Google Summer of Code 2019

Block header parsing tool

Arpit Gupta

March 28, 2019

1 Introduction

GNU Radio consists of a vast library of blocks, both in-tree and out-of-tree. Presently, it is hard to use these blocks in a context other than GNU Radio itself or as an interface with blocks automatically or programmatically. So, there is a need for a python based tool that can interact with block headers written in C++, to automatically parse them and extract information about them such as which getters/setters they have, their I/O signatures, factory signatures etc.

It would be great to come up with a tool that could be useful for GNU Radio itself since we could parse block headers and convert the information into other formats. For example, a tool that automatically creates GRC bindings from block headers. An abstract representation of the block can be created by analyzing the header code, which would be in the form of a tree, thus making the code readable in a hierarchical fashion. Then, the abstract representation could be easily parsed for any further use.
GRModtool currently provides us with a powerful CLI to work with and so could be easily used for our purpose because of parsing tool itself being a CLI. GRModtool can be extended with the parsing tool as one of its utilities.

The best python based parsers currently available that are used to parse C++ code are libclang (with python bindings) and pygccxml, so they will be used to parse GNU Radio block header files.

1.1 Primary features of the project

1. Implementation of both the parsers (libclang with python and pygccxml).
2. Abstract the implementation details of parsers and create an abstract representation of the header files.
3. Extend GRModtool CLI to provide support for the block header parser. Make GRModtool use this code itself instead of the built-in code.
4. Parse the abstract representation, create YAML files for GRC, write a script to add or modify new optional YAML key parameters.

5. Extend the utility of block header parsing tool to take a list of files or a directory and parse them and also extend it to parse different file types for example .cc files etc (if time permits).

2 Proposed Workflow

Initially, I will be doing the groundwork by setting up the abstract implementation of the parser (pygccxml). Then comes parsing the abstract representation and storing information in a python dictionary. Next step will comprise of creating YAML files for the GRC and integrating the tool with GRModtool. The same procedure will be followed for libclang.

The working prototype of the entire project (using pygccxml parser only) is available here. The file ast_generator.py implements the pygccxml parser to generate the abstract representation and the file text_parser.py parses the entire AST to get all the useful information. All the data that we get using the function in the latter file is then returned in a python dictionary and a YAML file is created in yaml_generator.py file. pygccxml_ast.txt is the AST generated for sample.h header file.

2.1 Implementation of parsing tools

GNU Radio API (headers) requires C++ for compiling, not straight C, which requires us to use a parser to parse C++. Libclang (python-based) and pygccxml both do an excellent job for this purpose and are especially good for GNU Radio header files because of their brilliant and elegant yet simple and most importantly their generic structure.
But there is a trade-off in their utility. Here are a few points of comparison in both the parsers:-

- Pygccxml takes a lot of computation time whereas libclang algorithms are better in this case.
- Pygccxml is a quite mature python package and also has proper documentation which gives it an advantage over libclang which has a very small documentation.
- Pygccxml generates a nice indented AST which is really understandable and easy to work with while this is not the case in libclang. Complex python functions are required to be implemented to get all the required information while parsing C++ code. (Section 2.4.2)
- But still libclang is a very popular C++ parser due to its well-written source code and is under continuous development which provides an excellent opportunity to explore it.

So, it is worth it to use both of the python packages to parse the header files (with a choice given to the user to use any one of them). Both the parsers can be abstractly implemented and either one of them could be used.

2.1.1 Implementation of pygccxml

Pygccxml provides a simple framework to navigate C++ declarations, using Python classes. It just needs to get access to the namespace in the file that we require to parse. It automatically works its way down the hierarchy of the code structure parsing all the variables, function calls and so on. It is compatible with Python 2.7, 3.4, 3.5, pypy and pypy3.

Implementation of pygccxml is as shown below used for parsing a header file from a gr-directory:

```
from pygccxml import utils
from pygccxml import declarations
from pygccxml import parser


def pygccxml_parser(filename, gr_directory):
    """
    Implementation of pygccxml parser
    """
    generator_path, generator_name = utils.find_xml_generator()

    # Configure the xml generator
    xml_generator_config = parser.xml_generator_configuration_t(
        xml_generator_path=generator_path,
        xml_generator=generator_name)
    filename = filename

    # Parse the c++ file
    decls = parser.parse([filename], xml_generator_config)

    # Get access to the 'analog' namespace in 'gr'
    global_namespace = declarations.get_global_namespace(decls)
    ns = global_namespace.namespace("gr")
    gr_block_type = gr_directory.split("-")[1]
    gr_namespace = global_namespace.namespace(gr_block_type)
```

2.1.2 Implementation of Libclang with python

Libclang is a python package used for analyzing C/C++/ObjC code at the compiler level which means it can also be used to analyze all the standard C++ libraries too. Libclang with python bindings provide us a pythonic way of implementing classes and functions to parse C++ files. It is also compatible with Python2.7, 3.4, 3.5, pypy and pypy3.

Implementation of libclang in python is as shown below for used for parsing a header file from a gr-directory:

```
import sys
from clang.cindex import Index
from pprint import pprint


```

```
def get_cursor_id(cursor, cursor_list = []):
    """
    Cursor point to the current node
    """
    if cursor is None:
        return None

    for i,c in enumerate(cursor_list):
        if cursor == c:
            return i
    cursor_list.append(cursor)
    return len(cursor_list) - 1

def get_info(node, depth=0):
    """
    Recursively get info of all the nodes
    """
    children = [get_info(c, depth+1)
                for c in node.get_children()]
    return { 'id' : get_cursor_id(node),
             'kind' : node.kind,
             'usr' : node.get_usr(),
             'spelling' : node.spelling,
             'location' : node.location,
             'extent.start' : node.extent.start,
             'extent.end' : node.extent.end,
             'is_definition' : node.is_definition(),
             'definition id' : get_cursor_id(node.get_definition()),
             'children' : children }


index = Index.create()
tu = index.parse(sys.argv[1], args=['-x', 'c++'])
pprint(('nodes', get_info(tu.cursor)))
```

But libclang has a drawback that definitely needs to be worked on.
Libclang parses C++ header files along with all the standard headers that are included in the code which generates a very long AST. This makes the AST very difficult to parse, besides it provides us with redundant information of standard libraries. So, the way I figured out is to create a list of standard C++ header files, write a python function to exclude them from being parsed.

2.2 Abstract implementation of parsers

It is due to the weird syntax of C++, all the current C++ parsers are still under constant development, thus need for their abstract implementation. Also, the trade-off in the utility of both the parsers mentioned in section 2.1 makes it necessary to do so. Abstracting the details of implementation of both the parsers will allow us to work on both of them easily without complicating the code, and we can even replace them with some other parser if need be.

Abstract implementation of both the parsers from CLI is as shown below:

```python
def AbstractASTGenerator(parser):
    """
    Abstract parser implementation
    """
    try:
        if parser.lower() == 'clang':
            ClangASTGenerator()
        elif parser.lower() == 'pygccxml':
            PygccxmlASTGenerator()
        # Can be easily extended further if required
    except Exception as e:
        pass

def InitailParserInput():
    """
    Parser Type Input
    """
    parser_list = [
        'clang',
        'pygccxml',
    ]
    parser = input('Parser type(' + '/'.join(parser_list) + '): ')
    if parser.lower() in parser_list:
        ast = input('Do you require an AST file? (y/n): ')
        if ast.lower() == 'y':
            AbstractASTGenerator(parser.lower())
        # Similarly other functions can be implemented
    else:
        raise Exception('Please specify a valid parser name!')
```

As shown above, other functionalities can be abstracted in such a way such that any other parser after being implemented can be easily called to use its functionality such as to generate AST, get make function, I/O signatures.
After the abstract implementation, we get Abstract Syntax Tree (AST) for the header code to work with. AST provides us with the whole complete tree structure of the header file to be parsed.

Example AST generated by both the parsers for a GNU Radio header file (agc2_cc_impl.h) are:

- Clang-generated-AST
- Pygccxml-generated-AST

2.3 Extend GRModtool CLI to provide support for this tool

GRModtool is a script in GNU Radio that provides a CLI for creating OOT modules. It provides various features, for example creating new OOT modules, editing makefiles, using templates, generating YAML files and much more.
Block header parsing tool also being a command line utility can easily be integrated with GRModtool.
Another GRModtool command can be called from CLI, for example from `gr_modtool parse` for this purpose.

I have already gone through the code of GRModtool, which is brilliantly written and it will be really easy to work with.

- CLI will be implemented in the `/cli` module of GRModtool, extending the use of click and click plugins.
- Implementation of the backend code of both the parsers will be implemented in the `/core` modules of GRModtool.

GRModtool already consists a parser written in `parser_cc_block.py` file to parse C++ files. It is although written elegantly but the code is limited only to parse GNU Radio cc files and if thought to be extended further as a generic API, it cannot be done because of the code structure which is written by analyzing the code pattern limited to GNU Radio headers, whereas for generic parsing (parsing every C++ file in a similar way), so that tool can be used as an API external to GNU Radio, both the python packages (pygccxml and libclang) do a great job.

Manual syntax check as shown below (code snippet from `parser_cc_block.py`) is no longer required as earlier done while parsing C++ files because both parsers throw compiler exceptions in case of syntax error (if any).

```
while not end_of_list:
    # Keep track of (), stop when reaching final closing parens
    if not in_string:
        if c[i] == ')':
            if parens_count == 0:
                if read_state == 'type' and len(this_type):
                    raise ValueError(
                        'Found closing parentheses before finishing '
                        'last argument (this is how far I got: {})'.format
                            \
                        (str(param_list))
                    )
```

This code is no longer required to be implemented for parenthesis check of every function defined.

2.4 Parse the abstract representation, Create YAML files for GRC

After generating the AST (abstract representation of the block header), it needs to be parsed to get all the useful information from it. Because of the use of two different parsers in this project, they generate AST in a different format, thus different parsing strategies need to be implemented for both of them.

2.4.1 Parsing pygccxml generated AST

Pygccxml generates AST (.txt file mentioned in section 2.2) in the form of tree data structure because of the implementation of the helper-print function in it to print all the declarations. The best way to parse the AST generated by pygccxml is to parse through the top-level hierarchy of the header file and then reach the bottom level that is parsing different defined functions and function declarations.

The architecture of pygccxml generated AST parser is as shown below:

- Parse through all the namespaces that are defined in the header file.
- Parse through the classes defined in the namespace used.
- Parse through the getters/setters defined in a class (public/protected/private).
- Parse through the make function defined in the public constructor of the class.
- Finally parse through different variables and functions defined in constructors (defined as parameters in YAML files).

Starting through the top-level, example parser implementation for the pygccxml generated AST is as shown below, not complete parser though, but a part of it, the complete parser will be implemented as part of my project:

```
def namespace_parsing(filename, gr_directory):
    """
    Function to parse the namespace defined in the file
    """
    with open(filename, 'r') as file_object:
        lines = file_object.readlines()
    namespace_name = []
    namespace_index = []
    gr_block_type = gr_directory.split("-")[1]
    for line in lines:
        if 'namespace_t' in line:
            namespace_name.append(line.split(":
                ")[1].strip().replace("'", ""))
            namespace_index.append(lines.index(line))
    if namespace_name[0] == gr_block_type:
        lines = lines[0: namespace_index[1]]
    elif namespace_name[1] == gr_block_type:
        lines = lines[namespace_index[1]: len(lines)]
    else:
        raise Exception('Wrongly specified namespace, must be
            {}'.format(gr_block_type))
```

Similarly, the rest of the parsing API will be defined according to the given architecture during my GSoC coding period.

2.4.2 Parsing libclang generated AST

Libclang (with python bindings) generates AST (.txt file mentioned in section 2.2) as a JSON which is really easy file format to work with. The architecture of libclang generated AST parser is as shown below:

- To parse the methods defined, we need to get all the child nodes with key-value pairs defined as ''kind': CursorKind.CXX_METHOD' from the JSON.
- To parse the parameters defined, we need to get all the child nodes with key-value pairs defined as ''kind': CursorKind.PARM_DECL', from the JSON.

This allows us to get all the possible variables, function declarations, and parameters defined in the header file, but we need their data types, return types too.

```
def traverse(node):
    """
    function that helps cursor traverse through all the nodes
    """
    for child in node.get_children():
        traverse(child)
    print('Found {} [line={}, col={}]'.format(node.displayname,
        node.location.line, node.location.column))
```

The output of this function will provide us with data types of all the variables and return types of different function declarations defined in the block header, which could be then used along with the parser architecture to get information of all the possible nodes of the AST.

2.4.3 Raising proper exceptions while parsing

Raising proper exceptions is also necessary because the header file should be defined according to GNU Radio header API standard and then only it should be allowed to generate YAML files. Few of these are:

- A public destructor must be defined along with a public constructor for main block header files so that it is called automatically when the object goes out of scope.
- A 'work()' function must be declared with return type 'int' with arguments 'gr_vector_const_void_star &input_items', 'gr_vector_void_star &output_items' and 'noutput_items'.

2.4.4 Creating YAML files for GRC

Once the data is parsed, store it in a python dict, create YAML files as per the layout of standard YAML files in GNU Radio.
GRModtool already consists of a YAML_generator used for the purpose of generating YAML files for the GRC. It has been implemented using Ordered Dictionary in python which is brilliant because key order definitely matters for YAML files. The code from YAML_generator in GRModtool can be extended reused and extended for this purpose. Other advantage is that this would allow us to create generic YAML structure for the GRC.

YAML file architecture in GNU Radio consists of these following keys:

- id, label, category, flags, parameters, inputs, outputs, templates, file_format.
- cpp_templates (only if flag includes cpp), documentation (optional).

Values of the keys namely id, label, flags (will be a CLI input) and file_format are implemented as shown below:

```
def file_name_parsing(file_name):
    """
    Function for generating YAML architecture
    """
    label_dict = {
        'c': '(Complex)',
```

```
    }
    label_part = (file_name.split("_"))
    if len(label_part[-1]) < 2:
        if label_part[-1] not in label_dict:
            label_part.pop()
        else:
            label_part[-1] = label_dict[label_part[-1]]
    data = {}
    data['id'] = file_name.split(".")[0].lower()
    data['label'] = (' '.join(label_part)).title()
    data['flags'] = ['python'] # optional cpp templates
    data['file_format'] = 1
    return data
```

All the other key-values for YAML are the output of parsed data.

- templates key-value pair is determined by the make function of the block header.
- I/O signature of the header file provides us the input, output key-value pair of YAML file.

A script for adding optional key-value parameters must also be implemented, for example, if the flag key has cpp parameter, cpp_templates key-value pair should be generated.

2.5 Manage different data-file type (if time permits)

One of the features that would be really good to implement is to take a list of files as an input, put them in a queue and parse them, reducing the effort to parse headers one at a time.
Also, the utility of this tool could be extended to implement a parser for .cc files in GNU Radio, just different text-parsing functions need to be implemented with almost the same code, not a difficult job at all.

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 into the source code. This will enable me to contribute more efficiently to the community. Moreover, I will figure out ways the header parsing tool in the most possible efficient way by using efficient python libraries for my purpose, squashing all (if not most) the bugs before completion. Additionally, I will define the 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 a 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 35-40 hours a week while in August I'll work for around 30-35 hours a week. I have made my deliverables accordingly on a weekly basis.

The expected timeline for my project is given below:

Timeline of the project

May 6 - May 27	Define the minute details of the project and build a sample parser architecture.
May 27 - June 3	Initialize the implementation of pygccxml to generate AST, abstract its implementation, define all the helper functions required.
June 3 - June 10	Start working on parsing pyggcxml generated AST.
June 10 - June 17	Complete the implementation on parsing AST generated by pygccxml with thorough testing.
June 17 - June 24	Extend the script in GRModtool to generate YAML files for GRC or if required write a new script for this purpose raising proper exceptions.
June 24 - July 1	Submit the code for Phase 1 evaluation and write a script to edit the optional YAML keys.
July 1 - July 8	Integrate the new tool with GRModtool so that it could be used as one of the utility of GRModtool.
July 8 - July 15	Complete the parsing tool that runs along with GRModtool as its utility, initialize the implementation of libclang to generate AST.
July 15 - July 22	Start working on parsing liblclang generated AST.
July 22 - July 29	Submit the code for Phase 2 evaluation and complete the implementation on parsing AST generated by libclang with thorough testing.
July 29 - Aug 5	Complete the remaining tasks for this parser (would be easy because most of the work is already completed earlier while fully implementing pygccxml).
Aug 5 - Aug 12	Thoroughly test the block header parsing tool, buffer time for completing the remaining tasks and start working on the extended part that is to take a file list as an input and parse them.
Aug 12 - Aug 19	Start working on extending this tool for .cc files.
Aug 19 - Aug 26	Complete the project and submit the final report.

4 Deliverables of GSoC 2019

The deliverables of the GSoC project are as follows:

- Implement abstract parsers compatible with python 3.
- Properly parsing the generated AST and creating YAML files from command line itself for GRC.
- Extend Command Line Interface of GRModtool to support the new tool that would abstract the use of parsers.
- Implement the complete tool with thorough testing.

4.1 Milestones

- Phase-1: Successfully parsing AST generated by pygccxml.
- Phase-2: Extend the support of GRModtool with the parsing tool and complete the first parser with thorough testing, complete parsing AST with libclang.
- Final Evaluation: Completion of the block header parsing tool using both parsers (pygccxml and libclang), successfully creating YAML files for the GRC, making the tool functional as a generic API so that it would be easy to work on this tool further, and could be used for other purposes.

4.2 Review/Merge Cycle

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

- June 24: Merge code for parsing AST generated by pygccxml.
- July 22: Merge code for complete parsing tool (Using pygccxml) integrated with GRModtool, and parsing AST with libclang.
- August 19: Merge the entire block header parsing tool along with it's extended utilities (if possible).

4.3 Testing

The code will be thoroughly tested locally at almost every step of the whole architecture, test scripts will also be written to ensure all the third-party packages are correctly installed. The code will also be compatible with standard pylint rules. Additional tests will be written to ensure that the block header files are parsed correctly.

5 Acknowledgment

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, signal processing, and deep learning. I am proficient in Python, C++, Java, JavaScript, and PHP. I am familiar with git environment as I work regularly on GitHub. I haven't contributed much to open source till now, but 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 Lectures and Tutorials Portal (Lectut), an intranet-based study portal, used by unique 4k campus students and faculty every month with an aim of assisting them to achieve their academic goals. The project has been implemented on Django, Django-Rest (python based framework).

I am also a member of Vision and Language Group a student group aimed at spreading culture for reading and programming deep learning research papers in the campus.
`Cyberspectrum is the best spectrum`.

I started off with GNU Radio in January 2019. To get familiarized with the code, I made the following contributions to the code base:

1. Pull request #2339: gr_modtool: remove unused code in rename.py
2. Pull request #2338: Update README.md to provide instructions for building GR with support of python3.x
3. Pull request #2350: grc: fix for GRC's block hotkeys
4. Pull request #2352: grc: parse prefix for numeric values
5. Pull request #2373: grc: Fix save as for existing grc file
6. Pull request #2330: Add C++ generation support to gr-digital

7. Pull request #2343: Add C++ generation support to gr-filter
8. Pull request #2361: gr_modtool: Fix for parameter wrap in cli inputs
9. Pull request #2398: grc: Fix color for input boxes in parameter widget according to dark-gtk themes
10. Pull request #2412: GRC: Fix for GRC crash in case of flowgraph error
11. Pull request #2418: gr_modtool: fix for .yml file make template
12. Pull request #2423: gr_modtool: fix for yaml generator

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 extending the utility of this tool to different types of files, with the tool deciding how to manage each data file type by its own. I'll always be available for fixing the bugs that come up in the block header file tool.

Here is the link to my CV.

7.1 Other Details

Address	:	Roorkee, Uttarakhand, India
Email	:	guptarpit1997@gmail.com
Github	:	https://github.com/aru31/
LinkedIn	:	https://www.linkedin.com/in/aru31/
Codechef	:	https://www.codechef.com/users/aru31121997

8 Conclusion

Block header parsing tool would be a great addition to the set of tools for GNU Radio that works independently of the whole code. Since, block headers are the most direct source of information of GNU Radio blocks, parsing them directly from the source code itself would be of great utility.