Evaluation of Clang Tools for Information Extraction

Pontus Ljungren
Abstract

This thesis examines the potential of clang tools as a means of extracting and analyzing information from source code. The study focuses on the specific problem of translating enum-types from numerical values to their names, and examines different implementations of the translation tool using three different methods: LibClang, Clang Plugins, and LibTooling. The research also compares the clang methods to an existing translation tool implemented in Python, providing a comprehensive analysis of the effectiveness of the different solutions.

The results of the study demonstrate the capabilities and effectiveness of clang tools in understanding the context and functionality of a program. Additionally, it emphasizes that the built-in infrastructure and availability of APIs that are available when developing clang tools eliminates the need for starting development from scratch, in contrast to using python to develop tools. The findings suggest that Clang Plugins is more suitable for small-scale, frequent tasks, while LibTooling is more efficient for more in-depth analysis that can be conducted outside of the build environment. Overall, this thesis presents an in-depth examination of the use of clang tools and its potential in understanding and manipulating code.

Faculty of Science and Technology
Uppsala University, Uppsala

Supervisor: Lars Lundstedt
Subject reader: Konstantinos Sagonas
Examiner: Johannes Borgström
## Contents

1 Introduction ................................. 1
   1.1 Purpose and Goal ......................... 1
   1.2 Problem Description ...................... 2
   1.3 Methodology .............................. 4
   1.4 Outline .................................. 4

2 Background ................................. 5
   2.1 Clang .................................... 5
   2.2 Clang Compiler Structure ................. 5
   2.3 Clang Abstract Syntax Tree ............... 7
   2.4 Clang Tooling ............................. 7

3 Existing Method ......................... 10
   3.1 Run Python Script ....................... 10
   3.2 Python Script Content ................. 11
      3.2.1 Classes ............................. 11
      3.2.2 Main ................................ 12

4 Implementation .......................... 14
   4.1 LibClang ................................ 14
   4.2 Clang Plugins ........................... 16
   4.3 LibTooling ................................ 17
   4.4 ASTMatcher and the Recursive AST Visitor 19
      4.4.1 Recursive AST Visitor .............. 19
      4.4.2 ASTMatcher ......................... 19
   4.5 Generate Enum Translation Files .......... 21
5 Comparison

5.1 Existing Method vs. Clang Tools

5.2 Comparison Between Clang Tools

5.2.1 LibTooling vs. Clang Plugins

5.2.2 LibClang vs. FrontendAction Tools

5.3 ASTMatcher vs. Recursive AST Visitor

5.4 Comparison Overview Table

6 Suggestions

6.1 Python Script or Clang Tool

6.2 When to Choose Which Interface

7 Conclusion

References
Chapter 1

Introduction

In software development, it is quite often useful to analyze information available in code sections to help solve problems such as limitations and ambiguousness, or to issue warnings about questionable implementation decisions. An issue with this is how to find the desired information in these code sections to be further analyzed. There are a number of different methods for extracting specific information from source files (e.g. functions that return integers, variables with the name i or enum declarations). A high level language could be used to parse the code, e.g. a python script. Or a lower level method closer to machine code could be used, such as using the clang front-end infrastructure to write tools that analyze source files and extract information. It is not always an easy choice to pick which method to use since many factors play a part, for example how easy it is to develop the method, how powerful it is and how maintainable. In this thesis a couple of methods will be compared in the context of systems at Svenska Spel.

1.1 Purpose and Goal

The purpose of this thesis is to examine what is possible to achieve with the clang front-end infrastructure and its interfaces available for writing tools. A more specific purpose is to determine the best performing tool for extracting and analysing information from code. The goal is to develop multiple tools, where all enum definitions are obtained from a given source code to then generate C/C++ code that translates the numerical value of each obtained enum type to the corresponding enum string name. Another goal is to compare the available tools for generating
enum translation functions and determine the best method.

1.2 Problem Description

Svenska Spel has a problem when logging data from enumerations in their log-files. An enumeration, or enum for short, is a data type with a fixed list of possible constant values, called enumerators. A common example is the days of the week, see figure 1.1. A programmer can define their own enumeration type with specified enumerators and assign a variable with an enum constant, e.g. Monday. The only problem is that internally the compiler treats the enumerators as integer values; meaning that the integer which represents the enumerator is printed instead of the enumerator’s name when trying to output an enum [7].

```
typedef enum {
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday,
    Sunday
} WEEKDAYS;
```

Listing 1.1: Declaration of enum WEEKDAYS.

When Svenska Spel’s enum-types are printed in their log-files the interesting part is to know the name of the specific enumeration, not the integer value of the enum, see listing 1.2 and figure 1.1 to see why. Unfortunately the name is not available when debugging and trying to output this information. Therefore functions that return the enumerator name for a given value are needed.

An easy fix for this problem would be to manually write a toString-method for every enum defined, however, this would require too much work since Svenska Spel has thousands of different enumerations that need translation and many of them have hundreds of values. Therefore, it would be more practical if they could be generated, see listing 1.3 and figure 1.2.
```cpp
int main()
{
    WEEKDAYS today = Monday;
    WEEKDAYS tomorrow = Tuesday;

    cout << "Today is " << today << " and tomorrow is " << tomorrow << "." << endl;
    return 0;
}
```

Listing 1.2: Print of untranslated enum WEEKDAYS.

```
pahk@pahk-mac-HTD6 clang_examples % ./exampleEnumProblem
Today is 0 and tomorrow is 1.
```

Figure 1.1: Result of listing 1.2

```cpp
int main()
{
    WEEKDAYS today = Monday;
    WEEKDAYS tomorrow = Tuesday;

    cout << "Today is " << weekdaysToString(today) << " and tomorrow is " << weekdaysToString(tomorrow) << "." << endl;
    return 0;
}
```

Listing 1.3: Print of translated enum WEEKDAYS.

```
pahk@pahk-mac-HTD6 clang_examples % ./exampleEnumTranslation
Today is Monday and tomorrow is Tuesday.
```

Figure 1.2: Result of listing 1.3

Earlier, the company solved this problem by running a python script that searches the entire code base to find all specified enum-definitions and generate functions to translate them. However, since the compiler already parses all of the code, it seems redundant to repeat it with the python script. Therefore, the clang infrastructure could potentially be exploited to do this task and generate the desired functions intertwined with the clang compilation procedure.
1.3 Methodology

The first step of the project is an analysis of the existing python script. The second step is an explanation of three different methods of implementing a clang tool which solves the problem described above. The three methods are API’s called LibClang, Clang Plugins and LibTooling \[14\]. The methods are then compared with each other to see which tool is optimal under different circumstances. They are also compared to the python script, to reach a conclusion on which method is preferred for information extraction from source files. The comparison is done in regards to:

- Usability/Readability (learning curve, how easy the tools are to understand, read, use and run).
- Maintainability (how easy it is to fix errors/bugs and replace sections of code).
- Scalability (how easy it would be to implement a new feature, or to upscale an existing one).
- How powerful the tools are, meaning what they can accomplish and their restrictions.
- If it is better to use a standalone executable or a plugin which is run at compile time.

1.4 Outline

Chapter 2 covers background information about the clang compiler and its infrastructure, including the APIs used for tooling. Chapter 3 analyzes the existing python script. Chapter 4 covers how to implement tools with: LibClang, Clang Plugins and LibTooling. It also covers how to use the ASTMatcher and recursive AST visitor interfaces. Chapter 5 compares the different tools along with the python script. Chapter 6 makes conclusions on what method to pick for information extraction and which tools fit best for each task. Chapter 7 concludes and points to future work.
Chapter 2

Background

The first section of this chapter covers clang in general (section 2.1). The next
section overviews the stages of the clang compiler (section 2.2). The chapter
then introduces the notion of abstract syntax trees in clang (section 2.3). The last
section explains clang tools and what they can be used for (section 2.4).

2.1 Clang

Clang is a compiler front-end infrastructure, and it performs tasks such as Lexical
Analysis, Syntactic Analysis and Semantic Analysis. Clang can handle the family
of C languages (C, C++, Objective C/C++, OpenCL, CUDA, and RenderScript)
and translate code written in these languages to executable programs. However,
there are a lot of other possibilities and use cases with clang. Tooling is extra
actions that can be performed during the compilation procedure; such as extra
checks for uninitialized variables, warnings for hard coded values etc [16].

2.2 Clang Compiler Structure

A compiler has different stages and they can usually be divided into a front-end
and a back-end. Clang is a front-end environment and in order to form a complete
compiler it needs to be linked together with a back-end, and the LLVM Core
project has the back-end to the clang compiler [24].

Front-end tasks:
1. The lexer reads the source files and produces a token stream.
2. The parser creates an AST from the token stream.
3. The semantic analyzer adds semantic information to the AST.
4. The code generator produces an intermediate representation (IR) from the AST.

Back-end tasks:
1. The back-end performs target-independent optimization on the IR.
2. It then selects instructions for the IR code.
3. After, it performs target-dependent optimizations on the instructions.
4. Finally, it emits assembler code or an object file.

A programming language consists of a number of different elements, for example, keywords, identifiers, numbers and operators. The lexical analyser (often called lexer) is made to find these elements and create their corresponding tokens. After the lexer has read all source files produces a stream of tokens. The next part is syntactic analysis which is performed by a parser. The parser takes the token stream as input and uses a specified grammar to consume the tokens and create an abstract syntax tree (AST). If the grammar can consume the entire token stream and output a valid AST, it means the syntax of the source code is correct. The abstract syntax tree for clang will be explained more in section 2.3. The next part of the front-end compiler is the semantic analyzer which traverses the AST to check if semantic rules are followed, for example, if variables are declared before use, or if a comparison of variables is between compatible types. The semantic analyzer can also issue warnings to improve the source code, for instance, a warning can be made if a declared variable is unused. The last step of the front-end is the code generation which takes the AST as input to generate new code in a different intermediate language (IR). For clang, this language is llvm-code which is similar to assembly code, but with a couple of differences. With the IR code, the back-end has everything it needs to continue with the compilation.

The reason for dividing front-end and back-end is re-usability. With a front-end that always translates to the same intermediate representation (IR), the same back-end can be used even if the compilation starts with C-code or Rust for example, there is just a need for a different front-end interface. The same idea works the
other way around, if there is a need for C++-code to compile both on a x86-
processor and an ARM-processor, it can use the same front-end with different
back-ends [8].

2.3 Clang Abstract Syntax Tree

The clang abstract syntax tree (AST) is built up by three core classes, Stmt, Decl
and Type. A Stmt instance could for example be an if-statement or an assignment
(e.g. \( x = 3 + 5; \)). A Decl is of course any type of declaration, such as a variable
declaration (\( \text{int } x = 0; \)). A Type is also quite self explanatory, it could be an array-
type, for example, \( \text{int } A[5]; \) of type ConstantArrayType. All AST-nodes inherit
from one of these three classes, see figure 2.1 to follow some examples of how the
inheritance could look [4].

The root node for all AST-nodes is the TranslationUnitDecl-node. It represents
the translation unit for an AST. An example of how an AST can look for a small
example file example.c is shown in listing 2.1 and 2.2.

2.4 Clang Tooling

To write tools that need information located in the source code, parsing and anal-
ysis has to be performed. Looking at C++ code for example, it can be hard to do
since it is quite a complicated language structure. But there is no need to parse it,
as clang already can do this in an effective manner, furthermore, clang provides
clever interfaces to help write tools to manipulate this information. Three inter-
faces which will be explained further are LibClang, Clang Plugins and LibTooling
[14].

1. The first method is a high level C interface called LibClang which uses
API-functions to traverse the AST with a cursor.

2. Another method is Clang Plugins which builds dynamic libraries that can
be loaded at run time by the compiler.

3. The last method is LibTooling which is a C++ interface meant to write stan-
dalone tools possible to run outside of the build environment.

See more about the interfaces in Chapter 4.
Figure 2.1: Class hierarchies for the three core classes in clang.

Listing 2.1: An example C file `example.c`. 
Listing 2.2: A dump of an AST for a small C file example.c.
Chapter 3

Existing Method

This chapter covers how the existing python script developed by Svenska Spel to translate enum-types to their names in string format is implemented. First, it covers how to run the script (section 3.1). It then moves on to explain the classes HeaderFile, EnumData and Symbol and how they coincide (section 3.2). Lastly it explains how the script generates the enum translation files (section 3.2.2).

3.1 Run Python Script

To run the python script, a number of arguments are required to configure what the script should look for when parsing the files. The arguments available are:

- **-file** specifies the file that should be parsed.
- **-type** the name of the enum type.
- **-prefix** the prefix to all enumerators (e.g. drawstate in drawstate_DEFINED).
- **-dst** destination of the generated files.
- **-suffix** the suffix of the c-file name that is generated.
- **-cpp** whether a cpp-file file should be generated.
- **-cpp_dst** destination of the generated cpp-file, if parameter is left out the same directory as for c-file is used.
-cpp_suffix the suffix to the cpp-file name that is generated, if parameter is left out the same suffix as for c-file is used.

-c_asCpp true if c-file should be generated as a cpp-file type.

-enumlist the path to enumlist.txt which contains arguments given when running the script.

The required arguments that the script needs to run are file, type, prefix, dst and suffix. An example of a run could therefore be (listing 3.1):

```bash
$ python3 sample/gendesc.py --file sample/header.h --type DRAW_STATE --prefix drawstate --dst sample/getname --suffix its_generated
```

Listing 3.1: Command line argument to run python script.

Of course this can be built with CMake by entering the command into a CMakeLists.txt but every time translation of new enums is needed the arguments have to be changed. To put in more than one enum to translate at the time the argument specification has to be repeated. The number of type, prefix and file arguments also have to be the same. See listing 3.2 where two enums are translated.

```bash
$ python3 sample/gendesc.py --file sample/header.h --type DRAW_STATE --prefix drawstate --file sample/header.h --type WEEKDAYS --prefix weekdays --dst sample/getname --suffix its_generated
```

Listing 3.2: Command line argument to run python script.

### 3.2 Python Script Content

The python script has to perform three tasks, first it has to handle the command line arguments, then it has to read the file for information, and lastly it has to use the retrieved information to generate a c-file. To help with these tasks there are three classes; Symbol, EnumData and HeaderFile.

#### 3.2.1 Classes

**HeaderFile**

The HeaderFile class is in charge of parsing the file to find the wanted enum definition. The parse_file method does this by using the python library re to search the file with regular expressions. If an enum is found an EnumData
instance is created, the enum name is checked against the meta data (the enum type) and if it is the wanted enum, it is further parsed by using the EnumData method parse_lines.

The class also contains a method add_to_c_files that is in charge of generating the c-file as well as the header files for each enum-type translated. This method uses the data about the enum gathered in the EnumData instance, as well as general print methods to output string formatted c-code to an output file.

**EnumData**

This class is instantiated in the `HeaderFile::parse_file` method and is given a number of lines of source code stored in the class variable `lines`, as a list of strings. The lines contain the code from inside the enum definition, i.e. the enumerators. Once the `parse_lines` method is called the lines are processed one by one by matching them to a regular expression that divides the enum members into groups: prefix, name and description. These groups are initialized as `Symbol` class instances. It then appends each group to the symbols class variable to later be able to generate the translation code.

Finally, `parse_lines` calls `generate_function_headers` to create all the function header strings in class variables to be able to generate the header-files later.

**Symbol**

This class contains the class variables `prefix`, `name`, `full_name`, `var_name_str`, `var_desc_str` and `desc`. There are also a number of methods that return formatted strings to help with the generation of the c-files. A `Symbol` class instance is used to store information from an enumerator.

### 3.2.2 Main

The script starts execution by parsing the arguments given in the command line and storing them in variables for later usage. The next step is to initialize `HeaderFile` instances for all enumerations in all the given files. The meta data or the prefix for every enum is also entered into the class instances. Now the files are ready to be parsed in order to gather information about every enum and save it to the class instances. This is done by calling the method `parse_file` on every file. The following stage is to create files where the generated code will go, which is done by
naming the files with some of the previously stored arguments. Some includes are written to the file before all code is generated by the `add_to_c_files` method. See figure 3.1 for a visual description of the implementation of the python script.

Figure 3.1: Flowchart of the python script that generates enum translation functions.
Chapter 4

Implementation

This chapter covers how to implement a clang tool in three different ways. It will not be a deep dive into the technical level on how every tool is implemented, but rather it provides an overview of how they work. First the LibClang implementation will be explained (section 4.1), then Clang Plugins (section 4.2), moving on to LibTooling (section 4.3), the next step is the ASTMatcher and the recursive AST visitor interfaces (section 4.4), and lastly an explanation how translation files are generated is given (section 4.5).

As part of this thesis the implementation is a tool that extracts all enum definitions from a code base, gathers specific data from enumerators, and generates source code for translating an enum type’s value to its string name.

4.1 LibClang

LibClang is a C-interface that provides API-functions to parse the AST, traverse it and to find out information about it. The idea behind LibClang is to stay relatively small, and not get too complicated. In return, the interface can stay stable between new releases. It only provides basic functionality needed to support development tools. Data types are prefixed with "CX" and functions with "clang_" to avoid conflicting names with other interfaces [21].

A LibClang tool that is going to traverse the AST needs to be set up in a certain way (see listing 4.1). First an index is created, where the index data structure contains different source files and information about how they are executed (line
9). It is essential to create a new translation unit that is going to be traversed. The translation unit can be created a couple of different ways, in this example one is created by parsing a source file given from the command line (line 10-14), but it can also for example be done by giving it an already parsed .ast file with createTranslationUnit. The next step is to create a cursor which is the LibClang data type that is used to traverse the AST (line 20). clang_visitChildren is used to traverse the entire AST by calling the function with the root cursor i.e. the translationUnitDecl-node (line 20-22). clang_visitChildren thereafter visits all immediate children of that node and invokes the visitor function, and depending on what the visitor function returns the traversal can either recurse down the AST, continue to the next immediate child or terminate immediately. After the traversal is done the memory has to be freed with clang_dispose functions to avoid memory leakage (line 24-25) [9].

```cpp
int main(int argc, char **argv) {
  if (argc < 2) {
    cout << "Add the source file you wish to parse as an argument";
    return -1;
  }

  CXIndex index = clang_createIndex(0, 1);
  CXTranslationUnit unit = clang_parseTranslationUnit(
    index,
    nullptr, argv, argc,
    nullptr, 0,
    CXTranslationUnit_None);
  if (!unit) {
    return -1;
  }

  CXCursor rootCursor = clang_getTranslationUnitCursor(unit);
  vector<EnumData> *enums = new vector<EnumData>();
  clang_visitChildren(rootCursor, visitor, enums);
  clang_disposeTranslationUnit(unit);
  clang_disposeIndex(index);
  return 0;
}
```

Listing 4.1: How to traverse the AST with LibClang.

The enum translation tool finds all enum definitions by a traversal of the entire AST, when the CXCursor reaches an enum declaration it saves the name of the declaration in an EnumData class instance and continues to move down the syntax tree. It will then visit the enum members of type CXCursor_EnumConstantDecl,
and save all the information about the members in the EnumData class instance declared earlier. This way a class instance will be created for all enums found and all information about them will be saved \[6\].

To execute the LibClang tool, simply invoke the executable with the file to be parsed as an argument. Listing 4.2 shows how to run the enum translation tool.

```
$ ./bin/ExtractEnumsLibClang test_src/HelloWorld-basic.cpp
   -fparse-all-comments
```

Listing 4.2: How to run the enum translation tool with LibClang.

### 4.2 Clang Plugins

Clang Plugins allow for extra user defined actions to be performed during a compilation, such actions could for example be a couple of extra semantic checks like checking for uninitialized variables and giving them their default value. The idea with Clang Plugins is that it should be really easy to integrate the plugin into the build environment, "as easy as adding a flag to the compilation" as stated by Cardoso Lopes and Auler (2014, p.270) \[4\]. As a result, it is convenient to use Clang Plugins when a tool should be run for every compilation. However, it is important to keep the execution time short, because in order for it to be sensible to run the tool for every compilation, it can not take up too much execution time and become a bottleneck \[4, 15\].

To write a Clang Plugin, a FrontendAction is needed, which is a class for implementing user specific actions during the compilation. To run a clang tool over the AST, the FrontendAction is executed. Clang Plugins provides a specific FrontendAction interface called PluginASTAction, which declares a function ParseArgs, which lets the user handle arguments from the command line. To be able to run a Clang Plugin, the PluginASTAction needs to be registered as a plugin, which is done in listing 4.3 \[19\].

```
static FrontendPluginRegistry::Add<MyPlugin> X(  
   "my-plugin-name", "my plugin description"
 );
```

Listing 4.3: How to register a plugin MyPlugin as a library that can be loaded by the compiler \[19\].

A call to the function prototype in listing 4.3 builds the plugin into a library file.
that can later be accessed by the compiler to load and execute the plugin. It does this by saving a shared library file (file extension .so) to the build directory.

To use the Clang Plugin enum translation tool, the library file must be loaded and the plugin added. The tool can then be executed, as shown in Listing 4.4, to produce an executable for the parsed file.

```
$ clang++ -fplugin=bin/enum_definitions_extraction/recursiveASTVisitor/libExtractEnumsRecursiveASTVisitor.so -fplugin-arg-extractEnumsRecursiveASTVisitor header.h -fparse-all-comments
```

Listing 4.4: How to run the enum translation tool with Clang Plugins.

### 4.3 LibTooling

LibTooling is a library to support writing clang tools, and it allows for creation of standalone tools, as opposed to running tools as a part of the compilation process like Clang Plugins. Implemented tools already exist on top of the LibTooling interface, and some of them are available in the Clang Extra Tools package, such as Clang tidy which can perform a number of different checks [17][22].

Like Clang Plugins, LibTooling utilizes a FrontendAction, which will be run over some code, but it is implemented in a different way. See listing 4.5 where the SyntaxOnlyAction FrontendAction is implemented as a standalone tool.
Listing 4.5: The frontendAction SyntaxOnlyAction implemented with LibTooling.

For a standalone tool to use clang, it first needs to figure out what compiler commands to use for a specified file that needs parsing in order to be able to generate the correct AST. To that end, a compilation database is required. There are different ways to create a compilation database, either by creating a compile_commands.json configuration file or by specifying needed parameters in the command line. The CommonOptionsParser class takes responsibility to parse command-line parameters related to compilation databases and inputs [22, 18, 20, 25].

To execute a tool implemented with LibTooling, the tool’s executable file can be run from the command line. Any necessary arguments can be passed at this time. For example, the command to invoke the enum translation tool can be found in listing 4.6.

Listing 4.6: How to run the enum translation tool with Libtooling.

```bash
$. /bin/enum_definitions_extraction/ASTMatcher/Extract EnumsASTMatcher header .h --extra-arg="-fparse-all-comments"
```
4.4 ASTMatcher and the Recursive AST Visitor

When implementing LibTooling or Clang Plugins there are two options to choose from on how to develop the FrontendAction of the tool. One is to use a recursive AST visitor which traverses specified nodes to perform some tasks inside the `visitNodeType` functions [19]. The other option is to use ASTMatcher which is implemented with predicates to match certain AST-nodes and a callback function that is called when a node matches [23]. For this thesis both options have been implemented.

4.4.1 Recursive AST Visitor

To access and run the recursive AST visitor, an AST Consumer class is needed. This class exists as an abstraction layer to abstract out how the AST was created, e.g. if the AST was generated by the parser or if it came from an AST-dump file. Whenever an entire translation unit has been read, the `HandleTranslationUnit` function is called by the ASTConsumer, and it is here the traversal of the AST should start with a recursive AST visitor. As mentioned, `visitNodeType` functions are used to visit the AST nodes. They are provided as "hooks", meaning they will be called automatically when a `traverseNodeType` method is called with a recursive AST visitor instance [19]. To traverse the entire AST, the `TraverseDecl` method can be called with the `translationUnitDecl` node as its argument. See a flowchart of how the recursive AST visitor can be implemented in figure 4.1.

4.4.2 ASTMatcher

The Clang team implemented ASTMatcher "to provide a simple, powerful, and concise way to describe specific patterns in the AST" (n.d., para. 3) [25]. The idea is that AST nodes can be found by writing patterns to match specified conditions, e.g. `traverse(enumDecl().bind("enum"));` will match all enum declarations.

To implement an ASTMatcher all that is needed is a matcher pattern and a callback function for that matcher. They are paired together when they are added to the `MatchFinder` class with the `addMatcher` method. Once all matchers have been added, an ASTConsumer is created. When this ASTConsumer is executed it will trigger a call to a paired callback function whenever a match is found. A clang tool using ASTMatcher can be created by initialising a FrontendAction with this ASTConsumer [13].
Figure 4.1: Flowchart of ExtractEnums recursive AST visitor implementation.

There are matchers for matching specific nodes, nodes fulfilling certain criteria, and even matchers for getting nodes related to another node. A specific node in the matcher pattern can be bound with the bind method. That way it is easy to access a specific node found in a matcher pattern inside the callback function.

The ASTMatcher library is implemented by using a recursive AST visitor to traverse all nodes in the AST. For every visited node, every matcher pattern in the MatchFinder class is compared to see if it is a match. In other words, the ASTMatcher library is built on top of the recursive AST visitor interface [12].

Once the EnumMatcher in the enum extraction ASTMatcher tool matches (determined by the function Matches to an enum declaration, the enum is bound
to "enum". The callback function EnumCollector::run which is triggered by the match, can therefore access the enum declaration by referencing the bounded string "enum". Now, it only needs to extract all the information from the enum and save it to an EnumData class instance, in order to later be able to generate the enum translation functions, see figure 4.2 for a visual explanation of AST Matchers.

Figure 4.2: Flowchart of ExtractEnums ASTMatcher implementation.

### 4.5 Generate Enum Translation Files

To generate the enum translation files, a class called EnumData is used. An EnumData instance is created and eventually filled with necessary data for every enum that is found in the source files. All EnumData instances are stored in a variable std::vector<EnumData> enumsRetrieved. To later generate the translation functions a method writeEnumTranslationToFile is called for every element in the enumsRetrieved array.
Chapter 5
Comparison

As seen earlier in the report, the existing solution for the enum translation is a python script. In this chapter the existing method will be compared to the clang tool implementations (section 5.1) and the multiple clang tool implementations will be compared against each other (section 5.2). When using LibTooling or Clang Plugins, there are two options to choose between when implementing the tools; ASTMatcher or a recursive AST visitor. These options will also be compared (section 5.3). Finally, an overview of comparisons between different implementations in terms of multiple criteria will be presented (section 5.4).

5.1 Existing Method vs. Clang Tools

There are a lot of differences between using scripts and the clang infrastructure for extracting information. The python script does not have any context about the program, except for the text of the source files which does not say anything unless analyzed. When using clang however, the AST along with a lot of other context information is available, giving more freedom and ease to draw conclusions about the program (e.g. to tell whether a variable is in scope).

For example, a case that is not be covered by the python script is automatically covered by clang; if there is a #ifdef directive inside an enum definition, see listing 5.1. When clang builds the source files that is going to be analyzed it stores the commands for how to build them in a compile_commands.json file (if specified in the build system). This way the tools can know how to compile the
typedef enum {
    Monday,
#ifdef ALIEN_PLANET
    Alienday,
#else
    Tuesday,
#endif
    Wednesday,
    Thursday,
    Friday,
    Saturday,
    Sunday
} WEEKDAYS;

Listing 5.1: An enum definition with a conditional directive inside.

files and get the correct AST. Looking at listing 5.1 it will only be known if the Alienday member is defined after the preprocessor has run, therefore the python script will not discover the #ifdef and take the Alienday enumerator regardless of whether ALIEN_PLANET is defined or not. For the python script to support the conditional directives, a lot of further development would have to be made and the defined macros from compilation would have to be given to the script somehow (e.g. as a flag --defined -DALIEN_PLANET).

The existing python script will take both enumerators with the same value (Tuesday and Alienday), and add them to the translation function. Resulting in one of the members not being defined even though it is present in the enum translation file. This will cause a problem when trying to compile the translation file for usage (see error message in figure 5.1). Clang would on the other hand discover the #ifdef as the preprocessor is run before the AST is created and thus choose the correct enumerator for the enum translation file [4].

The python script has the potential to invoke a preprocessor on the c++-code in order to avoid the conditional directive problem, but it does not do so automatically. This means that the developer must manually invoke the preprocessor and then

Figure 5.1: Error message caused by conditional directive inside enum when using python script.
parse the resulting file. On the other hand, the clang tools are able to handle this issue automatically, allowing for a more seamless and efficient process. This is an example of how the clang tools provide a more comprehensive and user-friendly solution for analyzing and manipulating C++ code.

Considering clang tools are built from the clang interface it is quite straightforward to integrate them to the build system as it can run as a part of clang compilation. To build clang tools a compilation database needs to be created, which can either be done by creating a `compile_commands.json` configuration file or by specifying needed parameters in the command line after a ' - ' separator. If the `compile_commands.json` exists the tool can simply be called with the input file that is to be parsed. If wished, the input file does not even need to be recompiled when the tool is executed. The tool can for instance use a saved AST-file as input if the `-emit-ast` command was used during a previous compilation [18, 20, 22, 2]. Whereas, with a python script re-parsing of the input file is standard because the parsing happens separately from compilation. This can be inefficient and time-consuming, especially for larger projects. However, it is possible to avoid re-parsing by using a library to compile the code first and then using the resulting AST file as input for the python script similarly to clang. This way, the script can parse the AST file instead of the original source code, which can significantly improve performance and efficiency. It allows the python script to take advantage of the benefits provided by ASTs, such as faster parsing and more accurate analysis. However, this approach requires additional setup and potentially more development effort since you have to write new python code to analyze an AST-file like the clang interface already can.

To find an enum definition with the existing python script at Svenska Spel the parsing of the file is written manually. Listing 5.2 shows how a part of this is achieved. Each line in the file is compared to a regular expression "typedef\s+enum\s*" in order to find the start of an enum definition. If the enum is not combined with the typedef it will not catch the enum definition. The expression could be altered to find the enum regardless, but it would lead to an even more complicated expression. Every line inside the enum is stored until the end of the enum is reached by matching the expression "\}\s*([a-zA-Z0-9_]*)\". To achieve the same result as the python script with a clang tool requires less code (see table 5.1). Furthermore it is easier to read, which makes it easier to develop. To compare the python script with a clang tool in terms of readability and usability; notice the difference between the python method (5.2) and a clang tool method (5.4 and 5.5).
for line in f:
    if re.search(r"typedef\s+enum\s+\*", line) != None:
        self.parsing_enum = True
        self.ed = EnumData()

    if self.parsing_enum:
        self.ed.add_line(line)

    m = re.search(r"\}\s*([a-zA-Z0-9.]+)\", line)
    if m != None and self.parsing_enum == True:
        self.type_name_found = m.group(1)
        self.ed.set_type_name(self.type_name_found)
        self.parsing_enum = False

    if self.type_name_found in self.enum_meta_data:
        self.ed.parse_lines(self.enum_meta_data[self.type_name_found])

    self.enum_data.append(self.ed)

Listing 5.2: Section of python script that finds code section containing enum definition.

<table>
<thead>
<tr>
<th></th>
<th>Python script</th>
<th>Clang tool (LibTooling with ASTMatcher)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total number of lines</td>
<td>260</td>
<td>130</td>
</tr>
<tr>
<td>Find enum declaration</td>
<td>16</td>
<td>10</td>
</tr>
<tr>
<td>Set up</td>
<td>50</td>
<td>9</td>
</tr>
</tbody>
</table>

Table 5.1: Comparison between number of lines in python script vs. LibTooling tool with an ASTMatcher. The number of lines is approximate as the python script has some other functionality that is not implemented for the tool.

The clang infrastructure comes with more functionality than just extracting the parsed data, it also has methods for handling and manipulating it which makes it more powerful than using python scripts, e.g. Svenska Spel has a certain standard naming convention that needs to be followed. This could be controlled and automatically fixed with clang tools if desired. This would require a lot of effort to achieve with python, as API’s are not available. In general the clang infrastructure is more scalable for writing tools compared to scripts due to the API’s available with clang. This vastly reduces the development time and gives more opportunities to develop new tools with different functionalities.

A benefit with the python script is that python rarely changes so drastically that a new version of python would break the script. Therefore, if the script does what it is supposed to do, the code is very easy to manage as it probably will not need much modifications. However, clang tools occasionally get updated and there is no guarantee that this will not break already implemented tools. This could lead
to some need for maintenance upon updates. With LibClang this is not as much of a problem since the Clang team tries to keep this interface stable. Overall maintenance for python scripts is easier. Also, more developers are familiar with python than with the clang infrastructure.

5.2 Comparison Between Clang Tools

Svenska Spel’s method is comparable to using clang tools, but there are also a lot of options when choosing between methods that clang tools has to offer. Different methods are LibClang, Clang Plugins and LibTooling. All methods were implemented to be able to weigh advantages and disadvantages between them.

5.2.1 LibTooling vs. Clang Plugins

LibTooling and Clang Plugins are quite similar in implementation as they both use a FrontendAction. Clang Plugins uses a PluginASTAction (a subclass of ASTFrontendAction) and LibTooling uses an ASTFrontendAction.

The main difference between the interfaces is that building a Clang Plugins tool creates a library (.so file) that is available to load and run on input files during compilation run time. This makes it really easy to integrate to the build system as the plugin can be run by just adding a couple of flags to the compilation (see listing 5.3). Due to this functionality Clang Plugins fits very well for tools that need to be run for almost every build. The tools should have short execution time to not delay the build system and preferably not have too much output, since compilation output should be clean to be able to see if everything compiled correctly.

LibTooling on the other hand creates an executable that can be run separately from compilation when built and therefore separate from the build system (see listing 5.3). As a result, LibTooling fits better for deeper analysis as it only needs to run when needed, meaning that the execution time can be longer, and more complicated actions can be performed. LibTooling is great for refactoring tools, e.g. if there is a wish to change variable names to follow a naming convention, or to automatically declare a default value to all undeclared variables [14].

In listing 5.3 the first command runs the Clang Plugins implementation, and the second command runs the LibTooling implementation. The clear difference is that Clang plugins also run normal compilation for the file_to_analyze.cpp file,
which will produce an executable. The second command is preferably run outside of the build environment and does not produce an executable.

\[
\text{
$\text{clang++ -fplugin=path/to/library/file/library_file.so -fplugin-arg=<plugin_name>-<arg> file_to_analyze.cpp}$

$\text{path/to/executable/tool/tool -p $BUILD_PATH file_to_analyze.cpp}$
}
\]

Listing 5.3: Commands to run clang plugins and LibTooling.

With LibTooling there is a little more freedom than Clang Plugins as LibTooling has complete power over how clang is set up and has global information of programs, such as global variables across multiple files. Clang Plugins only runs the tool over one file at a time and does not store global information. With LibTooling there is also a main function, therefore some tasks can be performed before and after execution of a tool. For example another tool defined in the main function could be triggered to run if certain conditions are filled for the first tool [3]. With Clang Plugins this is not possible since the FrontendAction is executed for every source file, and there is no way to execute a task before or after execution of the entire tool. This makes LibTooling more powerful than Clang Plugins.

5.2.2 LibClang vs. FrontendAction Tools

Above, Clang Plugins and LibTooling were compared, which are both tools that use a FrontendAction, meaning that they are quite similar. Now LibClang, which somewhat differs from these will be compared to both Clang Plugins and LibTooling simultaneously. Henceforth, Clang Plugins and LibTooling combined will be called FrontendAction tools.

LibClang’s aim is to be a stable interface, which means it should not break when new versions of LibClang are released. The clang team who develops LibClang strives to keep this interface working by maintaining the same API functions working in newer releases of LibClang, i.e. they strive for good backward and forward compatibility. This is not the case for the FrontendAction tools. There is no guarantee that old implementations of tools will work when newer versions are released. This means that the maintainability for LibClang tools is better and requires less work than FrontendAction tools.

On the other hand, LibClang does not have as much control over the AST as the FrontendAction tools. LibClang uses a cursor to traverse the AST, and there is no way to alter the AST. For example, if the name of an enumerator should be
changed to follow a certain naming convention it is not possible because LibClang does not have permission. The FrontendAction tools on the contrary has full control over the AST and therefore allows the developer to change the AST majorly if needed [4]. Due to this limitation, LibClang can not scale as well and it can not introduce new functionality as opposed to the FrontendAction tools (i.e. FrontendAction tools have better scalability and are more powerful).

### 5.3 ASTMatcher vs. Recursive AST Visitor

ASTMatcher has functionality that is largely the same as the recursive AST visitor interface, as it is built on top of it [12]. One advantage of using ASTMatcher is that the recursive AST visitor class does not need to be defined, resulting in a reduction in both development time and code volume. On the other hand, the use of a callback function is not required when utilizing the recursive AST visitor interface, as actions can be performed during the call to the visitNodeType function. In terms of overall code volume, the ASTMatcher interface requires fewer lines of code, as demonstrated in Table 5.2 in the context of this thesis’ implementation.

<table>
<thead>
<tr>
<th>(Number of lines)</th>
<th>ASTMatcher</th>
<th>Recursive AST Visitor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>130</td>
<td>166</td>
</tr>
<tr>
<td>Find enum declaration</td>
<td>10</td>
<td>23</td>
</tr>
<tr>
<td>Set up interface</td>
<td>9</td>
<td>25</td>
</tr>
</tbody>
</table>

Table 5.2: Comparison between number of lines when implementing ASTMatcher and recursive AST visitor for the enum extraction tool.

But the better method is not always determined by the lines of code required to develop a tool. It is also about readability and usability; meaning the better tool is easier to read and understand. For smaller programs ASTMatcher can be quite easy to understand as it is closer to english sentences than using a recursive AST visitor. Observing the differences between the methods, it is noticeably easier to understand the ASTMatcher pattern. Notice the differences between listings 5.4 and 5.5, which showcase the ASTMatcher implementation, and listings 5.6 and 5.7 which demonstrate the recursive AST visitor implementation [11].
Another factor when determining the better method is scalability. For bigger programs it can be messy to write multiple matchers and multiple callbacks. For example, if another pattern to extract all method declarations was added to the tool. With ASTMatcher, a new matcher and callback would have to be added like in listing 5.5. But with the recursive AST visitor, only a new visitNodeType function (see listing 5.8) would have to be added to the ExtractEnumsVisitor class in listing 5.7.
bool VisitCXXMethodDecl(CXXMethodDecl *Decl) {
  "enter code"
}

Listing 5.8: Find enum declaration with a recursive AST visitor.

If there was a need to express some logic between two different types of nodes that are separate in the code, it would not be possible with ASTMatcher. With the recursive AST visitor the visitNodeType functions are in the same scope, which allows access to a class variable for example. With ASTMatcher, two different matchers will have different callbacks, therefore they will be in different scopes and they will not have access to the same class variable [26]. Therefore the recursive AST visitor is more powerful than ASTMatcher.

5.4 Comparison Overview Table
<table>
<thead>
<tr>
<th>Tools</th>
<th>Usability/Readability</th>
<th>Maintainability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python Script and Clang Tools</td>
<td>Clang Tools are easier to use due to the interfaces available that abstracts away a lot of development. Available API-functions can be used instead of developing the functions manually with python.</td>
<td>Python requires less maintenance since updates on clang tool’s interfaces can break already implemented tools.</td>
</tr>
<tr>
<td>LibTooling and Clang Plugins</td>
<td>Clang Plugins is slightly easier to use as less code is required to set up the interface.</td>
<td>No difference.</td>
</tr>
<tr>
<td>LibClang and FrontendAction</td>
<td>Not a lot of difference if the developer is familiar with both interfaces. However, LibClang has a shorter learning curve than FrontendAction tools.</td>
<td>LibClang is easier to maintain because the clang team strives to keep this interface stable. FrontendAction tools can break upon updates of the interfaces.</td>
</tr>
<tr>
<td>ASTMatcher and Recursive AST Visitor</td>
<td>For smaller tools it is easier to use ASTMatcher because the interface is more readable when there are not too many matchers. For more complicated tools the recursive AST visitor is more readable and easier to use because of the structure that comes with wrapping everything inside a class.</td>
<td>No difference.</td>
</tr>
</tbody>
</table>

Table 5.3: Comparisons between tool implementations in terms of criteria.
<table>
<thead>
<tr>
<th>Scalability</th>
<th>How powerful</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python Script and Clang Tools</td>
<td>Clang tools are more scalable than python scripts, because with a python script everything has to be developed from scratch, and with clang tools the interface can be used to introduce new functionality.</td>
</tr>
<tr>
<td>LibTooling and Clang Plugins</td>
<td>LibTooling is more scalable because Clang Plugin tools are supposed to be run as a part of compilation and be kept short to not become a bottleneck to the compilation.</td>
</tr>
<tr>
<td>LibClang and FrontendAction</td>
<td>There are less possibilities to scale LibClang because less new functionality can be introduced.</td>
</tr>
<tr>
<td>ASTMatcher and Recursive AST Visitor</td>
<td>A recursive AST visitor is more scalable because less effort is required to introduce visitation of a new AST node, and the structure is easier to follow when the tool scales.</td>
</tr>
<tr>
<td></td>
<td>Everything done with ASTMatcher can be achieved with a recursive AST visitor. But there is some functionality that can be achieved with recursive AST visitor that can’t be done with ASTMatcher. As a result the recursive AST visitor is more powerful.</td>
</tr>
</tbody>
</table>

Table 5.4: Comparisons between tool implementations in terms of criteria.
Chapter 6

Suggestions

This chapter will present a recommendation on whether the existing python script method should be kept or whether one of the new implementations with clang should be used. It will also cover what clang tool implementation is best to use for different scenarios (section 6.2).

6.1 Python Script or Clang Tool

This thesis suggests it is always better to choose a clang tool over a python script when analyzing c-family source files. The infrastructure that comes with clang makes it superior to develop tools as it reduces development time, increases readability and scalability. The maintainability is not necessarily better with clang tools compared to a python script, but there are still more advantages than disadvantages. For python to work as well as clang tooling, an entire interface for tooling would have to be developed, and this is unnecessary as it already exists and works well for clang.

6.2 When to Choose Which Interface

For the case of the enum translation tool, the conclusion is that the Clang Plugins implementation is the better choice. This is because the tool currently is quite small and the execution is quick, therefore it works well to generate the functions for every compilation. The ASTMatcher interface is preferred over the recursive
AST visitor, due to the fact that it is slightly easier to read and learn for a small tool like this one.

**Clang Plugin**’s main advantage is that it is easy to integrate the tool to the build system, therefore it should be used for tools that need to be a part of the regular compilation procedure. The tool should have low execution time, in order for it not becoming a bottleneck when compiling a project. The Clang team recommends Clang Plugins for tools that includes: ”creating additional build artifacts from a single compile step” (n.d. para. 3) [14]. These tools usually have quite fast execution time due to the fact that they do not alter the AST in any way.

**LibTooling** is the tooling interface with the most freedom. It leaves the developer with more power over how to customize the tool to fit specific needs. LibTooling is the best choice when developing more extensive tools, for example, refactoring tools. A naming convention correction tool would be a great example where LibTooling would fit.

**LibClang** fits best for a tool which will not scale, and that does not alter anything in the AST. Since LibClang strives to keep a stable interface it requires less maintenance, which can be good if there are no resources available to maintain the tool.

**ASTMatcher** is quite easy to learn and understand. For minor tools it is better to use as the code is quite readable. For bigger tools that need access to many different nodes the interface quickly becomes messy and less desirable.

**Recursive AST Visitor** is in a way the opposite from ASTMatcher. For small tools there is a lot of boilerplate code and it can be quite hard to follow. But for bigger tools it is easier to structure the tool and it becomes quite convenient. For example, the recursive AST visitor is the more appropriate interface for a naming convention tool that not only checks that enumerations follow the naming convention, but rather that the entire code follows a naming convention.
Chapter 7

Conclusion

This thesis set out to explore the clang infrastructure and how to implement clang tools. This was done by developing three different implementations of a clang tool; the implementations being LibClang, Clang Plugins and LibTooling. The tools were implemented to solve a problem involving enumerations. The problem is when enum types are printed, they are output as integer values instead of as enumerator names. This problem was previously solved by using a python script that parses source files and generates functions to translate enum values to their corresponding name. The implemented clang tools achieves the same result.

The python script was evaluated along with the clang tools implemented. A comparison was made between them and the conclusion was that using a clang tool to solve this problem is to be preferred over using the python script. The reason was that it is easier to implement tools with clang since API’s are available to reduce the development time. The different clang implementations were also compared to each other in order to determine which tool is suited best for which situation. A clear result of which tool to choose depending on circumstances was reached.
References


