View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by ELTE Digital Institutional Repository (EDIT)

Eötvös Loránd University

Faculty of Informatics

Department of Programming Languages and Compilers

Clang-based Variable Name Suggestions for C++

Advisor:

Zoltán Porkoláb

Associate Professor

Author:

Mátyás Végh

Computer Science BSc

Budapest, 2019

Contents

1 Introduction **5**
1.1 Thesis Structure 5

2 User Documentation **7**
2.1 Building and Installation 7
2.1.1 Prerequisites 7
2.1.2 Building 8
2.1.3 Installing 9
2.2 Using 9
2.2.1 Generating the Variable Database 9
2.2.2 Generating Variable-name Suggestions 12
2.2.3 Performing a Rename 13
2.2.4 Using the Vim Plugin 14
2.3 Troubleshooting 15
2.3.1 Variable to suggest for is not in the database 15
2.3.2 Code Compilation Fails After a Rename 16

3 Developer Documentation **17**
3.1 Architectural Overview 17
3.1.1 Database Format 18
3.2 Clang-plugin Architecture 19
3.2.1 Plugin 19
3.2.2 Action 19
3.2.3 Consumer 20
3.3 Implementation 21

3.3.1 Visiting the Abstract Syntax Tree 21
3.3.2 Merging the Databases 24
3.3.3 Making Suggestions 25
3.3.4 Renaming 26

4 Testing **27**
4.1 Robot 27
4.1.1 Dump-Names 27
4.1.2 Suggest-Names 29

5 Summary **32**
5.1 Results 32
5.2 Further Work 32
5.2.1 Variable-kind Aware Suggestions 32
5.2.2 Only Suggest Names that can be Used 33
5.2.3 Follow Function-calls 33
5.2.4 Ignore Certain Paths 33
5.2.5 Speed 34

Appendices **35**

A Schema of Variable-Name JSON **36**

B Vim Plugin Documentation **38**

C Robot Keywords for Database Handling **40**

List of Figures

3.1 Overview of Architectural Elements 17
3.2 Structure of the Database 18
3.3 Loading the AST Action 19
3.4 Action Collaboration Diagram 20
3.5 Consumer Collaboration Diagram 21
3.6 Visitor Collaboration Diagram 22

Listings

2.1 Installing pip requirements 7
2.2 Installing apt requirements 8
2.3 Adding the LLVM apt repository 8
2.4 Invocation of Clang 9
2.5 Invocation of Clang with Defaulted Output Filename 10
2.6 CMake Compiler Setting 10
2.7 CMake Compiler Arguments 10
2.8 CMake Invocation without the Plugin 10
2.9 CMake Invocation with the Plugin 11
2.10 `example.cpp` 11
2.11 `examples.cpp.o.names.json` 11
2.12 `suggest_names --help` 13
2.13 Suggestions for `Foo::id` 13
2.14 `suggest_names_rename --help` 13
2.15 Invocation of `suggest_name_rename` 14
2.16 Adding the Vim plugin 14
2.17 Setting Databases Path 15
2.18 Traceback when requesting names for a variable not present . 15
3.1 Example Variable 24
3.2 Merged Variable 24
A.1 Schema of variable name database 36
B.1 Vim plugin documentation 38
C.1 DatabaseKeywords.robot 40

Chapter 1

Introduction

> There are only two hard things in Computer Science: cache invalidation and naming things.
>
> Phil Karlton

Several programs were written as part of this thesis, which when used together, assist a software developer in giving variables in their C++ code-base good names.

The primary goal of this thesis is to provide a refactoring tool that is aware of all variable declarations and their relations, so that accurate suggestions can be made. This is achieved by providing a compiler-plugin for Clang, the C++-compiler of the LLVM compiler infrastructure [1].

The compiler-plugin generates a database containing information on variables, which are then used by a set of scripts to suggest variable names, and to perform renaming operations.

A Vim-plugin is also provided to allow vim users to integrate the tools provided into their daily workflow.

1.1 Thesis Structure

This thesis is composed of multiple chapters, each dealing with a certain part of the work involved, from how to use the tools through how it is built

and tested. Chapter 1 on the preceding page provides the motivation for this project. Chapter 2 on the next page provides a guide on how to build, install and use the project. Chapter 3 on page 17 has detailed descriptions of the implementation, algorithms used, and the architectural overview. Chapter 4 on page 27 describes the testing, including a list of tests written. The last chapter 5 on page 32 summarizes the work, and what further enhancements can be made.

Chapter 2

User Documentation

This chapter describes how to use the various components included in this thesis.

2.1 Building and Installation

2.1.1 Prerequisites

This project was developed on Debian Stretch [2], and tested on Ubuntu Xenial [3].

2.1.1.1 Python packages required

To build, install and test the components of this project, the following dependencies are required: • meson [4] • ninja [5] • robotframework [6] • jsonschema [7] • yq [8]

These can be installed with:

Listing 2.1: Installing pip requirements

```
pip install meson ninja robotframework jsonschema yq
```

2.1.1.2 Apt packages required

• python3-pip [9] • python3-setuptools [10] • libclang-7-dev [11] • clang-7 [12] • llvm-7-dev [13] • libboost-all-dev [14] • texlive-full [15] • rubber [16]

• wget [17] • ca-certificates [18] • jq [19]

These can be installed with:

Listing 2.2: Installing apt requirements

```
apt-get install python3-pip python3-setuptools libclang-7-dev clang-7
    llvm-7-dev libboost-all-dev texlive-full rubber wget ca-certificates
    jq
```

If the LLVM-7 packages are not available for your distribution, they can be added with:

Listing 2.3: Adding the LLVM apt repository

```
apt-add-repository \
        deb http://apt.llvm.org/xenial/ llvm-toolchain-xenial-7 main
```

To help users on older distributions where the LLVM-7 packages may require packages which are not available, this project is also compatible with Clang and LLVM version 4.

2.1.1.3 LaTeX packages required

- Tikz-UML [20]

2.1.2 Building

Once the git [21] repository for this thesis project is cloned, setup a build directory and run meson.

```
cd /path/to/repository
mkdir build
cd build
meson ..
cd ..
```

The default build target will build the project, along with this thesis.

```
ninja -C build
```

To run the tests, use `ninja -C build test`.

2.1.3 Installing

Add suggest_names to your PATH.

```
$ export PATH=/path/to/repository/suggest_names:$PATH
```

2.2 Using

2.2.1 Generating the Variable Database

Before any variable suggestions can be made, the source-code must be indexed. This is achieved with a clang-plugin, which during compilation collects all variable occurences, and information about them, such as their type and name.

2.2.1.1 Clang Invocation

To generate a database for a given source file, run clang as follows:

Listing 2.4: Invocation of Clang

```
clang++ \
        -fplugin=/path/to/repository/build/libdump_names.so \
        -Xclang \
        -plugin-arg-dump-names \
        -Xclang \
        output-file \
        -Xclang \
        -plugin-arg-dump-names \
        -Xclang \
        /path/for/database \
        all... \
        other... \
        args... \
        /path/to/source.cpp
```

If output-file is not specified, the output file is derived from the output of compilation. For instance:

Listing 2.5: Invocation of Clang with Defaulted Output Filename

```
clang++ \
        -fplugin=/path/to/repository/build/libdump_names.so \
        -c
        /path/to/source.cpp
        -o /path/to/output.o
```

Results in `/path/to/output/o.names.json`. This is most useful when using the Clang-plugin as part of a build-system, as described next.

2.2.1.2 Build-System Integration

To generate the variable databases for a project larger than a few files, integration into the project's build-system is needed. As variable databases are generated per translation-unit, this means we have to change to way the C++ files are compiled, namely that Clang is used as the compiler, and that our plugin is used during compilation.

Several build-systems provide the option of the user to specify the compiler to be used, as well as additional compiler arguments. For CMake [22], these can be achieved as follows:

Set the compiler to Clang with:

Listing 2.6: CMake Compiler Setting

```
CMAKE_CXX_COMPILER=clang++
```

Set the compiler flags to use the plugin with:

Listing 2.7: CMake Compiler Arguments

```
CMAKE_CXX_FLAGS='-fplugin=/path/to/libdump_names.so'
```

For instance, when bootstrapping a project, instead of

Listing 2.8: CMake Invocation without the Plugin

```
mkdir build
cd build
cmake -G "Ninja" ..
```

do

Listing 2.9: CMake Invocation with the Plugin

```
mkdir build
cd build
cmake -G "Ninja" \
        -DCMAKE_CXX_COMPILER=clang++ \
        -DCMAKE_CXX_FLAGS='-fplugin=/path/to/libdump_names.so' \
        ..
```

2.2.1.3 An Example Run

In the following sections we will refer to the following example piece of C++ code:

Listing 2.10: `example.cpp`

```
1  struct Foo {
2      int id;
3  };
4
5  void f(int identifier) {
6      // ...
7  }
8
9  void g(int identifier) {
10     Foo f;
11     f.id = identifier;
12 }
```

Running the Clang-plugin as described above will give us a JSON file containing information on the symbols in the program:

Listing 2.11: `examples.cpp.o.names.json`

```
1  {
2      "Variables": [
3          {
4              "type": "int",
5              "name": "id",
6              "location": "./example.cpp:2:9",
7              "occurences": [
8                  "./example.cpp:11:7"
```

```
9                ]
10           },
11           {
12               "type": "int",
13               "name": "identifier",
14               "location": "./example.cpp:5:12",
15               "occurences": [
16               ]
17           },
18           {
19               "type": "int",
20               "name": "identifier",
21               "location": "./example.cpp:9:12",
22               "occurences": [
23                   "./example.cpp:11:12"
24               ]
25           },
26           {
27               "type": "struct Foo",
28               "name": "f",
29               "location": "./example.cpp:10:9",
30               "occurences": [
31                   "./example.cpp:11:5"
32               ]
33           }
34       ],
35       "Filename": "./example.cpp"
36   }
```

By default, all paths in the output are absolute paths. To make the output more easily legible, we pass the `-fdebug-prefix-map=/path/to/repository=.` command-line argument to Clang, so that the output paths are relative to the root of the repository.

2.2.2 Generating Variable-name Suggestions

Once the code-base is indexed, suggestions can be requested with the script `suggest_names`.

`suggest_names`' documentation is as follows:

Listing 2.12: `suggest_names --help`

```
usage: suggest_names [-h]
                     database [database ...] filename line
                     column

Suggest variable names

positional arguments:
  database     JSON produced by clang-plugin containing all
               variables
  filename     Filename containing variable to suggest
               names for
  line         Line number of variable to suggest names for
  column       Column number of variable to suggest names
               for

optional arguments:
  -h, --help  show this help message and exit
```

Invoking `suggest_names` to suggest an alternative to `Foo::id` can be done as follows:

```
suggest_names varnames.json ./examples/example.cpp 2 9
```

The resulting suggestions are as follows:

Listing 2.13: Suggestions for `Foo::id`

```
identifier
id
```

2.2.3 Performing a Rename

Renames can be performed with `suggest_names_rename`. When a rename is made, all references to the variable are adjusted to the new name, even accross translation-unit boundaries.

The documentation for `suggest_names_rename` is as follows:

Listing 2.14: `suggest_names_rename --help`

```
usage: suggest_names_rename [-h] --filename FILENAME
```

```
                            --line LINE --column COLUMN
                            --name NAME
                            database [database ...]

Rename variables

positional arguments:
  database              JSON produced by clang-plugin
                        containing all variables

optional arguments:
  -h, --help            show this help message and exit
  --filename FILENAME   Filename containing variable to
                        suggest names for
  --line LINE           Line number of variable to suggest
                        names for
  --column COLUMN       Column number of variable to
                        suggest names for
  --name NAME           New name for the variable
```

To accept `identifier` as the new name for `Foo::id`, we can then invoke `suggest_names_rename` as follows:

Listing 2.15: Invocation of `suggest_name_rename`

```
suggest_names_rename \
        --filename test.cpp --line 2 --column 6 \
        --name identifier \
        varnames.json
```

2.2.4 Using the Vim Plugin

To help making renaming easier, a Vim [23] plugin is included. With it, the renaming process can be done interactively without needing to use command-line tools. The Vim plugin can be added to your `.vimrc` by adding the following two lines:

Listing 2.16: Adding the Vim plugin

```
set rtp+=/path/to/repository/vim
helptags /path/to/repository/vim/doc
```

Following this, the documentation of the Vim plugin is available through the usual commands of Vim, such as `:help`.

To configure the path of variable databases, use:

Listing 2.17: Setting Databases Path

```
let g:suggest_names_database_path = '/path/to/project/builddir'
```

The output of querying the help for `:help suggest-names.txt` can be found in appendix B on page 38.

2.3 Troubleshooting

This section describes potential error messages the user may see during use of the tools described prior, and how to resolve them.

2.3.1 Variable to suggest for is not in the database

If the user requests variable-names for a variable not present in any of the databases, `suggest_names` gives an error along the lines of:

Listing 2.18: Traceback when requesting names for a variable not present

```
Traceback (most recent call last):
  File "./suggest_names/suggest_names", line 48, in <module>
    main()
  File "./suggest_names/suggest_names", line 44, in main
    print_suggestions(args.database, args.filename, args.line
        , args.column)
  File "./suggest_names/suggest_names", line 28, in
     print_suggestions
    filename, line, column)
  File "./suggest_names/suggest_names", line 19, in
     _find_corresponding_variable
    raise ValueError('Variable to suggest for is not in the
        databases')
ValueError: Variable to suggest for is not in the databases
```

In this case, the error was caused by querying a variable at line 2, column 10 of `example.cpp` (see listing 2.10 on page 11). To fix this, we need to pass column *9* instead of 10 to get the proper result.

When using the Vim plugin, this does not occur even if the cursor is not on the first character of the variable, as the Vim plugin first computes the position of the start of the variable before requesting suggestions.

Also note, that column numbers are based on the number of characters that precede them, not their apparent position when displayed on a screen. This is mostly important in code-bases where tabs are used for indentation.

2.3.2 Code Compilation Fails After a Rename

If after a rename, the code no longer compiles, there are generally two possible reasons *1)* The variable-name chosen was already in use and *2)* The rename was performed with out-of-date databases. There is no fix for the user to apply in the former case, only to undo the change and pick a different name. The general solution would be for `suggest_names` to only suggest names that are not in use in the surrounding scope as described in subsection 5.2.2 on page 33. The latter can be solved by always performing renames right after build, where the databases are up to date.

Chapter 3

Developer Documentation

3.1 Architectural Overview

Figure 3.1 provides an overview of the system. The clang-plugin developed produces a database for each translation-unit, which can then be merged to make all variable declarations available to the frontend, which can then suggest names for the variables.

The structure of the database can be seen in figure 3.2 on the next page.

Figure 3.1: Overview of Architectural Elements

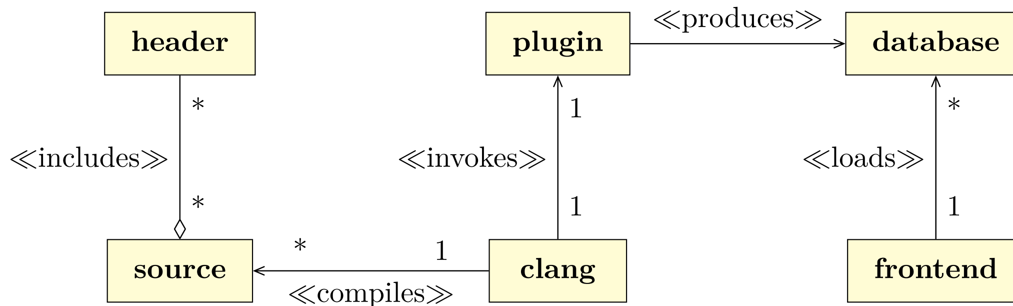
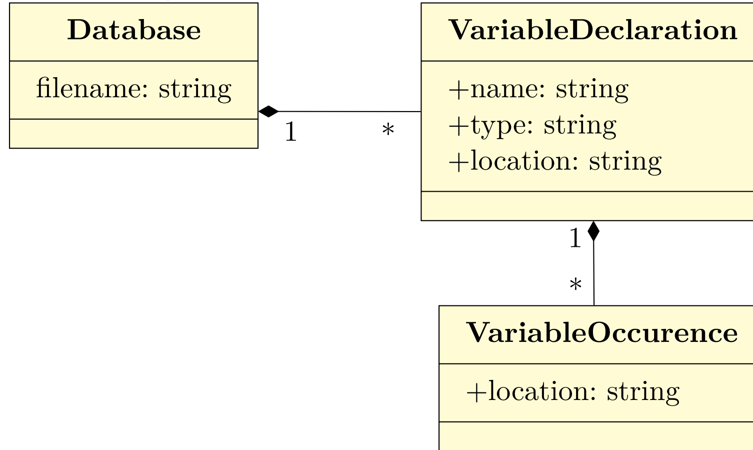


Figure 3.2: Structure of the Database



3.1.1 Database Format

The database produced by the compiler-plugin generates a list of declarations that are later used for suggesting variables. The database format is designed such that each translation unit creates its own database. This is so that when a project is being built with a build-system that supports incremental building –where each output binary is kept up-to-date by only recompiling the changed source files– the databases produced for those source files can then be recombined with the rest to produce a complete overview of all variables in the project. This is what makes cross translation-unit suggestions and renaming operations possible. For a formal schema of the database, see appendix A on page 36.

To support renaming of variables, we also need to track the location of any references to the variables. This is to ensure that after renaming the variable name at the point of declaration, we also must update every part of the code that refers to this variable. This can not be achieved by mere textual pattern-matching, as multiple scopes may have variables of the same name. This means we have to list all occurences during analysis. When merging the databases we also need to merge the list of occurences.

3.2 Clang-plugin Architecture

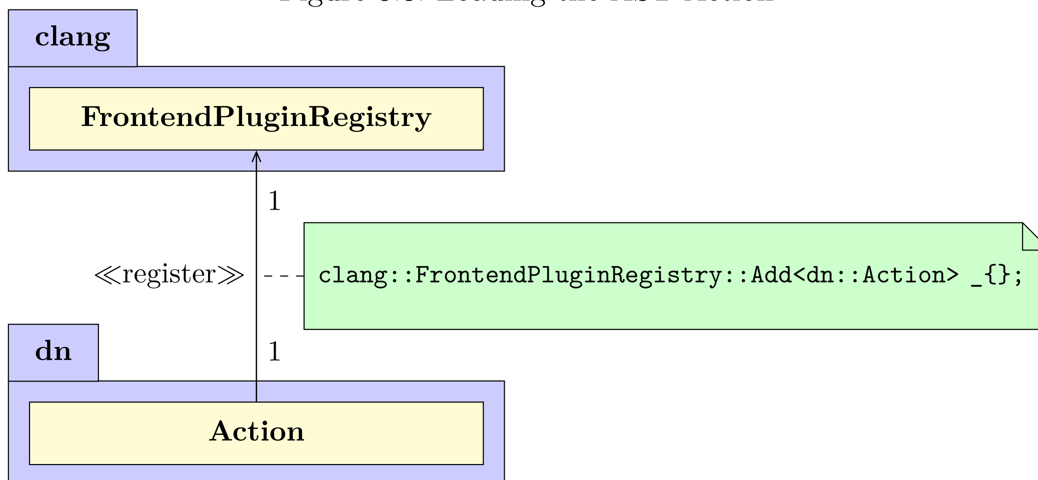
The clang-plugin contains all the code needed for traversing the Abstract Syntax Tree, along with the routines needed for dumping the names that occur in the codebase to a JSON file. All code in the clang-plugin is defined in `namespace dn`. The components listed here each derive from or interact with the interfaces of Clang designed for use in plugins.

Most of the work done by the plugin will be in visiting the Abstract Syntax Tree produced by Clang, which will be described in subsection 3.3.1 on page 21.

3.2.1 Plugin

The plugin first starts by loading `Action` into `clang::FrontEndPluginRegistry`.

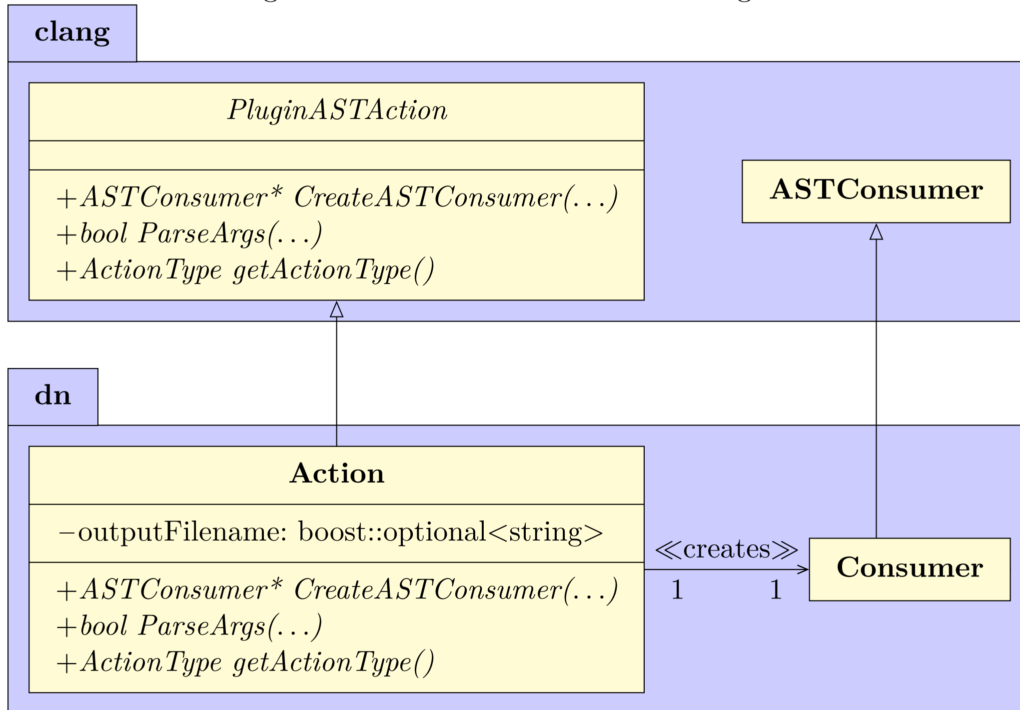
Figure 3.3: Loading the AST Action



3.2.2 Action

`Action` can then parse the command-line arguments passed to Clang, so that it can deduce the filename that was set for storing the output JSON. `Action` uses `clang::PluginASTAction::ActionType Action::getActionType()` to inform Clang that `Action` should not inhibit compilation, and should be invoked after the compilation has been done.

Figure 3.4: Action Collaboration Diagram

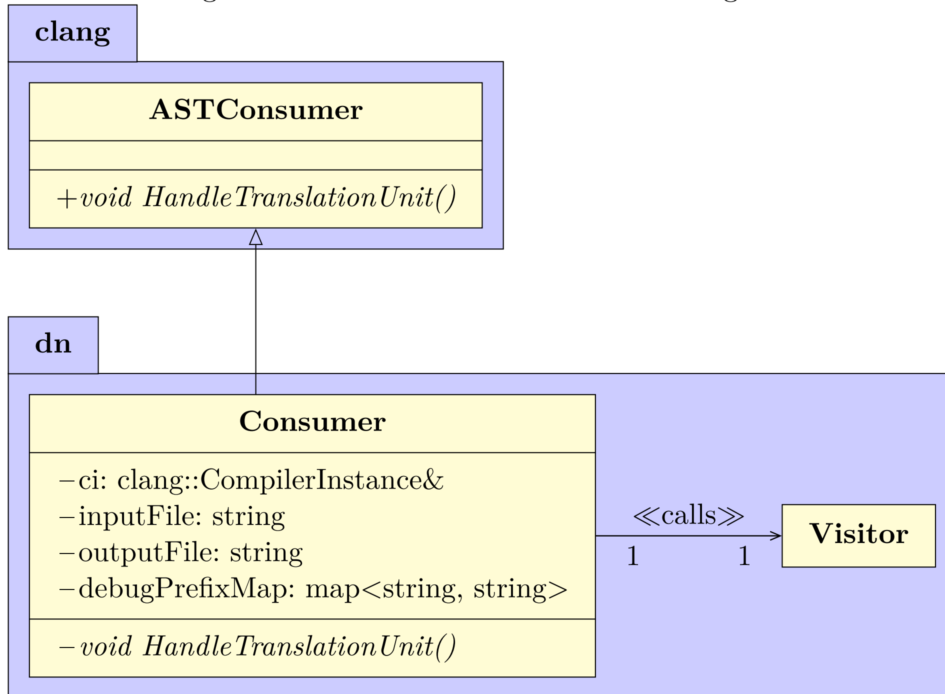


3.2.3 Consumer

Clang then loads the `Consumer` as specifed by `Action::CreateASTConsumer`, which can then proceed with traversing the Abstract Syntax Tree of the given Translation Unit.

`Consumer`'s sole responsibility is to invoke the `Visitor` when invoked by Clang. This is done in `Consumer::HandleTranslationUnit`, by first constructing a `Visitor`, calling it's `TraverseDecl` function with the current translation unit's declaration, then printing the result to the output file.

Figure 3.5: Consumer Collaboration Diagram



3.3 Implementation

3.3.1 Visiting the Abstract Syntax Tree

There are two main tasks to accomplish when traversing the Abstract Syntax Tree: the first is to record all variable declarations, the second is to record all references to these variables.

Both of these tasks are accomplished by the `Visitor` class. To facilitate these tasks, Clang's `RecursiveASTVisitor` provides two functions for us to override: `VisitDecl` and `VisitStmt`.

A diagram of the involved classes can be seen in figure 3.3.1 on the following page.

`VisitDecl` is used to visit all declarations as described in section 3.3.1.1 on the next page, whereas `VisitStmt` is used to visit all statements as described in section 3.3.1.2 on page 23.

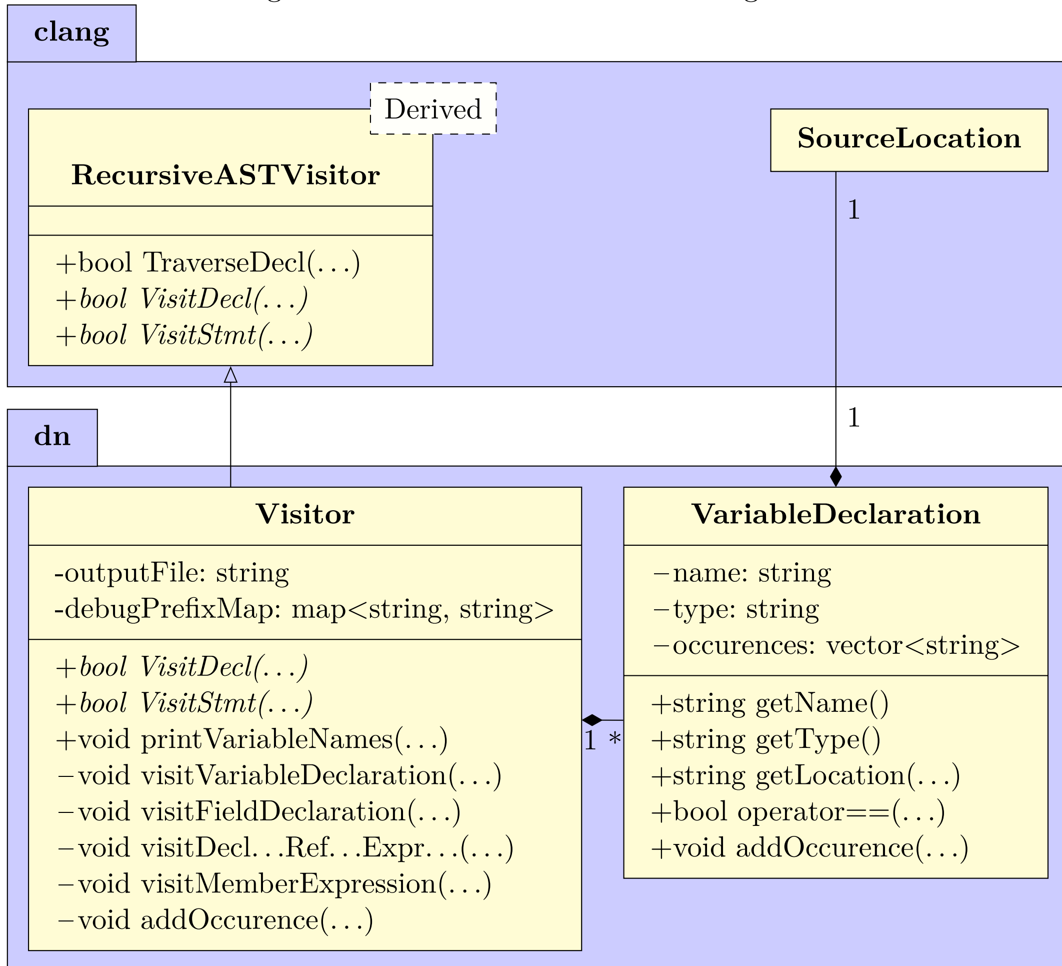


Figure 3.6: Visitor Collaboration Diagram

3.3.1.1 Visiting Declarations

There are two sorts of declarations that we are interested in: variable declarations, and field declarations.

The kinds a variable declaration can be are as follows:

- Function-local variable
- Global variable
- Function argument

Our task in all of these cases is to record the name, type and location

of the variable being declared. These are then used as a composite key to compare variables. The location not only contains the line and column of the variable, but also the filename. This is important as during the compilation of a translation unit, we may also visit declarations that come from another file, most notably header files. Once the databases are merged, duplicate variables will be removed, as described in section 3.3.2 on the next page.

3.3.1.2 Visiting Statements

When visiting statements, most statements are of no use to us. The only types of statements that are of interest to us, are those that refer to variable declarations (or field declarations). We record these references, so that when renaming a variable, we know what other parts of the code we need to change.

When traversing statements, we only need to handle `DeclRefExpr`s, and `MemberExpr`s, as all other expressions that refer to variable declarations derive from these classes.

In both cases, all we have to do is lookup the original declaration of the variable (or field) that the expression refers to, and add the current source location to its set of occurences. In most cases, the declaration will be visited before the statement that refers to it, however there are two notable exceptions.

The most obvious exception to this ordering is when we have a class that has a member function that refers to a member of the class that is defined before the member. In C++ the order of function definitions and members in a class defintion does not matter like it normally would, and Clang traverses these declarations and defintions roughly in the order they appear in the source-code. This means our visitor may occur a reference to a field before it has recorded the field itself.

The other –more subtle– case where we may see a reference before a declaration is in the case of template functions whose signatures are only determined fully upon instantiation. These function templates needn't be instantiated to be visited, however when they are, the arguments are visited as being references before they are visited as declarations.

In both of the cases above, we need to make sure that the variable to which we are trying to add an occurence already exists in the database. The naive approach to this is to record it if it is not already present, and doing the same when recording a declaration.

We cannot merely rely upon discovering all variable declarations by noticing them through statements, as some variables are declared but never referred to.

3.3.2 Merging the Databases

The database format as described in section 3.1.1 on page 18 was chosen to enable merging of the databases. As the database is merely a list of Variables, each with a type, name, location and a list of occurences, these can be merged simply. To support projects which may rename files during the lifetime of the project, we ignore databases that refer to files that no longer exist. These databases are referred to as stale.

Given a set of databases $input_database_1, \ldots, input_database_m$, we can construct the set of non-stale databases as follows:

$$database_1, \ldots, database_n = \left\{ input_database_i \Big|_{\wedge exists(input_database_i.filename)}^{i \in \{1,\ldots,m\}} \right\} \tag{3.1}$$

where $exists(filename)$ determines whether the file exists.

Given the following variable:

Listing 3.1: Example Variable

```
{
        "type": TYPE,
        "name": NAME,
        "location": LOCATION,
        "occurences": [ ... ]
}
```

and non-stale databases $database_1, \ldots, database_n$, we can construct the merged variable as follows:

Listing 3.2: Merged Variable

```
{
        "type": TYPE,
        "name": NAME,
        "location": LOCATION,
        "occurences": [
```

$$\bigcup_{i=1}^{n} \left\{ v.occurences \,\middle|\, \substack{v \in database_i.variables \\ \wedge v.type = TYPE \\ \wedge v.name = NAME \\ \wedge v.location = LOCATION} \right\}$$

```
        ]
}
```

Note that while each $input_database_i$ has a filename to determine whether it is stale, the output of the merge process does not include this field. This means that the merging operation is not associative, that is, merged databases cannot be merged further.

3.3.3 Making Suggestions

When the developer requests a suggestion for a name, the first thing we need to do is merge the databases that correspond to the project (as described in section 3.3.2 on the previous page).

Once we have a merged database of all variables, we can then make a suggestion based on the type of the variable we wish to rename.

3.3.3.1 Making Suggestions Based on Type

From the complete database named *database*, we first get all the candidate names by looking up all variables that have the same type as the variable we are suggesting names for. Let's call the variable we are suggesting a new name for *variable*. The set of names then becomes:

$$\{v.name \mid v \in database.variables \wedge v.type = variable.type\} \tag{3.2}$$

We also assign a weight to these names based on how frequently they occur.

3.3.4 Renaming

When renaming a variable, there are two main points at which to apply the rename, the declaration of the variable (or field), and at all occurences. This is implemented in the Python script `suggest_names_rename`.

The first thing this script needs to do, is load and merge all the databases, so that we can rename occurences of a variable located in different files.

Care must be taken when there are multiple references to a variable in the same line. If no two references occur in the same line, we can simply go to each location and change the old name to the new name, changing the length of the line as necessary. If there are multiple occurences in the same line however, changing the occurences left-to-right would invalidate the column index stored by the next occurence, so we have to perform each rename in the same line right-to-left. This invalidation also means that we require the user to rebuild their project to update the databases before another rename can be performed.

Chapter 4

Testing

This chapter describes the testing involved in this project. This project is tested using Robot Framework [6].

As the project is composed of multiple programs, each part is tested independently. This helps in comprehension of each test, as the reader can only deal with one part at a time, and also eliminates a combinatorial explosion that would emerge if every type of input that could emerge for the Clang-plugin then needed to be tested by way of testing the variable-name suggestions.

Keywords used by these tests can be found in appendix C on page 40.

4.1 Robot

4.1.1 Dump-Names

Tests that the compiler-plugin produces the appropriate database for an input. It achieves this by invoking the compiler with one or more source files, and then validates the database.

4.1.1.1 Variables

Tests that the part of the database that stores information about variable declarations is correct.

List of tests:

- *No Variables Should Make Variables Empty*

  Tests that an empty file produces an empty output.

  ```
  Given Empty File is Passed To Analyzer
   Then Variables Should Be Empty
  ```

- *Single Variable Makes Variables Have 1 Entry*

  Tests that an file with a single variable produces an output with a single entry. As this argument is modified twice, there should be two occurences.

  ```
  Given File single_int.cpp is Passed to Analyzer
   Then There Should Be ${1} Entries that Occur ${2} Times
  ```

- *Single Argument Makes Variables Have 1 Entry*

  Tests that a file with a single function with a single argument produces an output with a single entry.

  ```
  Given File single_int_argument.cpp is Passed to Analyzer
   Then There Should Be ${1} Entries that Occur ${1} Times
  ```

- *Single Unnamed Argument Makes Variables Empty*

  Tests that a file with a single function with a single unnamed argument produces an output with no entries.

  ```
  Given File unnamed_int_argument.cpp is Passed to Analyzer
   Then There Should Be ${0} Entries that Occur ${0} Times
  ```

- *Single Member of Struct Makes Variables Have 1 Entry*

  Tests that a file with a single struct with a single member produces an output with a single entry.

  ```
  Given File single_struct_member.cpp is Passed to Analyzer
   Then There Should Be ${1} Entries that Occur ${0} Times
  ```

- *Union Member Of Struct Makes Variables Have 3 Entries*

  Tests that a file with a single struct containing a union and a bool to discriminate upon, is handled correctly despite the indirect field.

  ```
  Given File member_union_access.cpp is Passed to Analyzer
   Then There Should be ${3} Entries that Occur ${1} Times
  ```

- *Union Makes Variables Have 3 Entries*

  Tests that a file with a function-local union and a bool to discriminate upon, is handled correctly despite the indirect variable.

  ```
  Given File union_access.cpp is Passed to Analyzer
   Then There Should be ${3} Entries that Occur ${1} Times
  ```

4.1.2 Suggest-Names

Tests that given appropriate databases of sources, the correct suggestions are made for each variable name. This is achieved by preparing databases and making expectations on the suggested name lists.

4.1.2.1 Merge

Tests that the logic involved in merging multiple databases merges the databases as though the source was concatenated before creating the database.

List of tests:

- *Empty database and empty database merge to empty database*

  ```
  Given Empty database
    And Empty database
   When Databases are merged
   Then The merged database has ${0} variables
  ```

- *Stale database and stale database merge to empty database*

  ```
  Given Stale database
    And Stale database
   When Databases are merged
   Then The merged database has ${0} variables
  ```

- *Stale database and empty database merge to empty database*

```
Given Stale database
  And Empty database
 When Databases are merged
 Then The merged database has ${0} variables
```

- *Empty database and single variable database merge to single variable database*

```
Given Empty database
  And Database with one variable
 When Databases are merged
 Then The merged database has ${1} variables
```

- *Stale database and single variable database merge to single variable database*

```
Given Stale database
  And Database with one variable
 When Databases are merged
 Then The merged database has ${1} variables
```

- *Two single variable databases merge to database with one variable*

```
Given Database with one variable
  And Database with one variable
 When Databases are merged
 Then The merged database has ${1} variables
```

- *Two different single variable databases merge to database with two variables*

```
Given Database with one variable
  And Database with an other variable
 When Databases are merged
 Then The merged database has ${2} variables
```

- *Same occurence passed twice gets merged to single occurence*

```
Given Database with one variable and one occurence
  And Database with one variable and one occurence
 When Databases are merged
 Then Each ${1} variables have ${1} occurences
```

- *Different occurences of same variable remain separate*

```
Given Database with one variable and one occurence
  And Database with one variable and an other occurence
 When Databases are merged
 Then Each ${1} variables have ${2} occurences
```

- *Different occurences of same variable remain separate and same ones get merged*

```
Given Database with one variable and one occurence
  And Database with one variable and one occurence
  And Database with one variable and an other occurence
 When Databases are merged
 Then Each ${1} variables have ${2} occurences
```

4.1.2.2 Suggest

Tests that the logic involved in suggesting variables based on complete information is correct.

List of tests:

- *Database with one variable should suggest its own name*

```
Given Database with one variable
  And Databases are merged
 When Suggestions are requested
 Then Suggestions have ${1} elements
```

- *Database with two variables of same type should suggest both*

```
Given Database with one variable
  And Database with an other variable
  And Databases are merged
 When Suggestions are requested
 Then Suggestions have ${2} elements
```

Chapter 5

Summary

5.1 Results

Several programs were implemented, that comprise a tool for suggesting variable-names for C++ codebases. This tool was tested automatically using Robot [6], and tested manually against Xerces-C++ [24]. Variable-name suggestions were based on the type of the variable in question, and suggestions and renaming was made available via a Vim plugin.

5.2 Further Work

Multiple improvements can be made to the way variable-name suggestions work, these are listed here.

5.2.1 Variable-kind Aware Suggestions

Several projects decorate variable names according to their kind, such as `m_` being used as a prefix for member variables. In Xerces-C++, the prefix `f` is used.

This should be incorporated into to the suggestion process, so that when `suggest_names` finds a candidate variable of kind A, for variable of kind B, the prefix (or other decoration) for A is removed, and the prefix for B is applied, thereby ensuring that variable-names suggested follow these conventions.

5.2.2 Only Suggest Names that can be Used

Section 2.3 on page 15 describes some error-cases, one of which being that `suggest_names` can suggest variable names after which the code fails to compile. A solution to this would be to track a list of conflicting variable names for each variable in the database. A variable A would conflict with a variable B, that is B could not be renamed to A, if A was declared in the same scope as B (as opposed to an ancestor scope), or if A was declared in an ancestor scope, but was referred to in the scope of B. This analysis would need to be performed by the Clang-plugin, as determining conflicts like this is C++ specific.

5.2.3 Follow Function-calls

Currently, suggestions are made purely based on the type of the variable. For certain types, the type does not strongly determine the name of the variable. For instance, a variable of type `int` could legitmately be called `i`, `fd`, `error`, or `temperatureOfBudapestInCelsius`. While many types can reasonably determine the names, types like `int` do not. In this case, we can follow the functions into which a variable is passed, and see what name was assigned to the parameter in the definition of the function, as well as the name of the returned value (if any). The current architecture of having several databases and then merging them would work well for this, as the function definition wouldn't need to be in the translation-unit where the rename takes place.

5.2.4 Ignore Certain Paths

Many code-bases have dependencies on large third-party code-bases, such as the C++ standard library, or Boost. Frequently however, code-bases use different coding conventions to that of the third-party library, so those could be omitted both for improving the results of variable-name suggestions. If they were omitted during the database-generation phase, merging would even be faster as there would be less variables to parse. For instance, a simple `#include <iostream>` typically generates upwards of 4500 variable dec-

larations, many of which are internal to the library and are of no use in renaming.

5.2.4.1 Map conventions

A more sophisticated version of the above, is to distinguish parts of the code-base that are written with different conventions. For instance, if a code-base calls file descriptors `fileDescriptor`, whereas the GNU C Library [25] calls them `__fd`, instead of ignoring calls into glibc, we could determine the convention as it applies in glibc, and map it back to the calling project's conventions.

5.2.5 Speed

Currently, there is a non-trivial performance cost of running variable-name analysis during the compilation. For instance, on an Intel Core i5-4590 CPU, Xerces-C++ without our plugin builds in 17s, where as takes upwards of 5 minutes with the plugin enabled. The primary cause of this is needing to track whether we have seen the variables declaration before adding an occurence. This is currently done with an array of Variables, storing an array of Occurences, which leads to cubic time-complexity. Using a hashmap or similar data structure would cut down the performance penalty drastically.

Appendices

Appendix A

Schema of Variable-Name JSON

Listing A.1: Schema of variable name database

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "type": "object",
  "title": "Schema for variable-name database",
  "required": [
    "Variables",
    "Filename"
  ],
  "properties": {
    "Variables": {
      "type": "array",
      "items": {
        "type": "object",
        "required": [
          "type",
          "name",
          "location",
          "occurences"
        ],
        "properties": {
          "type": {
            "type": "string",
```

```
            "examples": [
              "std::function<void (void)>"
            ],
            "pattern": "^(.*)$"
          },
          "name": {
            "type": "string",
            "examples": [ "f" ],
            "pattern": "^(.*)$"
          },
          "location": {
            "type": "string",
            "examples": [
              "/path/to/include/a.hpp:123:45"
            ],
            "pattern": "^(.*):[0-9]+:[0-9]+$"
          },
          "occurences": {
            "type": "array",
            "items": {
              "type": "string",
              "examples": [
                "/path/to/source/a.cpp:12:34"
              ],
              "pattern": "^(.*):[0-9]+:[0-9]+$"
            }
          }
        }
      }
    },
    "Filename": {
      "type": "string",
      "examples": [ "/dev/null" ],
      "pattern": "^(.*)$"
    }
  }
}
```

Appendix B

Vim Plugin Documentation

Listing B.1: Vim plugin documentation

```
suggest-names.txt            Suggest variable names for C++

============================================================
INTRODUCTION                 vim-suggest-names-introduction

This plugin wraps suggest_names so that you can
automatically have variable suggestions for C++ code.

============================================================
USAGE                               vim-suggest-names-usage

When editing C++, use :SuggestNames to have a buffer appear
with variable suggestions.

============================================================
COMMANDS                         vim-suggest-names-commands

vim-suggest-names provides the following commands:

:SuggestNames

------------------------------------------------------------
                                              :SuggestNames
```

```
:SuggestNames         Suggest variables for the variable
                      under the cursor.
                      The following maps are provided for the
                      suggestion buffer:

                      <CR> Accept the given suggestion, and
                           perform a rename of the variable.
                           This will trigger a change to all
                           buffers contain references to this
                           variable, so you will be prompted
                           to reload the impacted files.

==============================================================
OPTIONS                              vim-suggest-names-options

g:suggest_names_database_path

--------------------------------------------------------------
g:suggest_names_database_path

Path where name databases are stored. This path will be
recursively globbed in search of suitable jsons.

Default: $HOME/.suggest_names/
==============================================================
```

Appendix C

Robot Keywords for Database Handling

Listing C.1: DatabaseKeywords.robot

```
*** Settings ***
Library          Collections

*** Keywords ***
Empty database
    ${database} =       Make database
    Append To List      ${databases}    ${database}

Single variable
    [Arguments]         @{occurences}
    ${variable} =       Make variable   int     a
    ...                         test.cpp:1:1    ${occurences}
    Set Test Variable   ${variable}
    [Return]            ${variable}

Database with one variable
    ${variable} =       Single variable
    ${database} =       Make database   ${variable}
    Append To List      ${databases}    ${database}

Stale database
    ${variable} =       Make variable   int     stale
```

```
    ...                          /non_existent:1:1
    ${database} =       Make database    ${variable}
    ...                          filename="/non_existent"
    Append To List      ${databases}     ${database}

Database with one variable and one occurence
    ${variable} =       Single variable      test.cpp:2:1
    ${database} =       Make database    ${variable}
    Append To List      ${databases}     ${database}

Database with one variable and an other occurence
    ${variable} =       Single variable      test2.cpp:2:1
    ${database} =       Make database    ${variable}
    Append To List      ${databases}     ${database}

Database with an other variable
    ${variable} =       Make variable   int   b    test.cpp:2:1
    ${database} =       Make database    ${variable}
    Append To List      ${databases}     ${database}

Databases are merged
    ${merged_database} =     Merge databases    ${databases}
    Set Test Variable   ${merged_database}

The merged database has ${n} variables
    Length Should Be    ${merged_database.variables}    ${n}

Each ${n} variables have ${k} occurences
    Length Should Be    ${merged_database.variables}    ${n}
    :FOR    ${ELEMENT}  IN   @{merged_database.variables}
    \   Length Should Be    ${ELEMENT.occurences}       ${k}

Clear databases
    @{databases} =      Create List
    Set Test Variable   ${databases}

Clear variable
    ${variable} =       Evaluate     None
    Set Test Variable   ${variable}
```

Bibliography

[1] LLVM Developer Group. LLVM. https://llvm.org/, 2018. Last accessed 2019-03-07.

[2] Ian Murdock and The Debian Project. Debian. https://www.debian.org/releases/stable/, 2017. Last accessed 2019-05-07.

[3] Canonical Ltd. Ubuntu. https://releases.ubuntu.com/16.04.6/, 2016. Last accessed 2019-05-07.

[4] Jussi Pakkanen. Meson. https://mesonbuild.com/index.html, 2018. Last accessed 2019-02-25.

[5] Evan Martin. Ninja. https://ninja-build.org/, 2018. Last accessed 2019-02-25.

[6] Robot Framework Foundation. Robot Framework. https://robotframework.org/robotframework/3.1.1/RobotFrameworkUserGuide.html, 2016. Last accessed 2019-02-11.

[7] Julian Berman. jsonschema. https://github.com/Julian/jsonschema, 2019. Last accessed 2019-04-13.

[8] Andrey Kislyuk. yq. https://github.com/kislyuk/yq/, 2019. Last accessed 2019-04-14.

[9] Python Packaging Authority. The Python Package Installer. https://pip.pypa.io/en/stable/, 2017. Last accessed 2019-04-14.

[10] Python Packaging Authority. Setuptools. https://github.com/pypa/setuptools/, 2019. Last accessed 2019-04-14.

[11] LLVM Developer Group. libclang. https://releases.llvm.org/7.0.0/tools/clang/docs/Tooling.html#libclang, 2018. Last accessed 2019-04-14.

[12] LLVM Developer Group. clang-7. http://releases.llvm.org/7.0.0/tools/clang/docs/index.html, 2018. Last accessed 2019-04-14.

[13] LLVM Developer Group. LLVM. https://llvm.org/, 2018. Last accessed 2019-03-07.

[14] Boost C++ libraries. https://www.boost.org/, 2015. Last accessed 2019-04-14.

[15] Sebastian Rahtz and The TeX Users Group. Tex live. http://www.tug.org/texlive/, 2015. Last accessed 2019-04-14.

[16] Emmanuel Beffara. Rubber. https://launchpad.net/rubber/, 2015. Last accessed 2019-04-14.

[17] Hrvoje Nikšić. wget. https://www.gnu.org/software/wget/, 2015. Last accessed 2019-04-14.

[18] Michael Shuler, Raphael Geissert, Thijs Kinkhorst, Christian Perrier, and The Debian Project. Common ca certificates. https://packages.debian.org/stretch/ca-certificates, 2017. Last accessed 2019-04-14.

[19] Stephen Dolan. jq. https://stedolan.github.io/jq/, 2015. Last accessed 2019-04-14.

[20] Nicolas Kielbasiewicz. TikZ-UML. https://perso.ensta-paristech.fr/~kielbasi/tikzuml/, 2016. Last accessed 2019-04-14.

[21] Linus Torvalds and Junio C Hamano. Git. https://git-scm.com/, 2019. Last accessed 2019-05-09.

[22] Andy Cedilnik, Bill Hoffman, Brad King, Ken Martin, and Alexander Neundorf. Cmake. https://cmake.org/, 2019. Last accessed 2019-05-04.

[23] Bram Moolenaar. Vim. https://www.vim.org/, 2018. Last accessed 2019-03-07.

[24] Apache. Xerces-c++. https://xerces.apache.org/xerces-c/, 2018. Last accessed 2019-05-04.

[25] Roland McGrath and the GNU Project. Gnu c library. https://www.gnu.org/software/libc/, 2019. Last accessed 2019-05-05.