although gr-modtool works like magic and is extremely smooth and easy to use, the codebase is fairly static chunk of code with series of if-then-else rules which makes the code look slightly redundant and not very clear. These slight shortcomings can be easily tackled by making the code more functional.
For example, in modtool_add.py, several parts of the code are repetitive like

```
self._info['blocktype'] = options.block_type
if self._info['blocktype'] is None:
    print str(self._block_types)
    with SequenceCompleter(sorted(self._block_types)):
        while self._info['blocktype'] not in self._block_types:
            self._info['blocktype'] = raw_input("Enter block type: ")
            if self._info['blocktype'] not in self._block_types:
                print 'Must be one of ' + str(self._block_types)


self._info['lang'] = options.lang
if self._info['lang'] is None:
    language_candidates = ('cpp', 'python')
    with SequenceCompleter(language_candidates):
        while self._info['lang'] not in language_candidates:
            self._info['lang'] = raw_input("Language (python/cpp): ")
```

which can be made less redundant by functional approach like

```
def getValue(parameter, candidates):
   self._info[parameter] = options.parameter
   if self._info[parameter] is None:
       print str(candidates)
       with SequenceCompleter(sorted(candidates)):
           while self._info[parameter] not in candidates:
             self._info[parameter] = raw_input("Enter "+parameter+" type:
                 ")
               if self._info[parameter] not in candidates:
                   print 'Must be one of ' + str(candidates)
```

and then calling the function with the required parameters to get the value.
Moreover using a functional approach even for non redundant code eases the process of program development and program testing. It serves as procedural abstraction wherein a programmer uses it as a black box and just needs the name and parameters to invoke it.
So, I will make the entire code functional to make it more readable and make future development on tool a bit easy.

2.3 Plugin Architecture

Currently modtool is not available as a plug-in. The basic advantages of re-writing it as a plugin architecture are:-

- Implementing and incorporating application features become easier.
- Isolating a module becomes easier
- Custom versions of applications can be created without source code modifications.
- Disabling unwanted features becomes easier at user end

After the plug-in architecture is implemented the modtool can be extended to include VOLK and RFNoC.

There will be three main classes of the architecture:-

- **CLI**: This is the main command line interface. It handles user input and delegates execution to the plugin manager
- **PluginManager**: Loads plugins and calls the appropriate plugin method when the user invokes the command line.
- **AbstractPlugin**: Defines common behavior for all plugins. Each plugin class must extend this one to be considered a valid plugin.

The command line tool will have the following syntax: **cli** <**plugin**><**command**> <**arguments**> wherein arguments aren't mandatory.
For example:*cli gr_modtool add -t general* or *cli rfnocmodtool help* are some examples of valid commands.

The logic implementation of these classes (basic understanding) is:

- **CLI**: If the number of arguments are less than two, it will call pluginmanager to show the list of available plugins with their functionalities and commands. Else it will pass the plugin name and arguments to the pluginmanager without the command to print the help of the plugin.
- **PluginManager**: Firstly, it will dynamically initialize the list of all available plugins. After that, it will import the plug-in from the plugins list and load the same.
  Code for loading a plugin will look like:

```
def load_plugin(self, plugin_name):
    """ Loads a single plugin given its name """
    if not plugin_name in __all__:
        raise KeyError("Plugin " + plugin_name + " not found")
    try:
        plugin = self.__plugins[plugin_name]
    except KeyError:
        # Load the plugin only if not loaded yet
        module = __import__("plugins." + plugin_name,
            fromlist=["plugins"])
```

```
            plugin = module.load()
            self.__plugins[plugin_name] = plugin
        return plugin
```

After that it will call the given command of the given plugin with the user specified arguments (if any).
Proper exception handling is also a part of the project.

- **AbstractPlugin**: This is the base class for all plugins. It simply reads all public methods from the plugin class and exposes them to the plugin manager as commands that can be invoked.
  Its basic structure looks like:

```
def _commands(self):
    """ Get the list of commands for the current plugin.
    By default all public methods in the plugin implementation
    will be used as plugin commands. This method can be overriden
    in subclasses to customize the available command list """
    attrs = filter(lambda attr: not attr.startswith('_'), dir(self))
    commands = {}
    for attr in attrs:
        method = getattr(self, attr)
        commands[attr] = method
    return commands
```

  After that for building the plugins (presently just the gr_modtool plugin), the plugin needs to be put in the plugins folder and its class should extend the AbstractPlugin.

There are several other tasks like creating metadata files, design the functions in the gr_modtool plugin, etc. but that have been left intentionally.

3 Timeline

I will utilize the period of community bonding to familiarize myself with the GNU Radio community. I will also make sure to gain a deeper insight of the source code. This will then enable me to contribute more efficiently to the community. I will also investigate various ways to implement the plugin architecture and work on building a sample plugin architecture to get the hang of bugs and various issues that come with it. I will also define minute details of the project so that I face minimal difficulty in the coding period.

The necessary documentation will be done in parallel to to the development. There is 13 week coding period. I have made my deliverables on weekly basis. I have my holidays in the months of May, June and July. So, I'll work full time in these months, i.e., around 40-45 hours a week. In August, I'll work for around 30-35 hours a week.

The expected timeline of my project is given below:

Timeline of the project

Apr 23 - May 14	Define minute details of the project and build a sample plugin architecture.
May 14 - May 21	Make the entire modtool Python 3 compatible.
May 21 - May 28	Initialise the plugin architecture with the complete basic structure of CLI, PluginManager and the AbstractPlugin.
May 28 - June 4	Complete the basic structure of plugin architecture and the main plugin class which extends the AbstractPlugin.
June 4 - June 11	Restructure modtool_newmod.py
June 11 - June 18	Restructure modtool_base.py
June 18 - June 25	Restructure modtool_add.py
June 25 - July 2	Restructure modtool_rm.py + bug fixes of all previous modules
July 2 - July 9	Restructure modtool_disable.py, modtool_rename.py
July 9 - July 16	Restructure modtool_help.py and modtool_info.py
July 16- July 23	Restructure the remaining modtool files
July 23 - July 30	Thoroughly test the entire modtool, buffer time for completing the remaining tasks
July 30 - Aug 6	Start working on UI of the tool
Aug 6 - Aug 14	Complete the project and submit the final report

4 Deliverables of GSoC 2018

The deliverables of the GSoC project are as follows:

- Properly implemented version independent compatibility of python 2 and python 3 with thorough testing.
- Properly restructured code in favor of functional behavior.
- Properly implemented plugin architecture which can be easily extended to include rfnocmodtool and volk modtool.
- Slight work for the improvement of user interface.

4.1 Milestones

- Phase-1: Py3k compatibility, complete basic structure of Plugin architecture.
- Phase-2: Restructure modtool_newmod.py, modtool_base.py, modtool_add.py, modtool_rm.py.
- Final Evaluation: Complete rewriting modtool as plugin architecture, restructuring the modtool to make it more functional and make it entirely python version independent.

5 Acknowledgement

I have thoroughly gone through the GSoC StudentInfo page and GSoC Manifest page. I hereby assure that I will abide by the rules and regulations. I also accept the three strikes rule and the details mentioned.
I also assure that I will communicate with the assigned mentor regularly, maintain thorough transparency and keep my work up to date.

6 License

The entire code during the coding period will be transparent, i.e., available on Github. The code submitted will be GPLv3 licensed.

7 Personal Details and Experience

I am a second year undergraduate at Indian Institute of Technology Roorkee. My areas of interest are software development, competitive programming and applied probability. I am proficient in Python, C++, JAVA, Javascript and PHP. I am familiar with git environment as I work regularly on Gitlab. I haven't contributed much to open source but as we all know "**Cyberspectrum is the best spectrum**", so I'll really like to contribute to GNU Radio and make it as my first remarkable experience. I am proficient in two human languages including English.

I have the experience of working closely with a team as I am an active member of **Information Management Group** at IIT Roorkee, a bunch of passionate enthusiasts who manage the **institue main website**, internet and intranet activities of the university and the placement portal. My major project as a part of group is 'Forminator', an intranet based forms application in which the user can create forms, select the audience groups or individuals, create groups for future use and use the information database of the institute. The project has some remarkable features like conditional fields implemented using tree algorithm.

I am also a member of **Programming and Algorithms Group** which is aimed at spreading a culture for Algorithms and related stuff among people both in and outside IIT Roorkee by organising contests, delivering lectures, etc.

I started off with GNU Radio in February 2018. To get familiarized with the code, I made the following contributions to the codebase:

1. Pull request #1672: Edit Copyright gr-modtool generated files, add feature for adding copyright holder
2. Pull request #1676: Improve check for block(s) removal in modtool_rm.py
3. Pull request #1679: Add script for blocking the creation of same blockname

I will always be available on email or Google Hangouts for any kind of discussion or query.
I am highly interested to contribute to GNU Radio after the GSoC period. After the period, I'll mainly focus on the UI of the modtool.

Here is the link to my CV.

7.1 Other Details

Address	:	Roorkee, Uttarakhand, India
Email	:	swapnil.negi09@gmail.com
Github	:	https://github.com/swap-nil7/
LinkedIn	:	https://www.linkedin.com/in/swapnil07/
Codechef	:	https://www.codechef.com/users/swapnil07

8 Conclusion

gr-modtool is currently very powerful tool as it highly facilitates the user's experience by eliminating the necessity to type the boilerplate code, editing makefiles, etc. But the inclusion of the above mentioned features will make the code more customizable, extensible and will also ease the process of further program development.