Google Summer of Code 2018

gr-modtool overhaul

Swapnil Negi

March 27, 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 this can be accomplished by building a strong plugin architecture.
Currently, modtool is only usable as a command line program and isn't scriptable from other Python scripts which limits its scope and efficiency. So, building a Python API for the modtool will be highly advantageous.
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 the code needs to be functional for better usability and the methodology for building modtool as a pluggable API.

1.1 Primary features of the project

1. Version-independent compatibility with python 2 and python 3
2. Rewrite the tool as a plugin API/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 will work on making the entire modtool python 3 compatible. Then, I will work to initialize the plugin API. After that, I will start with the main task of making the code more functional and rewriting the modtool as a plugin API with present functionalities like add, remove, disable, etc.

2.1 Python version-independent compatibility

Although gr-modtool automates the boring, monotonous work involved in writing the boilerplate code, makefile editing, etc. for C++ or Python 2 developers, it is the need of the hour to make it Python 3 compatible.
So, the major patches of Python 3 incompatibilities 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 '{}'"
                     .format(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 them from __future__ python module.

There are several other variations like the difference in the print statement, meta-classes, integer incompatibilities, etc. which will be incorporated if required.
The version-independent compatibility can also be ensured using the Python 2 and Python 3 compatibility library "Six" which provides simple utilities for wrapping over differences between two versions.

2.2 Functional Code

Although gr-modtool works like magic and is extremely smooth and easy to use, the codebase is a 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 using the 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 {} type: '
                                                  .format(parameter))
                if self._info[parameter] not in candidates:
                    print 'Must be one of {}'.format(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 only requires the name and

parameters to invoke it.
So, I will make the entire code functional to make it more readable and make future development of the tool a bit easier.

2.3 Plugin API/Architecture

Currently, modtool is not available as a plugin. 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 become easier at the user end

Moreover, modtool is only usable as a command line program and is not available as an API. The basic advantages of building a Python API for the modtool are:-

- Sharing and distribution of the content becomes easier
- Redundancy reduces as the code written by the user is reusable
- Usability of the tool increases as it is also scriptable from other Python scripts

After the Plugin API 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 are not mandatory.
For example:*cli gr_modtool add -t general* or *cli rfnocmodtool help* are some examples of valid commands.

The basic structure of the architecture is as follows:

- CLI: If the number of arguments is 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.
- plugins/__init__.py: This is the required file to make Python treat the directory as a containing package. Here, it will also be used to set the __all__ variable, i.e., the list of the modules that the package exports as an API. The basic structure of the code will look like

```
import os
plugin_dir = "plugins"

__all__ = []
for filename in os.listdir(plugin_dir):
    filename = plugin_dir + "/" + filename
    if os.path.isfile(filename):
        basename = os.path.basename(filename)
        base, extension = os.path.splitext(basename)
        if extension == ".py" and not basename.startswith("_"):
            __all__.append(base)
```

- PluginManager: The module will load the required plugin from the plugins list and call it with the specified commands(if any).
  Code for loading and calling a plugin will look like:

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


def call(self, plugin_name, command_name, args):
    """ Calls the given command on the given plugin """
    try:
        plugin = self.load_plugin(plugin_name)
        if not command_name:
            self.help(plugin)
        else:
            try:
                command = plugin._commands()[command_name]
                return command(args)
            except KeyError:
                # Command not found in plugin. Print only plugin help
                self.help(plugin)
    except KeyError:
        # Plugin not found, pring generic help
        self.help_all()
```

  After this, it will call the specified command of the given plugin with the user-specified arguments (if any).

- 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 overridden
    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 this, 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, designing the functions in the gr_modtool plugin, etc. Their details have been left intentionally but they will be implemented along with the above architecture model.

2.4 UI Enhancements (if time permits)

Although modtool is quite simple to use with features like SequenceCompleter, it can be made more interactive by using Python libraries like Pygments (syntax highlighting library), Fuzzy Finder (library to narrow down suggestions with minimal typing), etc.
The work on Graphical User Interface will be done after the GSoC coding period, and hence is beyond the scope of the proposal.

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 enable me to contribute more efficiently to the community. Moreover, I will 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. Additionally, I will define minute details of the project so that I face minimal difficulty in the coding period.

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

The expected timeline for 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, __init__.py (for API support), and the PluginManager.
May 28 - June 4	Complete the basic structure of plugin architecture. Complete Abstract Plugin and the main plugin class for gr-modtool.
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 with python API which can be easily extended to include rfnocmodtool and volk modtool.
- Slight work for the improvement of Command Line Interface.

4.1 Milestones

- Phase-1: Py3k compatibility, the 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 along with python API, restructuring the modtool to make it more functional and making the modtool entirely python version-independent.

4.2 Review/Merge Cycle

The code can be reviewed as per the proposed timeline whereas it can be merged according to timeline stated below:

- May 28: Merge code for version-independent Python compatibility.
- The code of the restructured modtool scripts can be merged one week after the proposed end date of the restructuring of the particular script.
- August 6: Merge the entire restructured tool with Plugin API/Architecture.

4.3 Automated Testing

Although the code will be thoroughly tested locally by comparing the generated scripts with the gr-newmod files or the present modtool generated scripts, I will run Pylint against the generated scripts and work on building a minimal Python QA that creates a module, adds a Python source and Python QA file to it, runs the QA test with randomly generated numbers in the Python source and then deletes the entire directory.

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 (hosted on our own server; due to the group's policy and copyright issues, the code cannot be published). I haven't contributed much to open source but since we all know "Cyberspectrum is the best spectrum", I'll really like to contribute to GNU Radio and make it as my first remarkable experience. I will not get any extra credits for the GSoC project. 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 institute main website, internet and intranet activities of the university and the placement portal. My major project as a part of the 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 the culture of algorithms and competitive programming among people both in and outside IIT Roorkee by organizing 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 the 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 the 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. I'll always be available for fixing the bugs that come up in 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.