The Demacrofier

July 1, 2012

INSTALLATION AND USER MANUAL

Keywords: **source code rejuvenation; macros; C++11; refactoring; demacrofication**

1 Introduction

C++ programs can be rejuvenated by replacing error-prone usage of the C Preprocessor macros with type safe C++11 declarations. We have developed a classification of macros that directly maps to corresponding C++11 expressions, statements, and declarations. The demacrofier [6] is implemented in the (`cpp2cxx`) framework.

The demacrofication process begins takes the original source code, which contains macros, and finally generates source code in which macros have been replaced by C++11 declarations. There are three phases of translation:

1. identify the complete set of macros that can feasibly be replaced with C++11 declarations,
2. refine that set to only those transformations that produce valid builds, and
3. produce a final, working version of the program.

Currently, the ***demacrofier*** (Figure-1) is composed of three separate tools, each corresponding to one of the phases of the source code translation process. One of the tools (`cpp2cxx-suggest`) translates the macros into equivalent C++ declarations (Figure-2). The second tool (`cpp2cxx-validate`) checks the correctness of the program for each demacrofication (Figure-3). And the third tool (`cpp2cxx-finalize`) performs the cleanup and generates rejuvenated source code. In this user-manual we first describe how to use each tool (sec-2.1, sec-2.2) and finally we illustrate how to carry out the process of demacrofication with the help of a sample project 3.

In order to compile the tool, following dependencies have to be installed. The `cpp2cxx` framework depends upon the following libraries:

1. GNU C++ compiler (g++-4.6 in C++11 mode) [4],
2. boost C++ libraries (BOOST_LIB_VERSION "1_47") [1],
3. clang C++ front end (version 3.1) [2].
4. CMake (2.8) [3]
5. Python 2.7

1.1 How to build the `cpp2cxx` from source

To build the project one can choose either in-place or out of source build. It is recommended to use out of source build which is supported by the CMake build system provided with the tool's source code. The `cpp2cxx` framework contains two `CMakeLists.txt` files. The first one is in the project's root directory. It establishes the build configuration of the `cpp2cxx` framework. Before building the project, it needs minor modifications. First, the path to the directory where the clang front end has been installed has to be specified. Second,

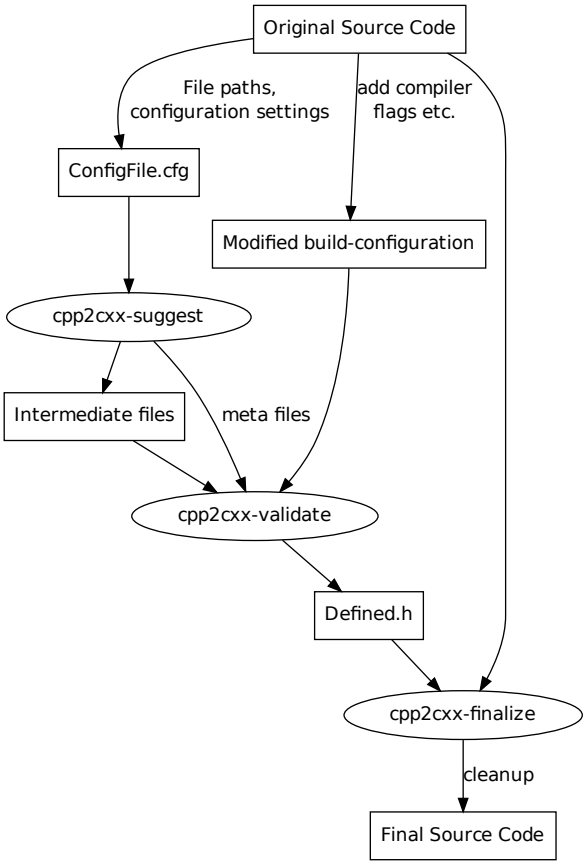

Figure 1: The demacrofier is composed of three separate tools. One of the tools (`cpp2cxx-suggest`) translates the macros into equivalent C++ declarations (Figure-2). The second tool (`cpp2cxx-validate`) checks the correctness of the program for each demacrofication (Figure-3). And the third tool (`cpp2cxx-finalize`) performs the cleanup and generates rejuvenated source code.

the path of the directory `general_utilities` has to be specified. Currently `general_utilities` is inside the project directory so no modifications are required for it, but future releases may have a different location. A typical example has been shown here.

```
SET(CFE_PREFIX_PATH "/home/Documents/llvm/install")
SET(UTILITIES_PREFIX_PATH ".")

INCLUDE_DIRECTORIES("${UTILITIES_PREFIX_PATH}/general_utilities")
INCLUDE_DIRECTORIES("${CFE_PREFIX_PATH}/include")
```

The second `CMakeLists.txt` file is inside the `ClangInterface` directory. This file defines how the interface to the clang front end would be compiled. It does not need any modification as it derives the parameters from the parent `CMakeLists.txt` file. Following sets of command (on a linux machine) will build the project.

```
#Assuming the project is in the directory \code{demacrofier}.
$cd \code{demacrofier}
$mkdir build
$cd build
$cmake ..
$make
$
```

We have not tested our implementation on non-linux platforms. It should work for non-linux systems because we use heavily ported software (boost libraries, clang front end, Python and CMake).

After the build there should be three executables (`cpp2cxx-suggest, cpp2cxx-validate.py, cpp2cxx-finalize`) in the `bin` directory relative to the directory where the demacrofier is build. The shared libraries (`libcpp2cxx-core.so` and `libASTConsumer.so`) are created in the `lib` directory. Since the file paths used in the executables as well as in the libraries are absolute paths, one can even copy the executables elsewhere (e.g. in the directory of the files to be processed) or execute it from the location where they are created. The libraries should remain in the same `lib` directory where they are created.

2 How to use the tools

2.1 How to use cpp2cxx-suggest and cpp2cxx-finalize

The `cpp2cxx-suggest` tool identifies macros that can be replaced and generates corresponding C++11 declarations. The output of this program is an intermediate version of the source code in which each transformation is guarded by a conditional directive (Figure-2).

The `cpp2cxx-finalize` tool takes original source files and list of *macro-switches* generated by the `cpp2cxx-validate` tool and generates rejuvenated C++11 programs. Currently this tool uses the same back-end as the `cpp2cxx-suggest` tool, and they have the similar user-interface, so they are described at one place.

From the user's perspective, learning how to use the tools requires getting familiar with the configuration file that has to be supplied to the tools.

The configuration file – called as `ConfigFile.cfg` – is a text file. In the `cpp2cxx-suggest` we have used the Boost.program_options [1] library to read the file `ConfigFile.cfg`. It contains the names and paths of all the files to be processed, the names and paths of all the files to be generated, and a set of configuration settings as per the project's requirements.

Once the configuration file is set up properly, running the `cpp2cxx-suggest` as easy as issuing the following command.

```
$./cpp2cxx-suggest ConfigFile.cfg
```

The config-file has the following entries:

1. list of input files: The tool can take a list of all the files that you want to process. The file name is relative to the `input-directory` which can be specified separately. For example:

   ```
   input-file = file1.h
   input-file = file1.cpp
   ```

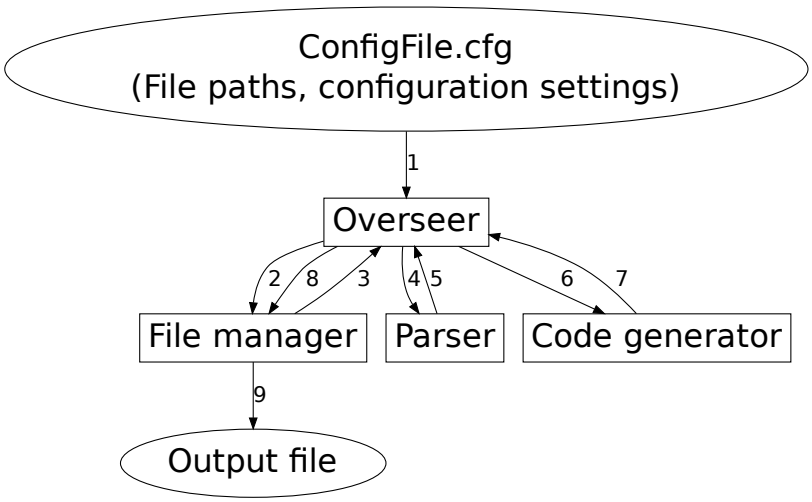

Figure 2: The `cpp2cxx-suggest` Framework. The numbers show the sequence in which each module gets involved during the translation of a file.

2. list of output files: corresponding to each input file name there has to be an output file name. So there is a one-to-one correspondence between the list of input files and the list of output files i.e. first output file listed will have the demacrofied output generated out of the first input file listed. Again the output file will be kept in the `output-directory`. So one can have same names to both the input as well as the output files as long as the codeinput-directory and the `output-directory` are different. If you want to keep both the files in the same directory please specify different file names for both.

```
output-file = file1.h
output-file = file1.cpp
```

3. File to keep information about all the macros processed (YAML [5] format).

```
macro-stat-file = macro_stat.yaml
```

4. File to keep information about all the macros refactored (YAML format).

```
stat-file = demacrofied_macro_stat.yaml
```

5. File that keeps all the global macros in a particular format. In this file odd-numbered lines contain the macro identifiers and the even numbered lines contain corresponding replacement text. Please look at the provided (`gMacros.dat`) file to clarify for yourself. The file already has all the macros predefined by gcc-4.6.3 (collected using `cpp -dM`)

```
global-macros-formatted = gMacros.dat
```

6. A log file which keeps all the error and warning messages in YAML format.

```
log-file = log.yaml
```

7. The directory from which all the input files are to be taken.

```
input-directory = .
```

8. The directory to which all the output files are to be put. The output directory has to be created if not already present. The tool does not create directories and will print the output to `std::cout` if the directories are not found.

```
output-directory = dm_dir
```

9. There are macros/identifiers which are implementation specific or compiler specific. The user can specify a list of identifiers/macros which, if referenced by a macro, should cause the refactoring to fail. For example:

```
ignore-macros = __FILE__
ignore-macros = __BASE_FILE__
ignore-macros = __LINE__
ignore-macros = __INCLUDE_LEVEL__
ignore-macros = __TIMESTAMP__
ignore-macros = __func__
ignore-macros = __FUNCTION__
ignore-macros = __declspec
ignore-macros = __attribute__
```

   The user can modify this list with their list of predefined macros depending upon the software/platform at hand.

10. The directory path to which the finally generated programs should be kept. This parameter is used by the `cpp2cxx-finalize` program. This directory has to be created by the user as the program does not create directories.

```
cleanup-directory = demac_cleanup
```

11. A flag which indicates whether you want the (helpful) warning messages to be displayed on screen or not. It is useful to enable it to `true` when processing a few files so that you may be able to read the messages. While processing a large number of files, it is good to set this flag to `false`; if you want the warning messages for all the files in a large project you can set it to `true` and *pipe* the console output (during the translation) to a file which may be read later. For *nix systems the piping of console output is easily done by issuing (`cpp2cxx-suggest ConfigFile.cfg &> error_file`). This will write all the warning messages to the ASCII file `error_file`.

```
enable-warning = false
```

12. A flag which enables the user to only collect statistical information about the macros and their usage by using the tool without even generating the intermediate files. This is a very useful feature because it helps the user to collect the useful information faster.

```
no-translate = false
```

   When this option is set to true only a `macro-stat.yaml` file will be generated in the `output-directory` which has list of all the macros analyzed by the tool.

13. A flag which might be configured to allow demacrofication of macros which are defined multiple times in a single file. It is good to set this flag to `false` unless you are completely aware that all the multiple definition of macros are properly isolated with conditionals. When this flag is set to `true` wrongly (when multiple defined macros are not properly insulated from each other), the file which has such macros would give additional compilation errors due to multiple-definitions.

```
mul-def = false
```

14. A boolean flag which suggest whether the cleanup program should generate the rejuvenated source files or not. This flag is read by the `cpp2cxx-finalize` tool.

```
cleanup = false
```

15. A list of search paths where the clang front end (CFE) will look for the included header files in the project. The `cpp2cxx-suggest` and the `cpp2cxx-finalize` tools use the clang front end (CFE) to collect useful information about the scope of a declaration and various other things. Since the CFE does not have any idea of the build-configuration of the program, the user has to provide all the search paths where it should look for the included header files.

    ```
    search-path = /usr/include
    search-path = /usr/local/include
    search-path = /home/user/Documents/MyProject/include
    ```

    To add your header search path, make a new entry of the `search-path` and provide the appropriate path. To remove a `search-path` just remove/comment that entry. Note the header search path is a directory name and not a file name.

In the `ConfigFile.cfg` there are some unused features during the release of this version of the tool. All the features will be used in future releases once all the three tools gets integrated into a unified framework.

1. The directory to which the programs under consideration should be backed up. This is an advanced feature useful when you want to validate the demacrofication as well. This feature will be implemented once the validator is integrated with the `cpp2cxx-suggest`. Since this feature has not been implemented yet so it may simply be ignored.

    ```
    backup-directory = demac_backup
    ```

2. The granularity with which the software/package needs to be validated. For faster validation one can configure the validator such that even if the validation of one macro fails the complete file is replaced with the original file. In that case the granularity would be `OneFileAtATime`, otherwise `OneMacroAtATime`. This may be ignored as of now. The default is OneMacroAtATime. This feature will be implemented once all the tools are integrated. Since this feature has not been implemented yet so it may simply be ignored.

    ```
    demacrofication-granularity = OneFileAtATime
    ```

3. Name of the file that will be generated by the validator script.

    ```
    validator-file = Defined.h
    ```

4. The build command which should be issued to build the library/software which to be demacrofied. This feature will be implemented once the validator is integrated with the `cpp2cxx-suggest`. Since this feature has not been implemented yet so it may simply be ignored.

    ```
    make-command = make
    ```

2.2 How to use cpp2cxx-validate.py

This tool transforms a software, one macro at a time by rebuilding the entire system for each transformation 3. It is written in Python [7]. It keeps the current list of macros introduced for each build, in a special header file, `Defined.h`

NOTE: Make sure the cpp2cxx-validate.py script is executable and a Python interpreter is installed.

2.3 Description

The tool does the following task:

1. Reads the intermediate file (`demacrofied_macro_stat.yaml`) having the information about demacrofied macros.

2. Replaces the original source files with the generated demacrofied files.

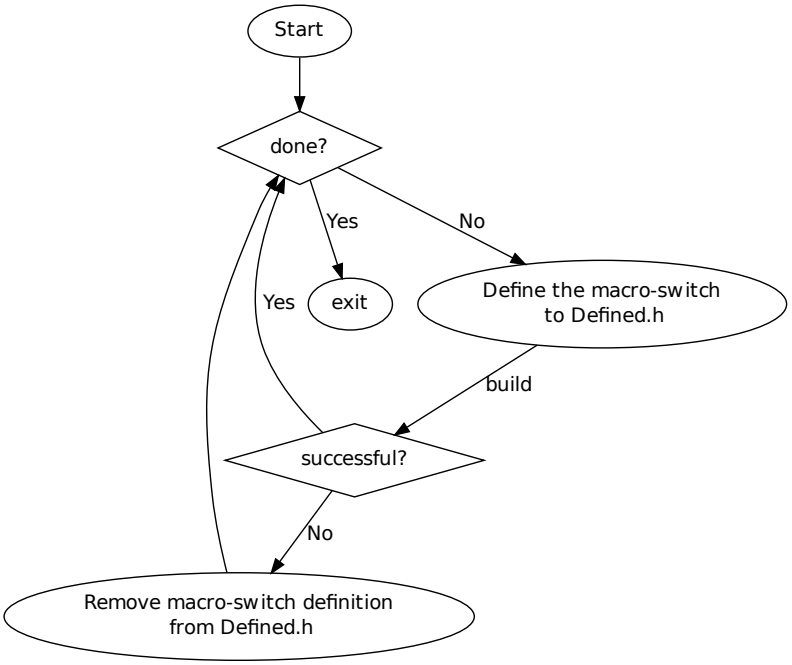

Figure 3: The cpp2cxx-validate Framework.

3. For each transformed macro:

   (a) Appends the definition of its macro switch in the header file `Defined.h`.

   (b) Builds the library.

   (c) If build fails, removes its switch from the header file `Defined.h`.

When the program finally terminates —it can take several hours depending on the number of macros, size of the library, and the computing resource available for this iterative process— the library is fully transformed and compiled.

Depending upon the software/library the user wants to rejuvenate, some of the task of this tool can be carried out manually. For example, sometimes it is easier to copy the translated (intermediate) files to their original locations and check if the library/software builds correctly after copying. The library/software should build successfully even after copying the intermediate files because the translated files have the new C++11 declarations enclosed within conditionals. After copying the user can start using the validator tool to do the iterative validation. The tool can be run in different modes (explained later) depending upon the amount of work the user wants.

2.3.1 Running the cpp2cxx-validate tool

Since the validator tool compiles and builds the software/library, using it greatly depends on the software being rejuvenated. The user is advised to have a look at the validator script `cpp2cxx-validate.py` before running it. The validator script is written in Python and has useful comments added to make the user familiar with its functionality.

By default, the tool issues a single build command (i.e., `make`) to the console; this should be modified depending upon the software and the platform. Please look at the global variable `build_command` and modify it with your own command if `make` is not the default build command for your software.

Before executing the tool, the user needs to understand the build system of the software at hand. After that, the compiler flags in the build system has to be modified to ensure that the C++ compiler compiles in

C++11 mode and it defines the macros in the header file `Defined.h` while compiling. If we take the example of gcc [4], the modification would be like:

```
CXX_FLAGS += -std=c++0x -imacros Defined.h
```

The header file (`Defined.h`) path in the compiler flag and in the validator script should point to the same file; if the Makefile and the validator script are in the same directory then there would be no modification required for this part. Appropriate modifications should be done in the compiler flags and the validator script so that they have correct paths to the header file `Defined.h`.

There are some parameters in the validator script which depend upon the software/library being rejuvenated.

1. `demac_dir`: The path to the directory `dm_dir` where the translated files are kept by the `cpp2cxx-suggest`.
2. `build_command`: string which contains the command to be issued to the console in order to build the project.
3. `backup_dir`: The path to the directory where the original source files have to be backed up.

The validator can be executed in three modes.

1. In the first mode it is capable of running the `cpp2cxx-suggest`, and then it copies the translated file from the *default* `dm_dir` directory into the project directory. After copying it would build the project iteratively by switching ON one refactoring at a time. This mode would be ideal for the rejuvenation task once the validator script has been setup properly for the project at hand. The validator can be run in this mode by passing a command line flag. For example:

   ```
   $./cpp2cxx-validate.py suggest
   ```

   During the validation phase, the validator tool writes *macro switches*, in the file `Defined.h`, which correspond to the macros which have correct refactorings.

   NOTE: A header file `Defined.h` will be created in the same directory from where the validator script would be run. Appropriate modifications should be done in the compiler flags (e.g., in `CXX_FLAGS`) and the validator script (e.g., in `defined_include_guard`) if the user wants the file to be created in a different location.

2. In the second mode, it takes from where the `cpp2cxx-suggest` left the rejuvenation process. A typical situation (with the default settings) would be like: The `cpp2cxx-suggest` completed successfully and left the translated files as well as the intermediate in the directory `dm_dir`. This mode is helpful when a single project directory contains all the (C++) programs. Nevertheless, the user should look at the `preprocess()` function in the validator script, and make appropriate modifications to run in this mode. Configuring and extending this mode is very simple for small/medium sized projects. Even the sample project is run in this mode. After that the validator can be run in the following way.

   ```
   $./cpp2cxx-validate.py no-suggest
   ```

   In this mode the validator would first build the original project. Then it would copy the translated file from the `dm_dir` to the project directory and build it once again. Both the builds should succeed because even the translated files have all the macros enabled and the new C++11 declarations are enclosed within conditionals.

   At the end of the validation phase a set of *macro switches* are written in the file `Defined.h` which correspond to the macros which have correct refactorings.

   NOTE: The header file `Defined.h` will be created in the same directory from where the validator script would be run. Appropriate modifications should be done in the compiler flags and the validator script if the user wants the file to be created in a different location.

3. In the third mode, the validator script takes from where the `cpp2cxx-suggest` left the rejuvenation process and the user has already copied the translated files from the `dm_dir` to their respective locations and also verified that the project builds after copying. This mode is helpful when the project would have several translated files all over the place in different directories (possibly out of the project directory as well) and it is easier to just copy the files from the `dm_dir` to the respective locations and build the project than write a script for the same. In this mode an empty header file `Defined.h` has to be created in the same directory from where the validator runs because in this mode, the validator starts by reading the file `Defined.h` which could have the list of all the macros that has been replaced from program.

   After that the validator can be run in the following way.

   ```
   $./cpp2cxx-validate.py resume
   ```

   This mode has additional advantage that if the validation gets terminated in between (in any mode), the user can resume the validation process just by starting the validator in this mode. Since the validator starts by reading the file `Defined.h` which could have the list of all the macros that has been replaced from program, it will not perform validation for those macro-switches already in the header file and avoid repititive validation. The user is warned not to manually insert macro-switches into the header file `Defined.h` and `resume` the validation. The header file should have only those macros which were successfully validated previously, otherwise the validator will keep getting compilation errors and no further additions shall be made to the header file.

   NOTE: Running the validator in any other mode than this one starts from scratch by removing all the entries (if any) from the file `Defined.h`.

Some libraries are written in such a way that their build system does not *re-build* with the issue of build command when only header files have been modified. If your build system is like that then you need to generate a file `include_dependency.yaml` which will contain a mapping of header files versus the files which include them in the YAML format. For example:

```
header_file1.h:
  - included_by.cpp
  - included_by2.cpp

header_file2.h:
  - included_by2.cpp
```

A C++ program (`HeaderDependencyBuild.cpp`) that would help you out, has been provided along with the demacrofier. Please read the comments in the program for relevant information. Then uncomment the following lines in the validator script to use this facility.

```
50 #file containing list of header file and dependent (source/header)files
51 #dep_file = open("dm_dir/include_dependency.yaml","r")
52 #dep_file_list = yaml.load(dep_file)
53 #dep_file.close()

104 # for f in get_dependent_file_list(file_name):
```

3 The process of demacrofication

In order to rejuvenate a library by the process of demacrofication, following a step by step method is required.

1. Download and extract the library to a location.
2. Make a backup directory and copy the files in the backup directory as well.
3. Edit the `ConfigFile.cfg` to specify list of `input-file`, list of `output-file`, `output-directory`, `cleanup-directory` etc.

4. Run the `cpp2cxx-suggest`

```
$ demacrofier/build/cpp2cxx-suggest ConfigFile.cfg
$
```

5. Edit the validator script `cpp2cxx-validate`, to specify the `build_command, demacrofied_macro_stat, cpp2cxx_suggest, backup_dir, demac_dir` and other settings if necessary.

6. Run the validator tool in one of the modes (see-2.2).

7. After the validator tool completes execution it generates a file `Defined.h` which will have all the macro-switches which corresponding to the macros that will be replaced from the library. At this time the current project directory will have the modified files. To proceed with the cleanup phase, copy the unmodified source files from the backup directory into the project directory.

8. Run the cleanup tool `cpp2cxx-finalize`. This will generate rejuvenated program files into the `cleanup` directory (as specified in the `ConfigFile.cfg`). Copy the files from the cleanup directory into their respective locations to get the demacrofied source code.

4 Common Errors and debugging suggestions

1. Assertion FileEnt && "Didn't specify a file entry to use?"

   It means the clang front end could not find a file specified. Input file path is generated by combining the input-directory as specified in the file `ConfigFile.cfg`. Make sure your input-directory combined with input-file give the correct file path.

2. multiple occurrences from option: <some-flag>

   This means the user has specified multiple options for some parameter in the config-file `ConfigFile.cfg`. Only `input-file`, `output-file`, `search-path` and `ignore-macros` can have multiple occurrences.

3. When the clang front end (CFE) is unable to find a header file (e.g., `stddef.h`).

```
/usr/include/stdio.h:736:64: error: unknown type name 'size_t'; did you mean 'ssize_t'?

usr/include/unistd.h:221:19: note: 'ssize_t' declared here
typedef __ssize_t ssize_t;
```

   Similar errors may arise if the CFE is unable to search other header files. The solution is to specify the proper search paths in the file `ConfigFile.cfg`.

   Another possibility is that you have specified the *project-specific* search-paths *before* the standard headers in the `ConfigFile.cfg`. If that is what you intend then it is fine. Otherwise, enter the project-specific search paths *after* the standard include paths to get rid of this error message.

References

[1] Boost C++ Libraries. `http://www.boost.org/`.

[2] clang: a C language family frontend for LLVM. `http://clang.llvm.org`.

[3] CMake. `http://www.cmake.org/`.

[4] GCC, the GNU Compiler Collection. `http://gcc.gnu.org`, 2012.

[5] O. Ben-Kiki, C. Evans, and B. Ingerson. YAML Ain't Markup Language (YAML^{TM}) version 1.1. *Working Draft 2008-05*, 11, 2001.

[6] A. Kumar, A. Sutton, and B. Stroustrup. Rejuvenating C++ Programs through Demacrofication. In *Software Maintenance, 2012. ICSM 2012. IEEE International Conference on*, page to appear. IEEE, 2012.

[7] J. Python. Python Programming Language. *Python (programming language) 1 CPython 13 Python Software Foundation 15*, page 1, 2009.