Clang LLVM frontend as a modern C++ source-code generation tool

Research and Development Project

Name	Student Number
Morten Haahr Kristensen	201807664
Mikkel Kirkegaard	201808851

Supervisor	Email
Lukas Esterle	lukas.esterle@ece.au.dk

Department of Electrical and Computer Engineering
Aarhus University
May 30, 2023

Abstract

Contents

1 Introduction

1.1 Preface

2 Project description

This chapter provides an overall description of the project including a brief overview of the main technologies used.

The overall topic of the R&D project is source-code generation, which derives from automatic programming. Automatic programming can be defined as the automation of some part of the programming process [1]. The process consists of parsing a specification as an input to the automatic programming system which outputs a program [2].
An example of an automatic programming system could be a compiler, where the specification is a program written in the desired programming language and the output is an executable program. Another example could be the tools provided by Visual Paradigm that translates the contents of UML diagrams to source code of different programming languages [3].
The motivation behind automatic programming is that it allows developers to express themselves more abstractly through specifications, allowing for smaller, more understandable and less error-prone programs [2].

Source-code generation can be considered a specific area of automatic programming where the output of the automatic programming system is source code. The motivation for source-code generation is similar to that of automatic programming, however, the output is a refined or generated specification rather than an executable program.

This R&D project investigates the usage of the library LibTooling for writing deterministic C++ source-code generation tools. The project involves the writing of three separate tools with increasing complexity that addresses some issues related to writing source-code generation tools through LibTooling.

The first tool is a simple renaming tool that can be used to rename functions and their matching invocations.
The second tool is also a refactoring tool which transforms arrays specified using the C-style notation into the more modern and secure `std::array` notation.
The third tool generates a `to_string` function for each enum declaration inside the program, i.e., a function that takes an instance of that enum as an argument and returns a string corresponding to the name of the value of the enum. If an existing `to_string` function exists, e.g., in another namespace, the tool updates it in place instead.

2.1 Technology overview

The LLVM project is a collection of compiler and toolchain technologies that can be used to build compiler-frontends for programming languages and compiler-backends for multiple instruction set architectures [4]. One project created alongside LLVM is Clang. Clang is a compiler-frontend for languages in the C language family including C, C++, Objective C and many others [5].
The LibTooling library was created as part of the Clang project and it allows developers to build standalone tools based on Clang [6]. The library provides access to Clang's AST parser and LLVM's command line parsing. As C++ is a complex programming language with context-sensitive grammar, having access to an existing parser through the library is a great starting point for tool development [7]. The tool developer APIs hook into the compilation pipeline after the parsing step, which guarantees that the tool is run on valid ASTs. Furthermore, Clangs AST is very fine-grained allowing for a detailed analysis of the provided source code, which is very helpful when developing tools [8]. The full compilation pipeline of Clang and LLVM can be seen on fig. 2.1. The structure of a tool created with LibTooling can be seen on fig. 2.2.

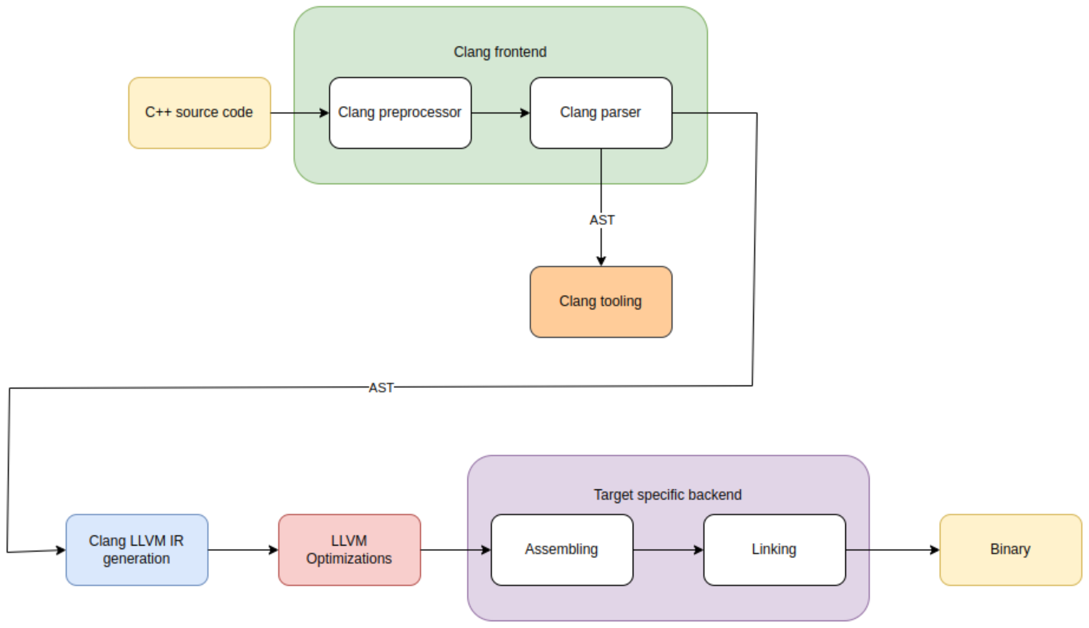

Figure 2.1: Overview of the Clang and LLVM compilation pipeline. The yellow blocks in the diagram are in/outputs of the pipeline. The green box is the Clang front end. The blue box is the conversion step between the Clang front end and the LLVM back end. The red box is the LLVM optimizations. The purple box represents the tasks of the backend for the specific instruction set architecture.

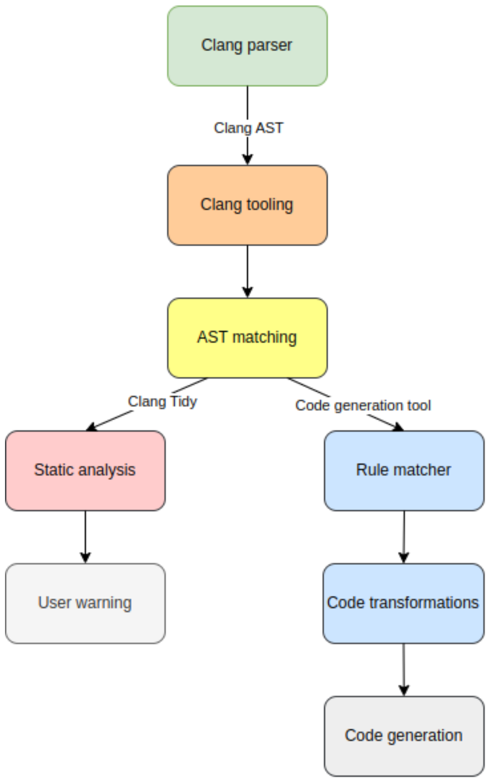

Figure 2.2: An overview of the Clang frontend and the LibTooling library. The green and orange boxes are identical to the boxes in fig. 2.1. The yellow box is the filtering/semantic analysis of the AST for the given tool. The tree splits into two to show that multiple different tools can use the library. On the left with the red box is the structure of the popular Clang-tidy tool[9]. On the right in blue is the structure of a custom tool as developed in this project. The two grey boxes indicate outputs from the tools.

2.2 Project delimitations

While LibTooling supports writing tools for programming languages in the C language family, this project is delimited to focusing on tools written for C++.

In many cases, it may be beneficial to integrate the tools developed with LibTooling into the development flow by providing it as a Clang plugin. While the project will not delve into this topic, it is worth mentioning that the developed tools have the potential to be exported as Clang plugins [10].

At last, LibTooling offers multiple APIs for developing standalone tools. Although the APIs may have different appearances, they essentially provide similar functionality. They can all be utilized to develop tools that leverage the Clang AST but adopt different software patterns in doing so. A few examples of the APIs are RecursiveASTVisitor, LibASTMatchers and Clang Transformer [11, 12]. This project will focus on the Clang Transformer API and, as a result, will not go into detail about the alternative APIs in LibTooling [13].

3 Methods

As described earlier three different tools were developed during the project. The tools are used as a progressive learning platform for exploring increasingly complex parts of the LibTooling library.

The first tool is a renaming tool, that will refactor a method with an illegal name ("MkX") into a legal name ("MakeX"). The tool will also rename all the calls to that method in order to keep the exact same functionality. The purpose of developing this tool is to get familiar with the basics of the LibTooling library, which will make later development easier.

The second tool is also a refactoring tool but with more complexity than the renaming tool. The purpose of the second tool is to convert traditional C-style arrays into the more modern and strongly typed `std::array`s. This tool is more complex than the renaming tool because there is more information associated with arrays than function names. Furthermore, there are more semantic considerations which have to be taken into account when making this type of change.

The third tool will analyze the code base for enums and generate "to_string" methods for each enum definition. The "to_string" method will return a string representation of the named enum constants defined in the enum. In order to ensure valid source code after the tool is run, existing "to_string" methods should be overwritten or updated as a part of the process. This tool is comparable in complexity to the C-style converter tool but adds the complexity of code generation to the tool.

Through these three tools, a deeper understanding of the LibTooling library and the C++ language is obtained.

4 Development

4.1 Installing LLVM and Clang

One might think installing LLVM and Clang should be a straightforward process but in reality, it can be quite complex. Therefore, this section means to describe the process that was used during the R&D project. The section is heavily inspired by [14] but more specialized to account for the concrete project.

The process of compiling LLVM, Clang and LibTooling can be considered a two-step process. Initially, the tools must be compiled using an arbitrary C++ compiler and then recompiled using the Clang compiler itself.

Before compiling the projects, one must install the needed tools, which include "CMake" and "Ninja". They can be installed using a package manager or by compiling it locally through `https://github.com/martine/ninja.git` and `git://cmake.org/stage/cmake.git`. Furthermore, one needs to have a working C++ compiler installed. The rest of this section assumes the compiler GCC is installed.

LLVM and Clang can then be compiled for the first time using GCC. Assuming the terminal is used, this can be done as seen in listing 4.1. First, the LLVM repository is cloned which also contains the Clang project. Line 2 - 4 goes inside the repository and sets up the build folder. Line 5 uses CMake to configure the project where Ninja is used as the generator[1], Clang and Clang Tools are enabled, tests are enabled and it should be built in release mode. The final line instructs Ninja to build the projects. Note that it is possible to skip building and running the tests but it is recommended to ensure that the build process succeeded.

```bash
1 git clone https://github.com/llvm/llvm-project.git
2 cd llvm-project
3 mkdir build
4 cd build
5 cmake -G Ninja ../llvm -DLLVM_ENABLE_PROJECTS="clang" -DLLVM_BUILD_TESTS=ON -DCMAKE_BUILD_TYPE=Release
6 ninja
```

Listing 4.1: Bash commands to initially compile LLVM and Clang.

The next steps consist of testing the targets to ensure that the compilation was successfull. This is done by running the tests as seen in the two first lines on listing 4.2. Finally, the initial version of Clang that is compiled with an arbitrary compiler is installed.

```bash
1 ninja check
2 ninja clang-test
3 sudo ninja install
```

Listing 4.2: Bash commands to test the LLVM and Clang projects and then finally install them.

Clang should now be recompiled using Clang to avoid name mangling issues [16], i.e., ensure that the symbolic names the linker assigns to library functions do not overlap. This time the project "cmake-tools-extra" should also be included to build LibTooling and the complementary example projects. The option `-DCMAKE_BUILD_TYPE=RelWithDebInfo` is also a possibility if one wishes to include debug symbols in the libraries which can be useful during development. This however comes with a performance trade-off,

[1]A CMake generator writes input files to the underlying build system.[15]

as Clang itself will also be compiled with debug symbols which will slow it down. The group has yet to find a way to compile LibTooling with debug symbols but Clang without it.

```
cmake -G Ninja ../llvm -DLLVM_ENABLE_PROJECTS="clang;clang-tools-extra" -DCMAKE_BUILD_TYPE=Release
↪  -DCMAKE_CXX_COMPILER=clang++
ninja
```

Listing 4.3: Bash commands to compile LLVM, LibTooling and Clang with Clang as compiler.

Finally, the steps from listing 4.2 should be repeated to verify the recompilation and install the tools.

4.2 Build environment

The documentation for writing applications using LibTooling such as [17, 13] mainly concerns writing tools as part of the LLVM project repository. While this is good for contributing to the project, it is not ideal for version control and developing stand-alone projects. It was necessary to create a build environment that allowed for out-of-tree builds which utilize LibTooling. A similar attempt was made in [18] but the project was abandoned in 2020 and LLVM has since moved from a distributed repository architecture to a monolithic one making most of [18] obsolete. The following section is dedicated to describing the important decisions made related to the build environment.

4.2.1 Build settings

Initially, some general settings for the project are configured which can be seen in listing 4.4. Line 1 forces Clang as the compiler which is highly recommended as LibTooling was compiled with Clang. Choosing another compiler may result in parts of the project being compiled with another standard library implementation, e.g., libstd++ that is the default for GCC. This may cause incompatibility between the application binary interfaces (ABIs) which is considered undefined behaviour, essentially leaving the entire program behaviour unspecified [19]. This concept is also known as ABI breakage. Line 2 defines the C++ standard version, which is set to C++17 since LibTooling was compiled with this. Line 3 defines the output directory of the executable to be in `<build_folder>/bin` which has importance concerning how LibTooling searches for include directories at run-time as described in section 4.2.2. Finally, line 4 disables Run-Time Type Information (RTTI). RTTI allows the program to identify the type of an object at runtime by enabling methods such as `dynamic_cast` and `typeid` among others. When compiling LLVM it is up to the user whether RTTI should be included or not. RTTI is disabled by default when compiling LLVM as it slows down the resulting executable considerably. This flag is propagated to the subprojects that were compiled with LLVM such as LibTooling. By default a project in CMake is compiled with RTTI and CMake will assume that the used libraries were compiled with the same flags. This will result in nasty linker errors and the RTTI should therefore be explicitly disabled in the tool project.

```
1 set(CMAKE_CXX_COMPILER clang++)
2 set(CMAKE_CXX_STANDARD 17)
3 set(CMAKE_RUNTIME_OUTPUT_DIRECTORY "${CMAKE_BINARY_DIR}/bin")
4 add_compile_options(-fno-rtti)
```

Listing 4.4: General settings for the CMake build environment.

4.2.2 Run-time include directories

When executing binaries created with LibTooling, a big part of the process is the analysis of the target source code. The analysis is done following the pipeline as shown in fig. 2.1. Most projects written in C++ make use of the C++ standard library that implements many commonly used functionalities in C++. Naturally, the tool needs to know the definitions for the standard library in order to analyse the target source code. For practical reasons, LibTooling provides a mechanism for the automatic discovery of header files that should be included when parsing source files. It finds the headers by using a relative path with the pattern `../lib/clang/<std_version>/include` from the location of the binary. Where `<std_version>` indicates which version of the standard library which the tool was compiled with (in this project it was 17).

This hard-coded approach is quite simple but limited, as it forces the users to only run the tool in a directory where the headers can be found in the relative directory `<current_dir>/../lib/clang/17/include`. If the user attempts to run it somewhere else, and the analyzed files make use of standard library features, they will get an error while parsing the files (e.g. that the header `<stddef.h>` was not found). This issue makes it more difficult to write truly independent tools as they still need some reference to the Clang headers, which would essentially mean moving the executable to the directory where Clang was compiled.

One existing solution is to provide the location of the headers as an argument to the binary when executed. This is possible since tools written with LibTooling invoke the parser of Clang, from where it is possible to forward the include directory as an argument to the compiler e.g. by specifying `-- -I"/usr/local/lib/clang/17"`. However, this was found to be impractical since the location of the include path may vary depending on the system and forgetting to write the path results in errors that can be very difficult to decipher.

Instead, it was decided to create a build environment where the user must provide the location of the Clang headers when configuring CMake or an appropriate error message is generated. Through CMake, the necessary headers are then copied to the build directory.

The solution is by no means perfect, as the user is still forced to execute the binary from the build directory. In many situations, this is sufficient, as most IDEs follow this behaviour as default and it allows the projects to be built out-of-tree. If the user wishes to run the binary from outside the build directory, they still have the option of specifying the location through the `-- -I"<clang_include>"` option. The solution can be found in the functions.cmake file.

In the future, it may be desirable to explore a solution using the LLVM command line library to search some commonly used directories for the Clang headers.

4.2.3 Configuring the target project

The target project on which source files the tool will be run should generate a compilation database. A compilation database contains information about the compilation commands invoked to build each source file. This file is used by LibTooling to detect the compile commands and include files necessary to generate the correct AST for the compilation unit.[20] Tools like CMake can auto-generate the compilation commands at configuration time. When using CMake the compilation database generation can be enabled by inserting `set(CMAKE_EXPORT_COMPILE_COMMANDS 1)` in the CMakeLists.txt file.

4.3 Tool structure

Through inspection of existing LibTooling tools, such as [21] and [22], and the development of tools for the R&D project, a common structure for LibTooling tools has been identified. The structure can be used as a way of categorizing the different parts of such tools and can be seen below.

- Command line parsing
- AST node matching
- Node data processing
- Handling the results

The **Command line parsing** part of the tool structure is responsible for the parsing of the command line options which the tool was invoked with and setting the configuration data that specifies the behaviour of the tool. A detailed description of this section of the tool structure can be found in section 4.3.1.

The **AST node matching** part of the tool structure is where the AST node matchers are defined and certain nodes of interest are bound to identifiers. AST node matchers traverse the AST of the source code passed to the tool through the command line and binds relevant nodes to identifiers which can be used in the Node data processing section. For a detailed description of the AST matching step, see section 4.3.2.

The **Node data processing** part of the tool structure is where information is extracted from the nodes that were bound during the AST node matching step. This is where the primary functionality of the tool is implemented. There are multiple interfaces which can be used to extract information from bound nodes and they are described in section 4.3.3.

The **Handling the results** part of the tool structure is where the matching and processing, defined in the previous parts, is applied. This is also where the results are handled and presented to the invoker of the tool. This part of the tool structure is described further in section 4.3.4.

4.3.1 Command line parsing

A good way of configuring the behaviour of a tool is through command line arguments. All LibTooling tools come with some common options which are always present, e.g., the option to parse a list of source files.

The way to add command line options to a tool is through the command line library provided by LLVM. This library makes it very easy to add new commands and provide help text for the options. A great example of how to use the library can be found in the implementation of ClangFormat [23], where they have multiple options with different types and default values.

4.3.2 AST node matching

The next step in working with LibTooling is to find the AST that is relevant for the tool through matching and binding. This is a very important part of the tool development as it defines what kind of information is available for the later stages of the tool. If the wrong AST is matched or the wrong nodes are bound, the later processing can become unnecessarily complex and complicated. Furthermore, the task of matching the AST can also be very difficult as there are currently more than 700 different matchers available as part of the library [24].

A proposed way of matching the AST is by writing or finding a very simple example of the code that should be matched upon. The entire AST for that file can be printed through Clang by invoking: `clang -cc1 -ast-dump input_file`. This command will print all the AST information in the input file to the console, which can then be analyzed manually for the wanted nodes/patterns.

When an AST node has been identified as interesting to the tool it should be matched, which is typically done by using the predefined matchers of LibTooling [24]. It is also possible to write custom matchers, which can be very powerful when the standard matchers are lacking. An example of such could be if one wishes to recursively traverse the declaration context, which was the case in one of the implementations of the enum tool. However, most the time the builtin matchers will be sufficient and they should therefore be the initial place to look when matching the AST.

LibTooling defines three different basic categories of "Matchers" available to the users: Node Matchers, Narrowing Matchers and Traversal Matchers. The Node Matchers are the most general matchers and matches specific types of AST Nodes, e.g., `enumDecl` and `constantArrayType`. A subset of the Node Matchers are also bindable, meaning that they can be bound to an identifier and processed in the later stages of the tool. The Narrowing Matchers can be used to filter nodes that fulfil certain requirements, e.g. `hasName` and `isClass`. It also contains logical expressions such as `allOf`, `anyOf` and `unless`. The Traversal Matchers can be used to traverse the node with its parents or children and thereby specify and bind to certain relations of subnodes. Examples of Traversal Matchers are `hasDescendant`, `specifiesType` [24].

When composing a matcher, i.e., combining several matchers, one typically starts by identifying the overall node type that wishes to be matched. This will be the outermost Node Matcher and in most cases, this should be bound. A combination of narrowing matchers and traversal matchers can then be used to filter the matches depending on the information the tool needs. In many scenarios, it may also be necessary to bind child nodes in order to get all the needed information.

Putting it all together, one could write a matcher that matches function declarations that are named "f" and that takes at least one parameter. Furthermore, the function declaration and parameter declaration could be bound to the identifiers "function" and "parameter". This can be achieved using a combination of the three categories of matchers with the statement:
`functionDecl(hasName("f"), hasParameter(0, parmVarDecl().bind("parmBind"))).bind("funcBind")`
In the example the matchers `functionDecl` and `parmVarDecl` are Node Matchers, `hasName` is a Narrowing Matcher and `hasParameter` [2] is a Traversal Matcher.

The composition of matchers is typically done iteratively where one starts by matching a superset of nodes which are iteratively narrowed down until the tool is left with the nodes that contain the needed information.

4.3.3 Node data processing

When the nodes have been matched and bound, the information stored inside of them must be extracted. The way to do so is by using an object that inherits from `MatchFinder::MatchCallback`. The `MatchFinder::MatchCallback` class defines the `run` method which must be overridden by its children. The run method has a `const MatchFinder::MatchResult &Result` parameter which contains the bound nodes of the match (e.g. "funcBind" and "parmBind" in the example provided earlier). The information inside the nodes can then be extracted and used e.g., for source code generation, diagnostic messages or other information relevant to the tool.

The `MatchFinder::MatchCallback` is the raw interface which allows for node processing but it lacks many convenience methods and requires the user to save the extracted information explicitly. The Clang developers have created an abstraction over the `MatchFinder::MatchCallback` with convenience methods and an automatic way of extracting information. This interface also allows for easy conversion to source code changes. The abstraction is called `Transformer` and is what will be used in this R&D project.

Transformers combine a rewriting rule with a result consumer. Rewriting rules combine a matcher with an AST edit and potential metadata. The AST edit is a change in the source code comprised of a source code location to rewrite and a concatenation of multiple Stencils which generate the new source text.

[2]The 0 provided in `hasParameter` indicates that it must match the first argument of the function.

Stencils extract information from bound nodes and convert the information to strings. A more detailed description of Stencils can be seen in section 4.3.3.

The result consumer is responsible for saving the relevant results so they can be processed by the `ClangTool` later. The result consumer is further described in section 4.3.3.

Stencils

The stencil interface is used to extract information from bound nodes and convert the information to strings. The stencil interface is an abstraction on top of `MatchComputation<std::string>` which is called on matched nodes through the Transformer API.

Examples of use-cases for the stencil interface could be to extract the element type of an array or to issue a warning at a given location. The functions of the predefined stencils are primarily focused on control flow, concatenation of stencils and expression handling. Therefore it is very likely that the creator of a tool will have to create custom stencils to extract the necessary data from the bound nodes.

Luckily the stencil interface allows the simple conversion from `MatchConsumer<std::string>` to a `Stencil` through the `run` method. The `MatchConsumer<T>` type is a typename for `std::function<Expected<T>>(const ast_matchers::MatchFinder::MatchResult &)` with `std::string` as the template parameter. This API allows the creator of a tool to write small methods that extract the necessary information from a bound node as a string and seamlessly concatenate them together through the `cat` stencil.

All the predefined stencils return strings but it is possible to create an similar library that returns any type of data if that is required for the tool. The reason the predefined stencils work solely with strings is that it is primarily used to generate source code changes which must be converted to strings of some sort in order to be written to disk.

Similarly to the stencil interface, LibTooling also defines the range selector interface. This interface also builds upon `MatchConsumer<T>` but with `CharSourceRange` as an output instead of strings. A `CharSourceRange` refers to a range of characters defined at a specific location of the provided source files. It thereby allows the tool implementor to add or modify source code exactly where they want it.

Combining matchers and stencils

It should be clear by now that in order to create tools, it is necessary to have both matchers and stencils. Within this lies some interesting design decisions of how the matchers and stencils should be used in conjunction.

In general, there are two approaches to take when using matchers and stencils in conjunction.
The first approach is to create a simple matcher that binds only to the outermost node and then create detailed stencils that extract the information based on the single binding.
The second approach is to create a detailed matcher that binds to multiple nodes and then create simple stencils that utilize the many bindings.
Both approaches can be used to implement the same functionality but the implementations look vastly different.

If the implementor chooses the first approach, then the stencils can easily become complicated because one must filter and extract information from a single node.

If the second approach is chosen, the responsibility of extracting information from the nodes is placed inside the composed matcher. This can lead to some very complex matchers that can be difficult to understand. However, the stencils that extract information from the bound nodes will be much simpler.

Furthermore, when following this approach one can easily fall into the pitfall of trying to match too much data that is not required at the end.
E.g., when developing the enum tool it was attempted to bind the namespace of the parameter of an existing `to_string` function since it was thought to be needed when writing the transformation. In the end, the binding was unnecessary as the namespace could easily be extracted through a stencil and the namespace binding itself did not provide enough context. However, a significant amount of time was spent on writing an exact matcher that could recursively traverse the namespace qualifiers of a parameter and bind it.

Ultimately, the best approach to follow depends upon the specific scenario. In some cases, it may be better to write detailed stencils and in other cases detailed matchers. The important thing is to not tunnel-vision too heavily on a single approach and keep an open mind towards the other. Perhaps the best approach lies within a mixture of the two.

Consuming the transformation changes

When the bound nodes have been processed through the Transformer API, the transformation changes should be consumed. This is done through a `Consumer` which is a type alias of a `std::function` which takes `Expected<TransformerResult<T>>` as a parameter. The `TranformerResult<T>` type contains any source code changes that were generated by the rule and the provided metadata with type T.

The consumer can make decisions based on the received edits and metadata, but the most common use case for the consumer is to have it store the metadata and changes to external variables so it can be later used for further processing.

4.3.4 Handling the results

When the rules and transformers have been specified it is time to run them on the source code, which is done through a `ClangTool`. `ClangTool` is the API that runs the match finders over all the specified source code. All tools made with clang need to use `ClangTool` to tie it all together.
The `ClangTool` class has a `run` method which takes a `FrontendAction` and runs it on the specified source code. This can be considered the method that executes the tool.

The `ClangTool` class can be extended to handle the results from the Transformer in different ways. A tool which runs a Transformer could, for example, save the source code changes to disk or present the changes to the caller of the tool and have them choose if the changes should be made. The extended `ClangTool` also often contains the variables which will be updated in the node processing step.

5 Tool examples

This section contains examples of implementations of Clang tools. The purpose of the examples is to show how the theory described in section 4.3 can be used. All the tools developed during this project will save the results to disk, but as described in section 4.3 other approaches could have been chosen. The code for the examples can be found in the git repository for this project.

Each example will be split into the four sections of a Clang tool as described in section 4.3.

5.1 Simple rename refactoring tool

The goal of this tool is to rename all functions in the provided source code that has the name “MkX” into “MakeX”. The tool should both rename the function declaration and the locations where it is called. The code in this section has been mostly stripped of the namespace specifiers in order to simplify the code.

5.1.1 Command line parsing

In order to make this tool as simple as possible, the name of the method to rename and the new name have been fixed in the code. Therefore the Command line parsing element of the tool will use only the general options available for all LibTooling tools. The common command line options can be used by making a `CommonOptionsParser`. The way to create such an object can be seen on listing 5.1.

```
int main(int argc, const char* argv[]) {
   auto ExpectedParser = CommonOptionsParser::create(argc, argv, llvm::cl::getGeneralCategory());
   if (!ExpectedParser) {
      // Fail gracefully for unsupported options.
      llvm::errs() << ExpectedParser.takeError();
      return 1;
   }
   CommonOptionsParser &OptionsParser = ExpectedParser.get();

    return 0;
}
```

Listing 5.1: Example code which shows the creation of the `CommonOptionsParser` used for all ClangTools.

5.1.2 AST node matching

For this tool to work two different types of nodes need to be matched. First, the function declaration with the name “MkX” has to be matched, and then all expressions which call the method have to be matched. This can be achieved through the two matchers shown in listing 5.2 and listing 5.3.

```
auto functionNameMatcher = functionDecl(hasName("MkX")).bind("fun");
```

Listing 5.2: This example shows a matcher that will match on any function declaration which has the name “MkX”.

```
auto invocations = declRefExpr(to(functionDecl(hasName("MkX"))));
```

Listing 5.3: This example shows a matcher that will match on any expression which calls to a function declaration with the name “MkX”.

5.1.3 Node data processing

In this tool, the act of processing the nodes is simple, as the tool just has to rename the method and all the locations where it is called. This is a native part of the rules API as described earlier (section 4.3.3).

The two renaming rules can be seen on listing 5.4 and listing 5.5.

```
auto renameFunctionRule = makeRule(functionNameMatcher, changeTo(name("fun"), cat("MakeX")));
```

Listing 5.4: The rename function rule used in the example. The rule consists of the functionNameMatcher as specified in listing 5.2 and the renaming action. In this case, the action is to change the name of the bound method to “MakeX”.

```
auto renameInvocationsRule = makeRule(invocations, changeTo(cat("MakeX")))
```

Listing 5.5: The rename invocations rule which updates the invocations to the renamed method. Here the entire expresion is changed to the new method name.

The two rules specified here are closely coupled as running just one of the rules would result in invalid source code. There is a way to group rules into a single rule and it is called `applyFirst`. This method creates a set of rules and applies the first rule that matches a given node. That means that there is an ordering to `applyFirst`. This ordering can be ignored for independent rules, like the two specified in this section, and in that case, it will simply create a disjunction between the rules. The combined rule can be seen on listing 5.6.

```
auto renameFunctionAndInvocations = applyFirst({renameFunctionRule, renameInvocationsRule});
```

Listing 5.6: A rule that both renames the function declaration and the invocations of that function.

The rules required for the simple renaming tool have been specified, but in order to extract the source code changes specified by the rules they have to be coupled with a transformer. The transformer is described in detail in section 4.3.

The transformer needs a consumer that saves the generated source code edits to an external variable. The consumer callback receives an expected array of `AtomicChange` objects which in turn contain the `Replacements` in the actual source code. The consumer shown in listing 5.7 extracts the `Replacements` from the `AtomicChange`s and saves them in a map variable that is defined externally.

```cpp
auto consumer(std::map<std::string, Replacements> fileReplacements) {
    return [=](Expected<TransformerResult<void>> Result) {
        if (not Result) {
            throw "Error generating changes: " + toString(Result.takeError());
        }
        for (const AtomicChange &change : Result.get().Changes) {
            std::string &filePath = change.getFilePath();
            for (const Replacement &replacement : change.getReplacements()) {
                Error err = fileReplacements[filePath].add(replacement);

                if (err) {
                    throw "Failed to apply changes in " + filePath + "! " + toString(std::move(err));
                }
            }
        }
    };
}
```

Listing 5.7: A transformer consumer that saves all the generated source code edits to an external map by filename.

The consumer and the rules can be used to create the transformer as shown in listing 5.8.

```cpp
Transformer transformer(renameFunctionAndInvocations, consumer(externalFilesToReplaceMap));
```

Listing 5.8: A rule that both renames the function declaration and the invocations of that function. The externalFilesToReplaceMap variable passed to the consumer will be discussed later.

5.1.4 Handling the results

This part of the tool is responsible for the creation of the actual tool and saving the results to disk.

The goal of this tool is a form of refactoring and Clang already has a tool for refactoring called clang-refactor. This tool is also created through LibTooling and it defines a class called `RefactoringTool` which extends the `ClangTool` class. The `RefactoringTool` adds a way to save `Replacements` to disk. The changes that should be saved to disk are located in a `std::map<std::string, Replacements>` map which is contained inside of the `RefactoringTool`. The `RefactoringTool` implementation already contains all the needed functionality to finish the rename refactoring tool. All that remains is therefore to create the tool and invoke it, which is shown in listing 5.9.

```cpp
int main(int argc, const char* argv[]) {
// CL parsing
...

RefactoringTool Tool(OptionsParser.getCompilations(),
                     OptionsParser.getSourcePathList());
auto &externalFilesToReplaceMap = Tool.getReplacements();

// transformer creation
...

//Register the transformation matchers to the match finder
MatchFinder Finder;
transformer.registerMatchers(&finder);

//Run the tool and save the result to disk.
return Tool.runAndSave(newFrontendActionFactory(&finder).get());
} // end main
```

Listing 5.9: This code snippet shows the creation of a `RefactoringTool` called 'Tool'. The construction of the tool requires the source code that was passed through the command line. The internal map in the Tool is used as input to the transformer, as seen in listing 5.8.

As can be seen in listing 5.9 the tool combines the results from the other parts of the tool structure into the final tool. The tool is then invocated by the `runAndSave` method call, which invokes the tool and saves the results to disk afterwards.

5.2 C-style array converter

The traditional way of making arrays in C is by using the subscript operator. A fixed-size C array consists of a type, a name and a constant size. The line `int my_array[10]` will create a variable called "my_array" which is a collection of 10 consecutive integers. In C++, this style of declaring an array is called a C-style array.
In C++11 the container `std::array` was added to the standard library, which acts as a wrapper around the C-style array. It combines the performance of a C-style array while having the benefits of a standard container, such as knowing the size of the container and providing access to iterators [25]. In C++ both ways of declaring arrays are allowed, however, `std::array` is generally preferred, as it is considered safer since it makes bounds-checking easier. Bounds safety is of great concern in C++ as the language does not provide built-in mechanisms to prevent out-of-bounds errors. These errors are considered a serious issue, and out-of-bounds writes were ranked as the most dangerous software weakness in 2022 on the Common Weakness Enumeration list[1] [26].

The goal of this tool is to find all the constantly sized C-style arrays in the source code and convert them into `std::array` s.

One of the test cases for this tool is to convert `static const int* const const_pointer_const_array_static[2]` into `static std::array<const int* const>, 2> const_pointer_const_array_static` . In order to achieve this, the storage (static) class and const-qualifiers of the type must be preserved through the transformation, which is a bigger challenge compared to the simple rename tool section 5.1.

[1]The list ranks software weaknesses based on their commonness and impactfulness.

5.2.1 Command line parsing

Like the renaming tool section 5.1 the customization of the command line arguments for this tool has been left out in order to cut down on complexity.

5.2.2 AST node matching

This tool works on C-style arrays with a constant size so the AST matcher for that type of node must be identified.

A C-style array is a type in the Clang AST. The library contains multiple matchers which match different variants of C-style arrays. The types of C-style arrays are: `Array` , `Constant` , `DependentSized` , `Incomplete` and `Variable` .
The `Array` type is a base type for all the other types of C-style arrays. The `Constant` array type is a C-style array with a constant size. The `DependentSized` array is an array with a value-dependent size. The `Incomplete` array is a C-style array with an unspecified size. The `Variable` array type is a C-style array with a specified size that is not an integer constant expression.

Each of the types has a corresponding matcher which allows the creator of a tool to match only the wanted types of C-style arrays. The focus of this tool is solely on `Constant` arrays, as they are directly convertible to `std::array` s. The same would probably also be true for the `DependentSized` array type, but this has been left out in order to simplify the tool.

The constant array type node contains the element type of the array as well as the number of elements in the array. The constant array type node does not contain the storage specifier for the type (e.g., static or extern) so that information must be bound in another way. The storage specifier is stored in the declaration of the array which is a `VarDecl` . For this tool, the bound declaration will also make it easy to access the namespace-qualifiers for the C-style array.

There is a problem however as this tool also aims to refactor the raw C-style arrays declared inside of classes. This distinction matters for the declaration of the variables, as variables inside classes are `FieldDecl` s and not `VarDecl` s. For this matcher, the type of node to match must therefore be more generic than the `VarDecl` . This can be achieved by using the `DeclaratorDecl` matcher which matches on `DeclaratorDecl` s, which is the common base class between `VarDecl` s and `FieldDecl` s.

The variable declaration for the arrays also contains the bounded location of the declaration in the source code. This can be easily extracted through the `typeLoc` matcher. This information will make it easier to change the correct location in the source code in the following steps and is therefore present in the matcher. The complete matcher for finding constant arrays can be seen in listing 5.10.

```
auto ConstArrayFinder =
    declaratorDecl(
        isExpansionInMainFile(),
        hasType(constantArrayType().bind("array")),
        hasTypeLoc(typeLoc().bind("arrayLoc")))
    .bind("arrayDecl");
```

Listing 5.10: C-style array matcher with bindings.

5.2.3 Node data processing

In this section, the relevant data from the bound nodes will be extracted and used to generate source-code changes. The Stencil and Transformer libraries will be used for the data extraction and code refactoring. As described in the previous section, there are multiple pieces of information which need to be extracted from the nodes.

From listing 5.10 three node types were bound. “array” is the type of constant array that was found, “arrayLoc” is the location of the constant array in the source code and “arrayDecl” is the entire declaration of the array.

RangeSelector

The first thing needed in order to make the change is to extract the source code range from the nodes. The “arrayLoc” node, which has been bound, is a `TypeLoc` which spans the `[number_of_elements]` part of the C-style array. This may seem a bit confusing at first, but the behaviour is due to the `typeLoc` referring to the underlying `ArrayTypeLoc`, i.e., the part of the source code that makes it a C-style array. “arrayLoc” can be used in conjunction with the “arrayDecl” node in order to get the entire source code range for the array declaration. The `RangeSelector` API has a convenience function `encloseNodes` which is meant for exactly this purpose.

StorageClass

Now that the correct source code range has been found, it is time to populate it with the correct information. As specified earlier the storage specifier for the declaration must be kept and it can be found through the `getStorageClass` function on VarDecl nodes. This function is not present in FieldDecl nodes because they don’t have storage specifiers. `static` FieldDecls inside classes are converted to VarDecls during the AST generation, as these variables are independent of the individual class instances.

The `getStorageClass` function can be implemented as a `MatchConsumer<std::string>` which will make it easy to use the function to make source-code refactoring (see section 4.3). An implementation of such a function can be seen on listing 5.11. Notice that if the node is a FieldDecl, the function will return an empty string.

```cpp
auto getVarStorage(StringRef Id) {
    return [=](const MatchFinder::MatchResult &Match) -> Expected<std::string> {
        if (auto var = Match.Nodes.getNodeAs<VarDecl>(Id)) {
            auto storage_class = var->getStorageClass();
            if (storage_class != StorageClass::SC_None) {
               return std::string(VarDecl::getStorageClassSpecifierString(storage_class)) + " ";
            }
        }
        return "";
    };
}
```

Listing 5.11: Method to extract the storage specifier string from a VarDecl node bound to Id.

Array element type

The array element type is stored inside the `ConstantArrayType` node. It can be accessed through the `getElementType` method call. Like the storage class specifier in listing 5.11 the easiest way to work with the element type for this tool is through the Stencil library. The getArrayElementType method will therefore be implemented as a MatchConsumer. The implementation of the function can be seen in listing 5.12.

```
auto getArrayElementType(StringRef Id) {
    return [=](const MatchFinder::MatchResult &Match) -> Expected<std::string> {
        if (auto array = Match.Nodes.getNodeAs<ArrayType>(Id)) {
            return array->getElementType().getAsString();
        }
        throw std::runtime_error("ID not bound or not ArrayType: " + Id.str());
    };
}
```

Listing 5.12: Method to extract the element type from the ConstantArrayType node.

Array size

The size of the array can also, like the array element type, be extracted through the ConstantArrayType node. The implementation of the extraction method can be seen in listing 5.13.

```
auto getConstArraySize(StringRef Id) {
    return [=](const MatchFinder::MatchResult &Match) -> Expected<std::string> {
        if (auto array = Match.Nodes.getNodeAs<ConstantArrayType>(Id)) {
            auto size = array->getSize().getZExtValue();
            std::stringstream ss;
            ss << size;
            return ss.str();
        }
        throw std::runtime_error("ID not bound or not ConstantArrayType: " + Id.str());
    };
}
```

Listing 5.13: Method to extract the element size from the ConstantArrayType node. The array size is a llvm::APInt and must be converted to a `uint64_t` through the `getZExtValue` method.

Declaration namespace-qualifiers

The last step is to keep the namespace-qualifiers of the declaration. The qualifiers of interest are the explicit namespaces in front of the name of the declaration, including the scope resolution operator. E.g., `uint8_t my_namespace::array[5];` has the qualifiers `my_namespace::`, while `uint8_t array[4];` has no qualifiers. These qualifiers are part of the declaration and the easiest way to extract them is to copy the literal text in the source code. The literal source code of declarations can be extracted through the source range of the node. Listing 5.14 shows how the qualifiers of a `DeclaratorDecl` can be extracted from the source-code.

```
auto getDeclQualifier(StringRef Id) {
    return [=](const MatchFinder::MatchResult &Match) -> Expected<std::string> {
        if (auto decl = Match.Nodes.getNodeAs<DeclaratorDecl>(Id)) {
            CharSourceRange qualifierRange = getTokenRange(decl->getQualifierLoc().getSourceRange());
            return getText(qualifierRange, *Match.Context).str();
        }
        throw std::runtime_error("ID not bound or not DeclaratorDecl: " + Id.str());
    };
}
```

Listing 5.14: Method to extract the namespace-qualifiers from the source code range of a DeclaratorDecl node.

Creating the transformation rule

With all the relevant information extracted from the bound nodes, a `RewriteRule` can be made to create the source-code refactoring. The code for the rule can be found on listing 5.15.

```
auto C-styleArrayConverterRule = makeRule(
    ConstArrayFinder,
    {
        addInclude("array", IncludeFormat::Angled),
        changeTo(
            encloseNodes("arrayDecl", "arrayLoc"),
            cat(run(getVarStorage("arrayDecl")),
                "std::array<",
                run(getArrayElementType("array")),
                ", ",
                run(getConstArraySize("array")),
                "> ",
                run(getDeclQualifier("arrayDecl")),
                name("arrayDecl")
            )
        )
    });
```

Listing 5.15: The entire rule for generating the `std::array` declaration. The rule both adds the `<array>` header and makes the source-code refactoring in one step.

As it can be seen from listing 5.15, the refactoring rule is relatively simple to read, as it solely uses the methods created through this section and strings of text, which is very descriptive. This makes each line easy to reason about and transparent. The difference between the node processing step for the C-styleConverter tool as compared to the RenamingTool, is that more information has to be extracted from the bound nodes. This requires a deeper understanding of which node types contain the wanted information, and how the information is encoded in the nodes.

Consuming the changes

In the renaming tool (section 5.1) the `Tranformer` consumer extracted the `Replacements` from the `AtomicChanges` generated by the rule. This was necessary, as the tool used the `RefactoringTool` class provided by the library. In the renaming tool, the rule only had a single change for the matches, which was quite easy to extract from the `AtomicChanges`. In this tool, the rule adds multiple changes and it even includes the `array` header. This added complexity to the extraction of the replacements in the consumer.

During the development of this tool it was discovered, that there is a convenience method in the library that allows the developer to apply `AtomicChanges` directly to the source code instead of unpacking it manually. This method will take care of all the header removals and includes them in each `AtomicChange`. The group wanted to use this feature in the tool to reduce the complexity of the consumer, so a custom `ClangTool` class had to be created.

The custom class has to work with `AtomicChanges` instead of `Replacements` but the overall structure of the `runAndSave` method, which saves the changes to disk, is the same. As opposed to `Replacement` objects, `AtomicChange` objects already contain the file name for the change so all the changes can be inserted into a single `std::vector` and handled by the tool instead of the consumer. This change in semantics is what makes the consumer method simpler.

The new consumer method takes a `AtomicChanges` reference as a parameter and returns a lambda method which is a `ChangeSetConsumer`. The lambda inserts valid `AtomicChange` objects into the provided `AtomicChanges` array for later processing. The implementation can be seen on listing 5.16.

```cpp
auto RefactorConsumer(AtomicChanges &Changes) {
        return [=](Expected<TransformerResult<void>> C) {
            if (!C) {
                throw std::runtime_error("Invalid Changes! " + toString(C.takeError()));
            }
            Changes.reserve(Changes.size() + C.get().Changes.size());
            std::move(C.get().Changes.begin(), C.get().Changes.end(), std::back_inserter(Changes));
        };
}
```

Listing 5.16: Implementation of the `AtomicChange` consumer.

5.2.4 Handling the results

In the previous section, a new consumer was made for the `Transformer`. This consumer uses an external instance of `AtomicChanges` to save the results of the tool. After all the `AtomicChange` objects have been gathered, the changes can be applied to the source code and saved to disk - like it was done in the renaming tool. This logic has to be custom-made to use `AtomicChange` instead of `Replacements`. This change will be discussed here.

The renaming tool calls the `runAndSave` method of the `RefactoringTool` class. This method first runs the specified `FrontEndAction` to get the result of the tool. Afterwards, it creates a `Rewriter` with sources provided to the tool. The `Rewriter` class is an interface between the tool and the rewrite buffers, which is a low-level interface to the filesystem of the operating system. This `Rewriter` will allow the tool to make source-code changes. With the created `Rewriter` object, the `RefactoringTool` calls the `applyAllReplacements` and `saveRewrittenFiles` methods. These methods will apply the `Replacements` to the source code and then save the rewritten source to disk. It is this behaviour that needs to be reimplemented for the `AtomicChanges`.

The creation of the `Rewriter` class is identical between the implementation in the library and the custom version. This is the case because the `Rewriter` is used solely to apply the changes and not to reason about the changes. The creation of the `Rewriter` instance has thus been excluded from this report. It can be found in the implementation of the C-styleArrayConverter on the project GitHub page.

The `applyAllReplacements` method is where the custom tool class gets interesting. This method has to be created by hand and it needs to call the `applyAtomicChanges` method on all the gathered `AtomicChanges` for each filename. The `applyAtomicChanges` method can apply changes from multiple `AtomicChange` objects at once, but they must be changes in the same source file. This has to be checked by the developer before the method is called. This check was performed in the consumer in the RenamingTool but has to be implemented in the tool here.

In order to apply all the gathered changes, the tool needs access to those changes. In the `RefactoringTool`, the `Replacements` container was a part of the tool instance. This part will be copied into the custom tool, so it contains an instance of the `AtomicChanges`.

The `AtomicChange` objects each contain the name of the file to which the changes should be applied. Because this information is present in the objects they can be grouped by their source files in a map. The grouping is shown in listing 5.17.

```
AtomicChanges Changes;
bool applyAllChanges(Rewriter &Rewrite) {
    std::unordered_map<std::string, AtomicChanges> FileChanges;
    for (const auto& Change : Changes) {
        FileChanges[Change.getFilePath()].push_back(Change);
    }
    ...
}
```

Listing 5.17: Code snippet that will group the collection of `AtomicChange` objects based on the files they should change. The use of the unordered map is for performance reasons and could also have been a `std::map`.

Now that the changes have been grouped, it is time to apply the changes. In order to apply the changes, the specific source code and the file handle have to be used. This information can be extracted from the `Rewriter` through its internal `SourceManager` and `FileManager` using the file path stored in the groups. In order to get access to the source file from a given path, first an `FileEntryRef` is extracted from the `FileManager`. Then the `SourceManager` can be used to create a `FileID`, which can be used to access the source code. The created `FileID` can also be used to specify the range which should be overwritten by the `Rewriter`. Putting all of this together will commit the found file changes to disk. The implementation of this method can be seen on listing 5.18.

```cpp
bool applyAllChanges(Rewriter &Rewrite) {
    // Group the Changes
    ...

    auto &sm = Rewrite.getSourceMgr();
    auto &fm = sm.getFileManager();

    for (const auto &[File, ChangesForFile] : FileChanges) {
        auto Entry = fm.getFileRef(File);
        if (!Entry) {
            llvm::errs() << Entry.takeError();
            return false;
        }

        auto id = sm.getOrCreateFileID(Entry.get(), SrcMgr::C_User);
        auto code = sm.getBufferData(id);
        auto new_code = applyAtomicChanges(File, code, ChangesForFile, ApplyChangesSpec());

        if (!new_code) {
            llvm::errs() << new_code.takeError();
            return false;
        }

        Rewrite.ReplaceText(
            SourceRange(sm.getLocForStartOfFile(id), sm.getLocForEndOfFile(id)), new_code.get());
    }
    return Rewrite.overwriteChangedFiles();
}
```

Listing 5.18: Implementation of the `applyAllChanges` method which will apply all the generated `AtomicChanges` to the source code and save it to disk.

5.3 C-style array parameter converter

So far the tool was able to rewrite variable declarations of C-style arrays into their corresponding `std::array` implementations. However, if the variable that was transformed is passed as an argument to a function, the resulting C++ code is invalid. As a result, the tool should be extended to change the declaration of such functions to accept `std::array`s[2].

5.3.1 Command line parsing

The functionality described in this section is an extension of the C-style array converter that is focusing solely on the parameter conversion, so it has no changes to the command line parsing as compared to the original tool.

5.3.2 AST node matching

[2]A different and potentially better approach is discussed in section 7.2.2

For this augmentation of the existing tool, a new matcher has to be created. The matcher must find all the function parameter declarations which are constant arrays. The matcher should additionally still extract the needed information like the variable declaration converter (section 5.2.2).

There is an existing matcher in the LibTooling catalogue that will match a function parameter declaration. That matcher is called `parmVarDecl` and it will be used in this matcher instead of the `declaratorDecl` used previously.

The built-in matcher `parmVarDecl` provides a way to match all parameter declarations, i.e., parameters of functions or methods. For this tool, the parameters of interest are only `ConstantArrayType`s. However, writing such a matcher is not trivial, due to certain aspects of the language.
At the lowest level of abstraction, a C-style array is essentially a pointer to a block of contiguous memory. This means that when a C-style array is used in certain contexts, it can be treated as a pointer, which is commonly referred to as array decaying. Clang AST uses this concept and represents C-style array parameters as pointers instead of arrays. The challenge was therefore to differentiate between normal pointer types from decayed ones.
Fortunately, there is a built-in matcher called `decayedType`, which allows matching on decayed pointers.

Matching on all `DecayedType`s, however, is too general, as it is also possible for the other array types (section 5.2.2) and function types to decay into pointers. Unfortunately, a matcher that only matches on `DecayedType`s that were previously `ConstantArrayType`s does not exist within the built-in libraries. It is, however, possible to pull the information from a node, as a `DecayedType` can be cast to an `AdjustedType`, which is a type that was implicitly adjusted based on the semantics of the language, i.e., due to array decay in this scenario. `AdjustedType` contains meta-information of both the new type and the original type.

Since `AdjustedType` exists, it is possible to write a custom matcher that finds the original type of the node, and thereby is able to check if the node decayed from a `ConstantArrayType`. Creating a custom matcher is done through helper macros in the library. There are many different helper macros, which allow the creator of the matcher to fine-tune the matcher to the exact needs. This flexibility also makes it somewhat complicated for first-time developers, as there are many options to sort through. In this case, the matcher is provided with an `AdjustedType` node, which has to be filtered on the original type. The original type has to be compared with another type. This means that a parameter is needed for the matcher. The correct macro for this type of matcher is the `AST_MATCHER_P` macro. This macro allows the user to specify the input node type and a single parameter which is given to the matcher. Because this matcher will compare types, the type of the input parameter is a `Matcher<QualType>`. The signature of the matcher is shown on listing 5.19.

```
AST_MATCHER_P(AdjustedType, hasOriginalType, ast_matchers::internal::Matcher<QualType>, InnerType) {
    ...
}
```

Listing 5.19: Signiture of the custom matcher `hasOriginalType`.

The implementation of the matcher is quite simple. As mentioned earlier, the matcher needs to extract the original type from the `AdjustedType` and compare it to the type provided as the parameter of the matcher. This can be achieved with the code shown on listing 5.20.

```
return InnerType.matches(Node.getOriginalType(), Finder, Builder);
```

Listing 5.20: Implementation of the custom matcher `hasOriginalType`.

The `Finder` and `Builder` variables are common across all matchers. The `Finder` variable is the `MatchFinder` variable created by the tool, and it is responsible for calling the callbacks when a valid match has been found. The `Builder` variable is used to bind nodes to specific names through the `.bind(NAME)` construct used in the tools.

With all the building blocks in place, the C-style array parameter matcher can be constructed. It looks similar to the variable C-style array matcher and is therefore very expressive. The C-style array parameter matcher can be seen on listing 5.21.

```
auto ParmConstArrays = parmVarDecl(
        isExpansionInMainFile(),
        hasType(
            decayedType(hasOriginalType(constantArrayType().bind("parm")))),
        hasTypeLoc(typeLoc().bind("parmLoc"))
    ).bind("parmDecl");
```

Listing 5.21: C-style array parameter matcher.

5.3.3 Node data processing

The goal of this tool expansion is the same as it was for C-style array variables, it just has to use `ParmVarDecl` s instead of `VarDecls` . `ParmVarDecl` s are a specialisation of `VarDecl` s, so all the node processing is the same for both types. Therefore the node processing can be reused for this tool expansion, and the different match filtering is all that is needed.

5.3.4 Handling the results

As this is an expansion to the C-style array converter tool, the handling of the results is identical.

5.4 Enum-to-string tools overview

This section describes a tool that is capable of generating `std::string_view to_string(EnumType e)` functions for each enum declaration defined in a C++ program. The `to_string` functions take an instance of the enum as an argument and returns a string corresponding to the name of the enumerator.

An example of the outputs of running the tool can be seen in listing 5.22. In the example, in part (1), the enum `Animal` is declared with two enumerators: Dog and Cat. In part (2), the `to_string` function that would be generated by the tool can be seen. Parts (3) and (4) show another enum declaration with another generated `to_string` function.

```cpp
// (1): Example enum declaration:
enum class Animal{
    Dog, // Dog is an example of an enumerator (aka. enum constant)
    Cat // Cat is another example of an enumerator
};

// (2): Function that the tool generates:
constexpr std::string_view to_string(Animal e){
    switch(e) {
        case Animal::Dog: return "Dog";
        case Animal::Cat: return "Cat";
    }
}

// (3): Another enum declaration:
enum Greetings {
    ... // enumerators for Greetings
};

// (4): The other enum declaration also gets a to_string function
constexpr std::string_view to_string(Greetings e){
    switch (e) {
        ... // cases for Greetings
    }
}
```

Listing 5.22: Example (1) declaring an enum in C++ and (2) the `to_string` function that the tool generates. In (3) another enum was declared from which another `to_string` function is generated (4).

5.4.1 Difference from previous tools

The tool differentiates itself from the previous examples by being a generative tool, meaning that it inserts source code into a file. In contrast, the renaming tool and C-style comparison tool were refactoring tools that would overwrite existing code lines. While the differences may seem subtle, it can be more challenging to design generative tools, as the generated code should be syntactically valid as part of the code context. E.g., with the enum-to-string tool, it is necessary to determine if a function named `to_string` with the same signature exists in the namespace. This has to be considered since the redefinition of functions is not allowed in C++. Because such a function can exist, a strategy for handling it must be determined. This could be solved in different ways e.g., by leaving the function untouched or overwriting it. The process of identifying the different edge cases which have to be handled can be very challenging for the tool writers. The only conceivable way of catching the edge cases is by writing tests and running the tool on existing databases. It can also be quite a challenge to ensure that all the edge cases which have been identified are handled in the tool. E.g. in the enum-to-string tool, it is quite a complex task to determine if there is an existing "to_string" method as one must analyze the compilation unit for its existence.

Likewise, there are typically extra semantic considerations to be made when designing a generative tool. E.g., if `std::string_view to_string(Animal e)` function exists in a namespace "A" and `Animal` was declared in namespace "B", then it would be syntactically correct to add the `to_string` function to namespace "B". The question of whether it semantically makes sense for these functions to coexist arises [3] and one needs to select a strategy for handling such scenarios.
Examples of such strategies could be to ignore the cases, warn the user about them, delete the non-generated version or overwrite the non-generated version.
It can be difficult to find and consider all the possible semantic strategies when developing a tool. For some problems, there could be infinite ways to generate the wanted behaviour, like there is when creating a program through a programming language. Some of the possibilities may be better than others but there is still a large design space that could be explored. To demonstrate this, a list of scenarios where one might need to consider the behaviour of the enum-to-string tool can be seen below. The examples in the list increasingly become more abstract and difficult to implement.

- A `to_string` function taking multiple arguments already exists.
- The enum is declared privately inside a class.
- The enum is declared inside an anonymous namespace.
- The enum is declared inside a namespace that by convention is intended to be ignored by users (e.g., `detail`, `implementation`, etc.).
- A function that implements the same behaviour as the generated `to_string` function exists.
- A function that implements a similar behaviour as the generated `to_string` function exists.
- A similarly named function exists that implements a similar behaviour as the generated `to_string` function exists (e.g. `toString`).
- ...

For the enum-to-string tool, it was decided to overwrite syntactically conflicting implementations of the `to_string()` functions. This has the benefit of allowing the user to change the enum and re-run the tool to see the updated changes. E.g., in listing 5.22 if a `Animal::Horse` was added to the enum declaration, re-running the tool would update the corresponding `std::string_view to_string(Animal e)` function. However, it also has the downside of essentially reserving the `to_string` name leaving the user unable to write their own versions of the function.
Furthermore, for this tool, it was decided that if a `to_string` function already exists in a different namespace, then the existing version must be overwritten. The reason for implementing this semantic rule was mainly because there are some interesting challenges to consider concerning the recursive traversal of the namespaces, which are described in section 5.5.2.

[3]This must be determined on a case-by-case basis. E.g. it might make sense for two `print(X)` functions to exist in separate namespaces. One that is part of the public API and one that is intended for debugging. However, it might not make sense for two `release(X)` functions to exist in separate namespaces as this would indicate there are several ways of releasing the resources allocated in X. (And yet in other cases, it might make perfect sense for two `release(X)` functions to exist.)

5.4.2 Implementations

The enum-to-string tool is more complex than the previously described tools as it consists of matching on multiple independent declarations of the source code simultaneously, i.e., the enum declarations and the existing `to_string` function declarations. The added complexity allows for a wider variety of design approaches, which was shown as part of the project, where three different implementations were considered and two were implemented.

The first approach could implement the tool in a single step, similarly to how it was done in section 5.2. Pseudocode for such a tool can be seen in listing 5.23. The pseudocode iterates over all the enum declarations in the source code and determines if a `to_string` already exists. Depending on the outcome, the function is either updated or generated.

```
for enum_decl in source_files:
    to_string_inst := find(to_string(enum_decl))
    if to_string_inst:
        update to_string_inst
    else:
        generate to_string(enum_decl)
```

Listing 5.23: Pseudocode for the enum-to-string tool.

An alternative implementation of the tool can be seen in listing 5.24, which follows a multi-step procedure. The first step consists of updating the existing `to_string` functions and saving the relevant enum types in a collection (`parameters`) for later use. The second step consists of finding all the enum declarations and generating the `to_string` functions that were not updated in step 1.

```
parameters := []
for existing_enum_to_string in source_files:
    update existing_enum_to_string
    parameters.append(existing_enum_to_string.parameter)

for enum_decl in source_files:
    if not enum_decl in parameters:
        generate to_string(enum_decl)
```

Listing 5.24: Pseudocode for the enum-to-string tool.

A third way of implementing the tool can be seen in listing 5.25, which is also a multi-step procedure. The first step consists of identifying the existing `to_string` functions. The second step consists of finding all the enum declarations. If the declaration already has a `to_string` function, found in step 1, then it is updated. Otherwise, it is generated.

The main difference between the second and the third implementation is regarding semantics. It might be simpler to divide the tool into two distinct phases consisting of a data collection phase and a post-processing phase, where the post-processing performs the actual logic of the tool - in this case, the updating/generation of the "to_string" function. This appears to be a common division of responsibilities and is among others used in a helper library that was written by Bloomberg [27].

```
parameters := []
for existing_enum_to_string in source_files:
    parameters.append(existing_enum_to_string)

for enum_decl in source_files:
    if enum_decl in parameters:
        update parameters[enum_decl]
    else:
        generate to_string(enum_decl)
```

Listing 5.25: Pseudocode for the enum-to-string tool.

The two designs from listing 5.24 and listing 5.25 have the benefit of being more modular than the one in 5.23 since they implement the tool behaviour in multiple steps. Each step in the designs can essentially be considered independent tools which are then chained together. This makes it possible to split the tool development across team members and also makes it easier to test the tool and reuse it in other projects. For instance in listing 5.24, the first for-loop can be considered a tool that identifies existing enum-to-string functions, logs them and updates them, and the second for-loop can be considered a tool that generates enum-to-string functions if they are not in the log.

During the project, the tool was implemented with the designs seen in listing 5.23 and listing 5.24. The design from listing 5.25 was considered but was not implemented.
The following sections describe the two "enum-to-string" tools that were implemented.

5.5 Enum-to-string – single-step

The following section describes the enum-to-string tool where the entire tool is implemented as a single `ClangTool`. The tool follows the structure of listing 5.23.

5.5.1 Command line parsing

Most of the command line parsing was implemented as described in section 5.1. However, the behaviour was extended with two new options; "in_place" and "debug _info". The "in_place" option allows the user to specify whether the changes should be printed to the terminal or saved directly to the file. The "debug_info" option makes the tool print extra debug information to the console during execution.

The additional options were introduced as booleans options through the LLVM command line API, as seen in listing 5.26. It is simple to add command line options as one simply needs to specify a description and add it to the `OptionCategory` – which is `MyToolCategory` in this case.

```
static llvm::cl::opt<bool> Inplace(
    "in_place",
    llvm::cl::desc("Inplace edit <file>s, if specified. If not specified the "
                   "generated code will be printed to cout."),
    llvm::cl::cat(MyToolCategory));
static llvm::cl::opt<bool> DebugMsgs(
    "debug_info", llvm::cl::desc("Print debug information to cout."),
    llvm::cl::cat(MyToolCategory));
```

Listing 5.26: Implementation of the newly introduced command line options.

The options can then be used as normal booleans throughout the implementation, as seen in listing 5.27.

```
if (!Inplace) {
    llvm::outs() << new_code.get();
}
```

Listing 5.27: Using the `Inplace` command line option to print the changes to the command line if `--in_place` was not specified when running the tool.

The addition of "in_place" option is in particular useful for the future development of tools, as it allows for easier system testing. It does so because test cases can be run on small and very specific virtual files instead of an existing code base. Similar tests can be seen throughout the tools in the LLVM repository.

5.5.2 AST node matching

The AST node matching was by far the most challenging part of developing this tool as one needs to write a matcher that implements the following logic for each enum declaration:

1. Find and bind enum declaration
2. Find the outermost namespace
3. Recursively traverse the namespace to potentially find a matching `to_string` function

The behaviour described above is quite complex compared to the previous tools and required the implementation of recursive matchers, which the built-in matchers do not support.
The custom matchers will be described in the steps below.

Finding enum declarations

A matcher for finding the enum declarations can be seen in listing 5.28 with the parts related to finding the `to_string` functions left out. The matcher is fairly straightforward except for lines 4 and 5. These are discussed below the listing.

```
1 auto enumFinder = enumDecl(
2         isExpansionInMainFile(),
3         has(enumConstantDecl(hasDeclContext(enumDecl().bind("enumDecl")))),
4         matchers::is_named(),
5         optionally(
6             // Find matching enum_to_string
7         ));
```

Listing 5.28: Matcher for finding enum declarations.

The intention of line 3 in listing 5.28 is simply to bind the enum declaration, similarly to how it was done in the previous tools. However, the implementation is different from the other tools, since the binding is used by the inner `to_string` function matcher. I.e., if the implementation was written as `enumDecl(optionally(/*find to_string*/)).bind("enumDecl")`, the “enumDecl” binding would not be accessible to the “to_string” matcher, as the outer binding only happens after all the inner matchers are evaluated. Therefore, it was necessary to make the binding earlier, and line 3 is a way of achieving this. The behaviour of the line is to find an enum declaration with an enum constant declaration[4] and then backtrack from the enum constant declaration in order to bind the original enum declaration. This logic allows the enum declaration to be bound before running the “to_string” matcher.

Line 4 in listing 5.28 fixes a bug that was found when running the tool on an external project, i.e., the JSON [28]. Before this line was introduced, the tool was unable to handle unnamed enums [5], which would throw an unhandled exception.
In the AST, an `EnumDecl` inherits from a `NamedDecl`, which contains the `IdentifierInfo` related to the node. An unnamed enum can be interpreted as a `NamedDecl` with no `IdentifierInfo` since it does not have a name. The matcher can therefore be written as listing 5.29, where the `IdentifierInfo*` is implicitly converted to a boolean, returning false in it is a nullptr.

```
AST_MATCHER(NamedDecl, is_named) {
   return Node.getIdentifier(); // nullptr if no name
}
```

Listing 5.29: Custom matcher for determining if a `NamedDecl` has been given a name.

Finding the outermost namespace

In order to potentially find the `to_string` method matching the `EnumDecl`, one must consider that the function might be placed inside a different namespace. For that reason, it is necessary to consider all the namespaces inside the compilation unit when searching. The implementation for such a matcher is somewhat similar to the implementation of the built-in `hasDeclContext` that was used in listing 5.28. The behaviour of `hasDeclContext` is to return false, if the declaration does not have a `DeclContext` [6], otherwise return the result of evaluating the `InnerMatcher` on the context casted to a `Decl`.
The behaviour for the recursive version[7] (`has_rec_decl_context`) is similar, but instead of evaluating the inner matcher in the immediate parent context, it is evaluated in the outermost context. The implementation can be seen in listing 5.30.

[4]Note that this also filters out empty enum declarations which are uninteresting in terms of having a `to_string` function.
[5]Unnamed enums were mainly used in C++ before `constexpr` was introduced, for defining compile-time evaluated constants for metaprogramming usages.
[6]An example of a case of a `Decl` that does not have `DeclContext` is the outermost `TranslationUnitDecl`.
[7]Note that while the implementation seen in listing 5.30 has been transformed to its iterative version, the methodology can be considered recursively traversing the AST.

```
AST_MATCHER_P(Decl, has_rec_decl_context, Matcher<Decl>, InnerMatcher) {
   auto cur_ctx = Node.getDeclContext();
   if (!cur_ctx) {
      return false;
   }
   const DeclContext *nxt_ctx = nullptr;
   while (true) {
      nxt_ctx = cur_ctx->getParent();
      if (!nxt_ctx) {
         return InnerMatcher.matches(*Decl::castFromDeclContext(cur_ctx), Finder, Builder);
      }
      cur_ctx = nxt_ctx;
   }
}
```

Listing 5.30: Custom matcher for finding the outermost context of an AST node.

Find matching "to_string"

The remaining part of the `enumFinder` matcher that was seen in listing 5.28 is to potentially identify the `to_string` functions. The implementation can be seen in listing 5.31 with a description below the listing.

```
auto enumFinder = enumDecl(
  /* The rest of enumFinder...*/
  optionally(
    matchers::has_rec_decl_context(hasDescendant(
      functionDecl(
        hasName("to_string"),
        parameterCountIs(1),
        hasParameter(0,
          parmVarDecl(hasType(
            elaboratedType(namesType(
              hasDeclaration(
                equalsBoundNode("enumDecl")))))
          ).bind("parmVar"))
      ).bind("toString")))));
```

Listing 5.31: The part of the `enumFinder` matcher that was left out of listing 5.28. It is responsible of optionally finding a `to_string` function matching the "enumDecl".

The `optionally` matcher indicates that the inner matcher is optional. `has_rec_decl_context` is then used to traverse to the outermost context. Curiously enough, unlike `hasDeclContext`, the `hasDescendant` function recursively matches on the descendants. This means that it can be utilized to recursively match the `functionDecl`s in the context. There are a couple of restrictions placed on the `functionDecl` in order to be matched. It has to have the correct name and only a single parameter. The parameter also has to be identical to the previously bound "enumDecl" node. The equality between the bound node and the parameter type is expressed through a series of matchers. As it is the type of the parameter which is interesting, the `hasType` matcher is used. As the enum type may have a namespace qualifier, the inner matcher must take that into account and that is done through the `elaboratedType` [8] matcher. The type

[8] An `elaboratedType` refers to a type that potentially has a qualifier ahead of it.

has to be extracted from the `elaboratedType` and that is done through the `namesType` matcher. The “enumDecl” tag, which was bound earlier, refers to a `Decl` instead of a `Type`, so `hasDeclaration` is used to refer to the underlying `Decl` stored in the `elaboratedType`. The last step is to compare the `Decl` to the bound “enumDecl” node.

The combined matcher satisfying the steps defined at the start of section 5.5.2 has now been completed. It should be clear to the reader that writing a single matcher that matches all the necessary nodes of the enum-to-string tool was no trivial achievement.

Finding the namespace in code

When developing the single-step enum-to-string tool, the initial solution also contained a potential binding of the namespace of the `parmVarDecl`, as it was thought to be needed during node data processing. Later during the “node data processing” step, this approach turned out to be unnecessarily complex compared to retrieving the namespace directly through the `SourceLocation`s. However, the resulting matcher turned out to be interesting and will be discussed in the following.

Binding the namespace of the `parmVarDecl` was challenging, as it, similarly to `has_rec_decl_context`, required recursively[9] traversing the AST until arriving at the outermost namespace qualifier of the `parmVarDecl`. Inspiration was found in the implementation of `specifiesNamespace` which only considers the immediate namespace of the node.

The implementation of the matcher can be seen in listing 5.32. Initially, it is verified that the node can be cast to a `NamespaceDecl`. The function `getPrefix()` can be used to check if there is a `NestedNameSpecifier` prefixing the current node. The function `getAsNamespace` can be used to convert the `NestedNameSpecifier` to its underlying `NamespaceDecl`. Finally, the `InnerMatcher` is run on the outermost `NamespaceDecl` of the `ParmVarDecl`.

```cpp
AST_MATCHER_P(NestedNameSpecifier, rec_specifies_namespace, Matcher<NamespaceDecl>, InnerMatcher) {
   auto ns = Node.getAsNamespace();
   if (!ns) {
      return false;
   }
   auto prefix = Node.getPrefix();
   while (prefix && prefix->getPrefix()) {
      ns = prefix->getAsNamespace();
      prefix = prefix->getPrefix();
   }
   return InnerMatcher.matches(*ns, Finder, Builder);
}
```

Listing 5.32: Implementation of the `rec_specifies_namespace` custom matcher.

5.5.3 Node data processing

The logic for the node data processing step of the enum-to-string tool can be seen in listing 5.33. The aforementioned `enumFinder` is used as the *matcher* and the *edits* is a single `changeTo` expression. Inside `changeTo` the `RangeSelector` is determined dynamically for each match, depending on whether “toString” was bound. If it is bound then the `SourceLocation` of the “toString” node is used, otherwise, the `SourceLocation` right after the “enumDecl” node is used. The edit is performed as seen in lines

[9] Once again, the implementation has been transformed into its iterative version.

5 to 10. The interesting parts are `print_correct_name` and `NodeOps::case_enum_to_string`, which will be discussed below.

```
1  auto enumRule = makeRule(
2      enumFinder,
3      changeTo(
4          ifBound("toString", node("toString"), after(node("enumDecl"))),
5          cat("\n\nconstexpr std::string_view to_string(",
6              print_correct_name,
7              " e){\n\tswitch(e) {\n",
8              run(case_enum_to_string(print_correct_name, "enumDecl")),
9              "\t}\n}")
10     ));
```

Listing 5.33: Implementation of the `enumRule` that is responsible for node data processing. The `RewriteRule` also adds the necessary headers, which have been left out of the listing to simplify.

Printing the correct name

In order to generate the `to_string` function, the correct notation for referring to the parameter must be used. E.g., if the enum declaration “Animals” is in the namespace “ns” and the `to_string` function is written outside the namespace, then the declaration must be `to_string(ns::Animals e)`. This is the behaviour that the callable `print_correct_name` implements. If the “toString” node is bound then the notation from the existing `to_string` function must be used, which is implemented in `run(get_declarator_type_text("parmVar"))`. Otherwise, the `to_string` function is generated in the same namespace as the enum declaration, so no prefix is necessary.

```
auto print_correct_name = ifBound("toString",
    run(get_declarator_type_text("parmVar")),
    cat(name("enumDecl")));
```

Listing 5.34: Logic behind `print_correct_name`.

The implementation of the stencil `get_declarator_type_text` can be seen in listing 5.35. The stencil operates on `DeclaratorDecl`s, which can be considered an abstraction that incorporates the shared behaviour of specific declaration types, such as `FunctionDecl` and `VarDecl`. It first extracts the `SourceRange` from the node, converts it to a `CharSourceRange` and extracts the corresponding text from the source file, which is finally returned.

```cpp
resType get_declarator_type_text(StringRef Id) {
return [=](const MatchFinder::MatchResult &Match) -> Expected<std::string> {
    auto node = Match.Nodes.getNodeAs<DeclaratorDecl>(Id);
    if (!node) {
        throw std::invalid_argument("ID not bound or not DeclaratorDecl: " + Id.str());
    }
    auto sourceRange = node->getTypeSourceInfo()->getTypeLoc().getSourceRange();
    auto charRange = CharSourceRange::getTokenRange(sourceRange);
    auto sourceText = tooling::getText(charRange, *Match.Context).str();
    return sourceText;
};
}
```

Listing 5.35: The `get_declarator_type_text` stencil.

Generating enum cases

The final part in terms of node data processing, is to generate the case expressions inside the `to_string` function, i.e., generate a case expression for each `EnumConstantDecl` inside the enum declaration.

Given the complexity of the stencil, the group decided to employ the concept of “function composition” to combine several simpler stencils into a more intricate one. This approach allows for the construction of a complex stencil by sequentially applying and composing simpler stencils.

The first stencil that was developed with such an approach, is `foreach_enum_const` and can be seen in listing 5.36. The generic stencil allows for the execution of a callback function for each `EnumConstantDecl` that is defined inside an `EnumDecl` and concatenates the results into a single string. The callback function is required to accept two parameters: A `MatchResult` and an `EnumConstantDecl*`. Additionally, the function should return a string as its result. The specific function definition is enforced through metaprogramming techniques, although they are omitted from the example provided in the listing.

```cpp
template <typename F>
resType foreach_enum_const(StringRef Id, F callback) {
   return [=](const MatchFinder::MatchResult &Match) -> Expected<std::string> {
      if (auto enum_decl = Match.Nodes.getNodeAs<EnumDecl>(Id)) {
         std::stringstream ss;
         for (const auto enum_const : enum_decl->enumerators()) {
            ss << callback(Match, enum_decl, enum_const);
         }
         return ss.str();
      }
      throw std::invalid_argument("ID not bound or not EnumDecl: " + Id.str());
   };
}
```

Listing 5.36: The `foreach_enum_const` stencil.

A stencil capable of generating case expressions for each `EnumConstantDecl` was now required. This is seen in listing 5.37 where `case_enum_to_string` is implemented. The stencil is capable of generating a case expression for a single `EnumConstantDecl`. It is then used in conjunction with `foreach_enum_const` to iterate over each `EnumConstantDecl`.

The stencil takes another stencil as a parameter that is used to retrieve the correct identifier to use in the switch case. In practice, this involves using `print_correct_name` as seen in listing 5.34. The stencil then returns a string containing the complete case expression, e.g., `case ns::Animal::Dog: return "Dog";`.

```
resType case_enum_to_string(StringRef Id, Stencil getName) {
   auto lambda = [getName](const MatchFinder::MatchResult &Match, const EnumDecl *enum_decl,
                  const EnumConstantDecl *enum_const_decl) {
         auto name = getName->eval(Match);
         if (!name) {
            throw std::invalid_argument("Could not get name");
         }
         return "\t\tcase " + name.get() +
               "::" + enum_const_decl->getNameAsString() + ": return \"" +
               enum_const_decl->getNameAsString() + "\";\n";
      };
   return foreach_enum_const(Id, lambda);
}
```

Listing 5.37: The `case_enum_to_string` stencil.

5.5.4 Handling the results

The results are handled identically to how it was done in the “C-style array converter” tool section 5.2.

5.6 Enum to string – multiple steps

There are few differences between the multi-step and single-step enum-to-string tools and they are all in either the matching or the node processing steps. However, the command line and result handling are identical between the two tool types.

The structure of the tool, as shown in listing 5.24, is to first find, log and update all the existing “to_string” methods and secondly to generate “to_string” methods for the rest of the enum declarations. This will be done with two `ClangTool` s that are executed in succession. These are respectively denoted the “to_string” and the enum declaration tools. The rest of the section is structured by consecutively describing the relevant parts of each tool.

5.6.1 AST node matching – “to_string” tool

In the multi-step version, there is no longer a need to recursively traverse the declaration context through `matchers::has_rec_decl_context(...)`. However, the `functionDecl` matcher from listing 5.31 can be reused. Additionally, the matcher in this version is intended to find all “to_string” methods and not a specific one, so the complicated comparison between the parameter type and the bound enum declaration can be left out. The “to_string” matcher for this version of the tool can be seen on listing 5.38.

```
auto to_string_matcher = functionDecl(
  isExpansionInMainFile(),
  hasName(to_string_method),
  hasParameter(0,
    parmVarDecl(
     hasType(
       enumDecl().bind("enumDecl"))).bind("enumParm"))).bind("toString");
```

Listing 5.38: The final “to_string” matcher for the multi-step version of the enum-to-string tool.

5.6.2 Node data processing – “to_string” tool

The node data processing needed to update the existing “to_string” functions is similar to how it was done in the single-step tool. However, the “to_string” tool also needs to log which `EnumDecl` s were used as parameter in the updated “to_string” functions, for the second part of the multi-step tool. Two challenges were identified regarding this: How to log an `EnumDecl` and how to use the logged declarations for filtering.

Logging a declaration consists of recording data from the node and saving it for later usage. When logging an `EnumDecl` it is necessary to identify a unique way of classifying the specific `EnumDecl` . There are multiple conceivable ways of doing this. One way could be to save the source code location of the declaration. Another could be to save the fully qualified name of the declaration[10]. The choice of data representation should align with the filtering methodology. In this case, the matcher `hasAnyName` will be used, and its details will be provided in the forthcoming section. However, it is worth mentioning that `hasAnyName` expects a vector of names, hence making it the designated data representation.

Due to the decision of using `hasAnyName` , the fully qualified names had to be extracted from the bound `EnumDecl` s. This can be done through a stencil, which takes the id as well as a vector, for the names, as parameters. The fully qualified name can be extracted from any `NamedDecl` with the method `getQualifiedNameAsString` and then inserted into the vector. The implementation of the function can be seen on listing 5.39.

```
1  auto addNodeQualNameToCollection(StringRef Id, std::vector<std::string> *decls) {
2     auto lambda = [=](const MatchResult &Match) -> Expected<std::string> {
3        if (auto *decl = Match.Nodes.getNodeAs<NamedDecl>(Id)) {
4           decls->emplace_back("::" + decl->getQualifiedNameAsString());
5           return "";
6        }
7        throw std::invalid_argument(append_file_line("ID not bound or not NamedDecl: " + Id.str()));
8     };
9     return lambda;
10 }
```

Listing 5.39: The implementation of the function which extracts the fully qualified name of a bound `NamedDecl` . The addition of the “::” on line 4 will be further discussed in the upcoming section.

[10]The fully qualified name is the name of the declaration with all the namespaces, and it can be used as a unique identifier as this is required by the C++ specification.

5.6.3 AST node matching – enum declaration tool

The node matching for the enum declaration part of the tool is similar to the single-step tool, except it must also handle the aforementioned filtering. When working with LibTooling, utilizing a narrowing matcher is often the simplest approach to implementing a node filter. By looking through the reference of the built-in matchers, the `hasAnyName` matcher was discovered. This matcher compares the name of a `NamedDecl` [11] to a vector of names and returns true if the name matches any of the specified names. Since the desired behaviour is the opposite, the matcher `unless` can be used to logically invert the expression. The availability of `hasAnyName` ultimately led to the decision of using the fully qualified name of the `EnumDecl` s as the identifier.

The matcher for the enum declaration tool can be seen in listing 5.40, with most of it being reused from the single-step tool.

```cpp
auto find_other_enums = enumDecl(
    isExpansionInMainFile(),
    matchers::is_named(),
    unless(hasAnyName(existing_enums))
  ).bind("enumDecl");
```

Listing 5.40: The matcher for finding enum declarations with no existing "to_string" function. `existing_enums` is a vector of fully qualified names of the enums

It is important to note that due to the internal workings of the `hasAnyName` matcher, the first tool needs to be executed before instantiating this matcher. This is because `hasAnyName` creates a copy of the names in the collection at instantiation time. Another thing to note about the `hasAnyName` , is that it has two modes that define its behaviour. If none of the names in the provided collection contains "::", then the matcher will do a non-qualified check on the AST. If however, any of the names contain the "::" string, then the matcher will use a fully qualified name check. In this tool, the fully qualified should be used in all cases, and this is the reason for the addition of the "::" on line 4 of listing 5.39.

5.6.4 Node data processing – enum declaration tool

The node data processing for the second tool is similar to the single-step tool. The difference between the two is that all the existing "to_string" methods have been handled, so the dynamic `RangeSelector` can be removed in favour of placing the "to_string" function after the declaration of the enum.

[11] `ValueDecl` s, such as an `EnumDecl` , all inherit from the `NamedDecl` class.

6 Testing

The group was curious regarding the potential performance differences between the two implementations of the enum-to-string tool. This investigation aimed to provide insights into the performance implications associated with each approach. Prior to conducting the tests, the assumption was that the multi-step tool would be the slowest due to its requirement of running two tool invocations. However, there was also an expectation of performance penalties with the recursive matchers used in the single-step tool.

The tests were performed on a laptop with 16GB RAM and an Intel i7-8565U CPU[1]. Furthermore, the laptop was running Linux as the operating system.

The first tests were run on a simple test file that was also used to verify the behaviour of the tools. The tools were invoked on the file 100 times and the results can be seen in table 6.1. The "user" and "sys" fields indicate the processing time spent in user mode and kernel mode respectively. The "sum" field indicates the sum of the two fields.

Tool	User	Sys	Sum
Single-step enum-to-string	11.395s	0.936s	12.331s
Multi-step enum-to-string	0.591s	0.469s	1.060s

Table 6.1: Results when running the tools 100 times on a simple test file.

The tools were also tested on "JSON" which is a popular open-source JSON header-only library, containing around 25000 lines of code. The tools were invoked once on the JSON library and the results can be seen in table 6.2.

Tool	User	Sys	Sum
Single-step enum-to-string	3m58.606s	0m30.003s	4m28.609s
Multi-step enum-to-string	3m45.828s	0m28.958s	4m14.786s

Table 6.2: Results when running the tools once on the JSON library.

Overall, the results of both test scenarios were quite surprising.
The results from table 6.1 indicate that the multi-step tool was more than 11 times faster than the single-step tool when running on the simple test file. It is expected that this difference is due to the recursive matchers being ineffective.
The results from table 6.2 were perhaps even more surprising, as they showed that there was only a 5.28% difference[2] between the two tools when running on the JSON library. The group's current hypothesis is, that it is due to the parsing of the AST as the file is so large. An attempt was made to validate this hypothesis by profiling the tools. However, the profiling process proved challenging as it was difficult to differentiate the application code from the library code. This challenge arose from a combination of factors, including the dynamic linking of the library code and the use of custom stencils and matchers passed as parameters in the application code. In the case of dynamic linking, the profiler can only capture and profile the public function calls available in the library's API. This means that libray's internal function calls are invisible to the profiler. Since the custom stencils and matchers are essentially passed as function pointers, they are being executed inside the internal parts of the library, which is inaccessible to the profiler. As a result of the challenges, the profiling results were inconclusive.

The results of the tests should be interpreted cautiously due to the small sample size and potential variations in other source files.

[1] An 8th generation mid-tier CPU designed for power efficiency.

[2] Calculated with $\frac{|268.609-254.786|}{(268.609+254.786)/2}$

7 Discussion

Write something here

7.1 Finding the correct node types and data methods

Much of the development time for the C-style array conversion tool and the enum-to-string tool was spent looking through the different AST node types defined in the LibTooling library and determining how the data was represented in the AST. This was a long process during the development phase of the project, as the library defines many different class specializations and almost identical node types with vastly different data. One example of these difficulties was the extraction of the qualifier for the `DeclaratorDecl` in the C-style array conversion tool. The qualifier is the explicit namespace in front of the variable declaration, e.g., the qualifier of `A::B::C` would be `A::B::`. This was eventually extracted directly from the source code through the `getQualifierLoc` method implemented for `DeclaratorDecl` nodes, but many other options such as `getQualifiedName` method were tried first. These small differences between node types and method names took a while to navigate through and because the information is needed both when creating the matchers and when doing the data extraction from the nodes it was the most time-consuming part of creating a tool. It can also be quite hard to even identify which nodes are of interest, as there are so many different ones to choose from. In the end, the group developed the tools by figuring out what data was needed, then looking through the existing matchers/nodes in order to find a promising candidate and then trying the method out. This method is a systematic form of trial and error and caused some frustration in the process because it is so hard to identify the exact combination of node types and method calls to get the wanted information.

7.2 Semantic considerations

In this section, some of the semantic considerations that were processed and discovered during the development of the C-style array conversion tool will be discussed. The C-style array tool in particular led to many discussions regarding language semantics, as the transformation is non-trivial when all potential scenarios must be considered. Furthermore, it is crucial for an automatic refactoring tool to handle all scenarios correctly, as introducing a bug can be difficult to spot across a large codebase and defeats the purpose of automatic refactoring.

7.2.1 Array conversions for constant types

The conversion from a C-style `int[5]` array to a `std::array` is straightforward. The type and size are moved inside the template parameter list, e.g., `int array[5]` becomes `std::array<int, 5> array`.

A more interesting scenario to consider is how to handle a C-style array of constant integers (`const int array[5]`). For this conversion, there are two possibilities. Either with a constant `std::array` declaration (`const std::array<int, 5> array`), or with a constant template parameter (`std::array<const int, 5> array`). The two solutions are almost identical in meaning, as they will both prevent the programmer from changing the elements inside the `std::array`. The difference between the two approaches is that the non-const member functions[1] are unavailable for `const std::array<..>` version, as it follows the normal rules for const-qualified member functions. They are technically available to the other version, but calling them will result in a compile-time error as the functions will attempt to modify

[1]An example of a non-const qualified member function for `std::array` is `swap` that swaps the contents of two arrays.

the values of the `const int` s. As a result, the main distinction for the programmer lies in the type of compile-time error that occurs and the readability of the code.
For readability of the code and the error messages, the group prefers the `const std::array<..>` representation. However, in the implementation of the conversion tool, the std::array<const ...> version is created. This choice is made because it is easier in LibTooling to extract the entire array element type, including the const qualifier, than it is to separate the two. Since the difference between the two versions is so minimal the ease of development was prioritised.

Pointer type arrays

Similar considerations have to be made regarding pointers but they are perhaps more interesting, as they can have multiple const qualifiers. They can either be pointers to constant objects, constant pointers to objects or constant pointers to constant objects (respectively `const int*` , `int* const` and `const int* const`) [29]. Like the array of value types, arrays of pointer types can be converted directly to `std::array` s by moving the type into the template parameter list. E.g., `const int*[4]` becomes `std::array<const int*, 4>` and `int* const[4]` becomes `std::array<int* const, 4>` .

It is also possible to represent a constant array with a pointer type through `const std::array<int*, N>` . However, this representation is tricky, as it is unclear which of the aforementioned pointer type arrays it corresponds to. A simple way of testing the similarity between the representations is seen in listing 7.1. The idea behind the test is that a `const Test*` and `const Test* const` should not be allowed to execute the `test` method as it is a non-const member function. However, `Test* const` should. This logic can be used to infer, that if `a0->test()` compiles, then `const std::array<Test*, 1> a0` must be similar to `Test* const a0[1]` . The results indicated that the code compiled successfully. This means that `const std::array<int*, 1>` and `std::array<int* const, 1>` are similar except for the differences discussed earlier for the value type arrays. The implementation in the tool uses the `std::array<int* const, 1>` version as it is easier to extract from the AST nodes.

```cpp
struct Test {void test(){}};
int main() {
    Test t;
    const std::array<Test*, 1> a0 {&t};
    a0[0]->test();
}
```

Listing 7.1: Test of conversion similarity .

7.2.2 Array parameter conversions

8 Related work

The focus of this project has been on deterministic source-code generation using user-provided source-code as the specification. This is, however, not the only methodology of source-code generation.

Similar work was made in [30] which provides a presentation on how LibTooling can be used as a deterministic source-code generation tool. However, [30] provides a more superficial overview of this, and focuses on the general challenges associated with developing such tools. For instance, the work in [30] implements a simpler enum-to-string tool[1] where existing "to_string" functions are not taken into consideration, meaning that the tool can not be used consecutively on a code base.

One example of a different approach to source-code generation comes from microcontroller manufacturers such as STMicroelectronics [31]. Tools such as STM32CubeIDE enable developers to conveniently add startup and configuration code to their codebases through a user-friendly checkbox tool [32]. The configuration code is pre-written by the manufacturers and it is used to specify the state of the peripherals inside the microcontrollers.
This form of source-code generation is also deterministic but raises the level of abstraction by defining a predetermined set of options. These options serve as the specification instead of the users' code. The limited set of options simplifies the number of edge cases that must be considered but at the cost of flexibility. In contrast, the tools developed during this project are generic allowing them to be used with any C++ specification, as demonstrated with the JSON library [28] in chapter 6.

Another approach to automatic programming and source-code generation is to use tools that are based on a probabilistic model. A probabilistic model can be very useful for source-code generation, even though it is non-deterministic [33]. This type of source-code generation is often employed to generate code from natural language (NL)[2] descriptions of problems, as deterministic models can be impractical [34]. Research on probabilistic automatic programming has been ongoing for years and the newest tools in the field are actively being used by the public [35, 36]. This type of model usually consists of some kind of neural network which is used to process the specification. The result of the processing can then, in some cases, be used as a specification for a deterministic model as shown in [37].
The distinction between deterministic and probabilistic tools lies in their approach to transformation. Deterministic tools strictly adhere to a predefined set of rules and only execute those specific transformations. Consequently, deterministic tools may not be able to handle all types of inputs or prompts. In contrast, probabilistic tools operate based on probabilities and strive to respond to all inputs, although the quality of the responses may vary.
As a result, deterministic tools may be more suitable to deploy in a completely automated environment where precise and reliable transformations are required. On the contrary, probabilistic models may be more suitable for assisting humans in the development process, where their flexibility and ability to handle a wide range of inputs can be valuable.

[1]To compare, their matcher simply consists of `enumDecl(isExpansionInMainFile())`.
[2]An example of a NL could be English.

9 Conclusion

During this project, multiple tools were developed using the LibTooling library created by Clang. The tools each presented different challenges concerning tool creation and understanding of C++ source code. Through the developed tools numerous crucial considerations, that must be taken into account during tool development, were identified. It was shown through the tools that it is possible to both refactor existing and generate code. Through experimentation with the two versions of the enum-to-string tool, it was found that using multi-step tool development leads to easier development and faster tools.

9.1 Future work

In order to fully explore the different tool structures, the third concieved implementation strategy for the enum-to-string tool (listing 5.25) could be explored. This third version could then be compared to the two versions developed for this project in order to determine which strategy is easiest to work with.

After the development for this project was finished, some edge cases were discovered for the enum-to-string tool implementations. One of the unhandled edge cases is as follows: If an enum definitions is located in the header file, then the tool will look for "to_string" function implementations in that header file only and not all the provided .cpp files. This can lead to the generation of invalid source-code as multiple definitions of the "to_string" function exist. Another limitation of the implemented tools is that they do not handle the difference between a function definition and a function declaration. This limitation can also lead to invalid source-code generation. In order to use the enum-to-string tool in a development environment these edge cases should be handled.

References

[1] Avron Barr and Edward A. Feigenbaum. "Automatic Programming". In: *The Handbook of Artificial Intelligence*. Vol. 1. Elsevier, 1982, pp. 295–379. ISBN: 978-0-86576-090-5. DOI: 10.1016/B978-0-86576-090-5.50010-0. URL: https://linkinghub.elsevier.com/retrieve/pii/B9780865760905500100 (visited on 01/31/2023).

[2] Gordon S. Novak Jr. *CS 394P: Automatic Programming p. 2*. The University of Texas at Austin. URL: https://www.cs.utexas.edu/users/novak/cs394p2.html (visited on 04/03/2023).

[3] Visual Paradigm. *UML/Code Generation Software*. URL: https://www.visual-paradigm.com/features/code-engineering-tools/ (visited on 04/03/2023).

[4] LLVM. *The LLVM Compiler Infrastructure Project*. URL: https://llvm.org/ (visited on 04/03/2023).

[5] Clang. *Clang C Language Family Frontend for LLVM*. URL: https://clang.llvm.org/ (visited on 04/03/2023).

[6] LLVM. *LibTooling — Clang 17.0.0git Documentation*. URL: https://clang.llvm.org/docs/LibTooling.html (visited on 02/23/2023).

[7] Swarnim. *Problems & Pains in Parsing: A Story of Lexer-Hack - DeepSource*. URL: https://deepsource.com/blog/problems-and-pains-in-parsing (visited on 05/29/2023).

[8] Clang. *Introduction to the Clang AST — Clang 17.0.0git Documentation*. URL: https://clang.llvm.org/docs/IntroductionToTheClangAST.html (visited on 04/03/2023).

[9] Clang. *Clang-Tidy — Extra Clang Tools 17.0.0git Documentation*. URL: https://clang.llvm.org/extra/clang-tidy/ (visited on 05/29/2023).

[10] Clang. *Clang Plugins — Clang 17.0.0git Documentation*. URL: https://clang.llvm.org/docs/ClangPlugins.html (visited on 05/29/2023).

[11] Clang. *How to Write RecursiveASTVisitor Based ASTFrontendActions. — Clang 17.0.0git Documentation*. URL: https://clang.llvm.org/docs/RAVFrontendAction.html (visited on 03/17/2023).

[12] Clang. *Welcome to Clang's Documentation! — Clang 17.0.0git Documentation*. URL: https://clang.llvm.org/docs/index.html (visited on 04/03/2023).

[13] Clang. *Clang Transformer Tutorial — Clang 17.0.0git Documentation*. URL: https://clang.llvm.org/docs/ClangTransformerTutorial.html (visited on 02/08/2023).

[14] Clang. *Tutorial for Building Tools Using LibTooling and LibASTMatchers — Clang 17.0.0git Documentation*. URL: https://clang.llvm.org/docs/LibASTMatchersTutorial.html (visited on 02/02/2023).

[15] CMake. *Cmake-Generators(7) — CMake 3.26.0 Documentation*. URL: https://cmake.org/cmake/help/latest/manual/cmake-generators.7.html#cmake-generators (visited on 03/16/2023).

[16] IBM. *IBM Documentation*. Apr. 14, 2021. URL: https://ibm.com/docs/en/i/7.3?topic=linkage-name-mangling-c-only (visited on 02/28/2023).

[17] Clang. *Matching the Clang AST — Clang 17.0.0git Documentation*. URL: https://clang.llvm.org/docs/LibASTMatchers.html (visited on 02/08/2023).

[18] Firat Kasmis. *Clang Out-of-Tree Build*. Jan. 28, 2023. URL: https://github.com/firolino/clang-tool (visited on 02/28/2023).

[19] Cppreference. *Undefined Behavior - Cppreference.Com*. URL: https://en.cppreference.com/w/cpp/language/ub (visited on 02/28/2023).

[20] Clang. *JSON Compilation Database Format Specification — Clang 17.0.0git Documentation*. URL: https://clang.llvm.org/docs/JSONCompilationDatabase.html (visited on 02/16/2023).

[21] Clang. *External Clang Examples — Clang 17.0.0git Documentation*. URL: https://clang.llvm.org/docs/ExternalClangExamples.html (visited on 02/23/2023).

[22] LLVM. *The LLVM Compiler Infrastructure - GitHub*. LLVM, Apr. 3, 2023. URL: https://github.com/llvm/llvm-project (visited on 04/03/2023).

[23] Clang. *Clang Format - ClangFormat.Cpp - GitHub*. Clang, Apr. 3, 2023. URL: https://github.com/llvm/llvm-project/blob/db3dcdc08ce06e301cdcc75e2849315a47d7a28d/clang/tools/clang-format/ClangFormat.cpp (visited on 04/03/2023).

[24] Clang. *AST Matcher Reference*. URL: https://clang.llvm.org/docs/LibASTMatchersReference.html (visited on 02/07/2023).

[25] Cppreference. *Std::Array - Cppreference.Com*. URL: https://en.cppreference.com/w/cpp/container/array (visited on 05/26/2023).

[26] Mitre. *CWE - 2022 CWE Top 25 Most Dangerous Software Weaknesses*. URL: https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html (visited on 05/26/2023).

[27] Bloomberg. *Clang-Metatool - A Framework for Reusing Code in Clang Tools*. Bloomberg, May 4, 2023. URL: https://github.com/bloomberg/clangmetatool (visited on 05/04/2023).

[28] Niels Lohmann. *JSON for Modern C++*. Version 3.11.2. Aug. 2022. URL: https://github.com/nlohmann (visited on 05/11/2023).

[29] Cppreference. *Pointer Declaration - Cppreference.Com*. URL: https://en.cppreference.com/w/cpp/language/pointer (visited on 05/27/2023).

[30] Sergei Sadovnikov. “Using Clang for Source Code Generation”. In: ().

[31] STM. *STMicroelectronics: Our Technology Starts with You*. URL: https://www.st.com/content/st_com/en.html (visited on 05/29/2023).

[32] STM. *STM32Cube Development Software - STM32 Open Development Environment - STMicroelectronics*. URL: https://www.st.com/en/ecosystems/stm32cube.html (visited on 05/29/2023).

[33] Mark Chen et al. *Evaluating Large Language Models Trained on Code*. Comment: corrected typos, added references, added authors, added acknowledgements. July 14, 2021. arXiv: 2107.03374 [cs]. URL: http://arxiv.org/abs/2107.03374 (visited on 01/31/2023). preprint.

[34] Uri Alon et al. *Structural Language Models of Code*. Comment: Appeared in ICML’2020. July 29, 2020. arXiv: 1910.00577 [cs, stat]. URL: http://arxiv.org/abs/1910.00577 (visited on 01/31/2023). preprint.

[35] *What Is ChatGPT and Why Does It Matter? Here’s What You Need to Know*. ZDNET. URL: https://www.zdnet.com/article/what-is-chatgpt-and-why-does-it-matter-heres-everything-you-need-to-know/ (visited on 05/29/2023).

[36] Johnmaeda. *Choosing an LLM Model*. May 23, 2023. URL: https://learn.microsoft.com/en-us/semantic-kernel/prompt-engineering/llm-models (visited on 05/29/2023).

[37] Pengcheng Yin and Graham Neubig. “A Syntactic Neural Model for General-Purpose Code Generation”. In: *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). Vancouver, Canada: Association for Computational Linguistics, 2017, pp. 440–450. DOI: 10.18653/v1/P17-1041. URL: http://aclweb.org/anthology/P17-1041 (visited on 01/31/2023).