

Google Summer of Code 2019

Block header parsing tool

Arpit Gupta

March 16, 2019

1 Introduction

Presently if a new block header file is created, we also need to create a YAML file in order to create a block which could be then viewed in the GNURadioCompanion (GRC). Also there must be mechanism to check the validity of the header (.h) files specifically to GNU Radio by parsing them. Thus implied we need a tool to parse them.

So, it would be great to create a tool which not only allows to create YAML files for the GRC but also could give us an abstract representation of the block by analysing the whole header file code, which would be in the form of a tree, thus making the code readable in an hierarchical fashion. Once we get the abstract structure, it could be easily parsed for any further use.

GRModtool is currently a powerful CLI tool and so this could be easily used for our purpose if the tool is built within it as another one of its utility.

The best tools that are presently being used to parse C++ in python are libclang with python and pygccxml, so these tools will be used to parse GNU Radio header files.

1.1 Primary features of the project

1. Implementation of both the parsing tools (libclang with python and pygccxml).
2. Abstract the implementation details of both parsers and creating abstract representation of the header files.
3. Extend GRModtool CLI to provide support for this tool. Make GRModtool to use this code itself instead of the built-in code.
4. Parse the abstract representation, create YAML files for GRC, make a script to change or add new optional YAML key parameters.
5. Extend its utility to take a list of files as input, or even a whole directory and parse them and also extend its utility to different file types, with this tool deciding how to manage each data-file type by its own (if time permits)

2 Proposed Workflow

Initially, I will be doing the ground work by setting up the abstract implementation of both the tools with the use of GRModtool command line interface. Then I will be working on parsing the abstract representation and storing information in a python dictionary. Finally, I will be creating YAML files for the GRC.

2.1 Implementation of parsing tools

As GNU Radio API (headers) require us to use C++ for compiling, not straight C, so we need parsers in python to parse C++, so libclang in python and pygccxml both do an excellent job for this purpose and are especially good for GNU Radio header files because of their brilliant, elegant yet simple and most importantly their generic structure.

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 and we are good to go. It automatically work it's way down the hierarchy 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:

```
1 from pygccxml import utils
2 from pygccxml import declarations
3 from pygccxml import parser
4
5
6 def pygccxml_parser(filename, gr_directory):
7     """
8     Implementation of pygccxml parser
9     """
10    generator_path, generator_name = utils.find_xml_generator()
11
12    # Configure the xml generator
13    xml_generator_config = parser.xml_generator_configuration_t(
14        xml_generator_path=generator_path,
15        xml_generator=generator_name)
16    filename = filename
17
18    # Parse the c++ file
19    decls = parser.parse([filename], xml_generator_config)
20
21    # Get access to the 'analog' namespace in 'gr'
22    global_namespace = declarations.get_global_namespace(decls)
23    ns = global_namespace.namespace("gr")
24    gr_block_type = gr_directory.split("-")[1]
25    gr_namespace = global_namespace.namespace(gr_block_type)
```

2.1.2 Implementation of Libclang with python

Libclang is a tool 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 pythonic way of implementing classes and functions to parse C++ files. This tool 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:

```
1 import sys
2 from clang.cindex import Index
3 from pprint import pprint
4
5
6 def get_cursor_id(cursor, cursor_list = []):
7     """
8     Cursor point to the current node
9     """
10    if cursor is None:
11        return None
12
13    for i,c in enumerate(cursor_list):
14        if cursor == c:
15            return i
16    cursor_list.append(cursor)
17    return len(cursor_list) - 1
18
19 def get_info(node, depth=0):
20     """
21     Recursively get info of all the nodes
22     """
23    children = [get_info(c, depth+1)
24                for c in node.get_children()]
25    return { 'id' : get_cursor_id(node),
26            'kind' : node.kind,
27            'usr' : node.get_usr(),
28            'spelling' : node.spelling,
29            'location' : node.location,
30            'extent.start' : node.extent.start,
31            'extent.end' : node.extent.end,
32            'is_definition' : node.is_definition(),
33            'definition id' : get_cursor_id(node.get_definition()),
34            'children' : children }
35
36
37 index = Index.create()
38 tu = index.parse(sys.argv[1], args=['-x', 'c++'])
39 pprint(('nodes', get_info(tu.cursor)))
```

Libclang parses all the standard C++ header files thus generating a very long AST which thus makes them 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 and thus exclude them from being parsed in the header

file itself.

2.2 Abstract implementation of parsers

It is due to the weird syntax of C++, all the current C++ parsers are not perfect, thus need for their abstract implementation. Abstracting the details of implementation of both the parsers is necessary so that we can replace them with some other tool or might even easily extend it, if need be.

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

```
1 def AbstractASTGenerator(parser):
2     """
3     Abstract parser implementation
4     """
5     try:
6         if parser.lower() == 'clang':
7             ClangASTGenerator()
8         else if parser.lower() == 'pygccxml':
9             PygccxmlASTGenerator()
10            # Can be easily extended futher if required
11    except Exception as e:
12        pass
13
14 def InitailParserInput():
15     """
16     Parser Type Input
17     """
18     parser_list = [
19         'clang',
20         'pygccxml',
21     ]
22     parser = input('Parser type(' + '/'.join(parser_list) + '): ')
23     if parser.lower() in parser_list:
24         ast = input('Do you require an AST file? (y/n): ')
25         if ast.lower() == 'y':
26             AbstractASTGenerator(parser.lower())
27             # Similarly other functions can be implemented
28     else:
29         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 it's functionality in form of such as python functions to get their make function, function calls, 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 tool in GNU Radio that provides a CLI for creating OOT modules. It provides various features, for example creating new OOT modules, adding various blocks, YAML generator and much more.

So, block header parsing tool being a command line utility can easily be integrated with GRModtool, also extend its code base for the new tool to be in use.

Another script can be called from CLI, for example from 'gr_modtool parse' to serve our purpose.

I have already gone through the code of GRModtool, which is brilliantly written thus making it really to extend it.

- 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 analysing the code pattern limited to GNU Radio headers, whereas compiler level parsing is required in order to achieve the above goal which is achieved by both the parsers.

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).

```
1 while not end_of_list:
2     # Keep track of (), stop when reaching final closing parens
3     if not in_string:
4         if c[i] == ')':
5             if parens_count == 0:
6                 if read_state == 'type' and len(this_type):
7                     raise ValueError(
8                         'Found closing parentheses before finishing '
9                         'last argument (this is how far I got: {})'
10                        .format(
11                            str(param_list))
12                    )
```

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

This subsection consists of the most important part of the whole project that is to correctly parse the header files by analysing all the function signatures, raising proper exceptions wherever required according to the convention followed in GNU Radio. Because of the use of two different parsers in this project, they generate AST in 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 form of tree type data structure because of the implementation of 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.

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 minor part of it, complete parser will be implemented as part of my project:

```
1 def namespace_parsing(filename, gr_directory):
2     """
3     Function to parse the namespace defined in the file
4     """
5     with open(filename, 'r') as file_object:
6         lines = file_object.readlines()
7         namespace_name = []
8         namespace_index = []
9         gr_block_type = gr_directory.split("-")[1]
10        for line in lines:
11            if 'namespace_t' in line:
12                namespace_name.append(line.split(":")
13                                     [1].strip().replace("\"", ""))
14                namespace_index.append(lines.index(line))
15            if namespace_name[0] == gr_block_type:
16                lines = lines[0: namespace_index[1]]
17            else if namespace_name[1] == gr_block_type:
18                lines = lines[namespace_index[1]: len(lines)]
19            else:
20                raise Exception('Wrongly specified namespace, must be
21                               {}'.format(gr_block_directory))
```

Similarly, 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 in python generates ast (txt file mentioned in section 2.2) as a JSON which is really easy to parse. 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 , we need to get all the child nodes with key-value pairs defined as "kind": CursorKind.PARM_DECL', from the JSON.

Thus we get all the possible variable, function declarations in the header file but we need their data types, return types too.

```
1 def traverse(node):
2     """
3     function that helps cursor traverse through all the nodes
4     """
5     for child in node.get_children():
6         traverse(child)
7     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 sorted with the use of parser architecture.

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 should be allowed to create YAML from them. 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.

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

- id, label, 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:

```

1 def file_name_parsing(file_name):
2     """
3     Function for generating YAML architecture
4     """
5     label_dict = {
6         'c': '(Complex)',
7     }
8     label_part = (file_name.split("_"))
9     if len(label_part[-1]) < 2:
10         if label_part[-1] not in label_dict:
11             label_part.pop()
12         else:
13             label_part[-1] = label_dict[label_part[-1]]
14     data = {}
15     data['id'] = file_name.split(".")[0].lower()
16     data['label'] = (' '.join(label_part)).title()
17     data['flags'] = ['python'] # optional cpp templates
18     data['file_format'] = 1
19     return data

```

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

A script for adding optional key-value parameters must also be implemented (in case required), for example if cpp flag is removed, cpp_templates key-value should be removed from the YAML file.

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, utility of the 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 of 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 it's completion. 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 June, July and August so I'll work full time during this period, i.e., around 35-40 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

- May 6 - May 27 • Define minute details of the project and build a sample parser architecture.
- May 27 - June 3 • Initialize the implementation of both the parsers to generate AST.
- June 3 - June 10 • Extend GRModtool to make it compatible with the new tool.
- June 10 - June 17 • Abstract the implementation of both the parsers, defining all the helper functions required.
- June 17 - June 24 • Complete the first three primary features of this project, and make GRModtool CLI ready to take proper inputs (of course still without actually text parsing).
- June 24 - July 1 • Start working on parsing pygxcxml generated AST.
- July 1 - July 8 • Start working on parsing libclang generated AST and complete the implementation of the previous one.
- July 8 - July 15 • Complete the implementation on parsing AST generated by both parsers with thorough testing.
- July 15 - July 22 • Creating YAML files for the GRC with properly raised exceptions.
- July 22 - July 29 • Make a script to edit the optional YAML keys.
- July 16 - Aug 5 • Thoroughly test the block header parsing tool, buffer time for completing the remaining tasks.
- Aug 5 - Aug 12 • 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.
- Extend Command Line Interface of GRModtool to support the new tool that would abstract the use of parsers.
- Properly parsing the generated AST and creating YAML files from command line itself for GRC.
- Implement the complete tool with thorough testing.

4.1 Milestones

- Phase-1: Making GRModtool capable of doing all the new tasks, abstract the implementation of the parsers.
- Phase-2: Successfully parsing AST generated by the parsers.
- Final Evaluation: Completion of the tool, 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 by other tools.

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:

- June 24: Merge code for extended GRModtool utility and abstract parser structure.
- July 22: Merge code for parsing AST generated by parsers.
- August 19: Merge the entire block header parsing tool along with it's extended utilities (if possible).

4.3 Automated 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 and will 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. The code will also be compatible with standard pylint rules.

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, 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 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 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 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 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

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 it's utility to different types of files, with the tool deciding how to manage each datafile 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 is one of the tools necessarily required because this would not only help in developing bug free header files, but also according to the conventions followed in GNU Radio that too from a CLI without actually using GRC. This tool itself could also be used as an API by any other tool (external to GNU Radio) to parse any header file. Also, no more manually creating YAML files for GRC.