Guide to Clang-Chimera

Federico Iannucci
University of Naples Federico II

26th May 2016

Contents

1 Introduction

Clang-Chimera can be seen either as a framework or a library. Indeed it can be compiled and integrated in other projects, but for a rapid prototyping of *transformation operators* I suggest to use it as a framework, i.e. with an *in-source extension* approach (see Section **??**), which also allows to use the build system without any changes (but with little additions).

Indeed, Clang-Chimera provides only means to easily apply `mutation operators` (or simply operators) to source codes, e.g. enabling pre-process actions, such as macro expansion or reformat. But the main logic is implemented in `mutators`, which are grouped in `mutation operators` (see Section **??**). Clang-Chimera generates only `mutants`, i.e. mutated source codes, that are syntactically valid.

Clang-Chimera was created to be part of a mutation testing system, so it is very easy to apply local mutations across a source code. However multiple mutations are possible, but the implementation code could be a little cumbersome for newer users.

Clang-Chimera is based on Clang's LibTooling library, in particular it uses `ASTMatchers`, `MatchFinder` and `Rewriters`.

1.1 Details on Mutators and Mutation Operators

A mutation operator implements a specific set of transformations (`mutators`), while a `mutator` should represent a *general* and *reusable* transformation. Indeed a mutator expresses global transformation rules, while an operator narrows down their application on a per-function basis.

A mutator could have multiple *mutation types*, that is, as genetic mutation, given a matching rules which identifies an AST pattern (e.g a binary operation like $a + b$), it can be transformed in multiple related (that is very similar, for which the mutation code is almost the same with some small variations) variants (e.g $a - b$, $a * b$). In case of those type of transformation is useful to exploit the mutation types (using a `switch` or `if`s that probe the mutation type which is passed in input to the mutate function), for each of them it will be generated a mutant. Therefore a single mutator can generate multiple mutants, one for each match/mutation type.

Mutators can be of FOM or HOM type. A *First Order Mutation* mutator is the default and it behaves as previously written. A *High Order Mutation* is designed to *accumulate mutations* (technically while in the FOM mutator for each mutate call is passed a different `Rewriter object` and so the modification are isolated, in this case is passed always the same), so for each match the mutations are accumulated.

Mutation operator can also be of FOM or HOM type. A *First Order Mutation* operator can be seen as a simple mutators container, they can be FOM or HOM. A *Higher Order Mutation* operator is used when global mutations has to be applied and it MUST have only HOM mutators. HOM mutators of a FOM operator act on a per-function basis, so at most a mutant can have mutations

accumulated on single functions, while a HOM operator accumulates the transformations on all the functions (for which it has been configured with the funOp configuration file).

Summarizing:

- FOM Operator - It narrows down the applications of its mutators on a function:
  - FOM Mutator - For each match it creates a number of mutant equals to its mutation type (if they are all syntactically valid), because for each mutation type a local mutation is applied.
  - HOM Mutator - It cannot have a mutation type (at the moment) and for each match the mutation is accumulated, so it has associated a single mutant (but as said it is limited to the function of the operator in which it is).
- HOM Operator - All its mutators are HOM, it accumulates mutations on all the functions for which is specified, therefore it generates a single mutant.

2 Build

2.1 Requirements

Clang-Chimera requires LLVM/Clang 3.7.0, and it is compiles using their static libraries (see next section).

2.2 CMake

Clang-Chimera is in the form of a CMake project. Then to build it, it is enough to launch `cmake` (minimum version 3.3) on the CMakeLists.txt in the root directory, in case of error they are self explanatory.

The most recurrent one is relative to the impossibility of finding the LLVM and Clang libraries. To resolve it, it is necessary to pass to `cmake` the installation path of those libraries through the variable `LLVM_LIBRARY_DIR`: `cmake path/to/CMakeLists.txt -DLLVM_LIBRARY_DIR:PATH=path/to/llvm_clang_libraries`.

An example of a build starting from the root directory and supposing that the installation path of llvm libraries is `/usr/lib64/llvm`:

```
mkdir build
cd build
cmake ../ -DLLVM_LIBRARY_DIR:PATH=/usr/lib64/llvm
make
```

If the building successes, the same directory contains the `clang-chimera` executable.

2.3 Documentation

The source code is documented using the Doxygen syntax.

It is also possible to generate the doxygen documentation using the command target `doc`, which is generated by CMake. In particular, in the build directory the following instructions will build the documentation:

```
make doc
```

3 Usage

Launching `clang-chimera` without any arguments will prompt the help message. The help message contains the usage method and all the options supported.

3.1 Compilation Database

Clang-Chimera is a Clang-Tool, so it needs a compilation database. It can be passed in two ways, as a `compilation_database.json` file or manually after a double dash (–) as last arguments to Clang-Chimera. It depends on how complex are the compilation commands. The easiest way is to create a CMake project and use CMake to generate the `compilation_database.json` file, which it has to be in a parent directory from the one in which Clang-Chimera is executed or if it is in another directory, it can be specified using the `-p` or `-cd-dir` options. For more details see the compilation database related to Clang.

An example of manually passed compile commands are the following:

```
clang-chimera input.cc -- -I/path/to/includes -std=c++11
```

Those additional commands will be used to *analyse* the file `input.cc`, for example `-std=c++11` means that is written in the C++11 dialect, knowing that the Clang compiler will not warn about that.

3.2 FunOp Configuration file

A configuration file in CSV format can be passed to Clang-Chimera to narrow down the application of mutators. The format is:

```
function_name, Operator1, Operator2, ...
function_name2, Operator2, Operator4, ...
```

The reserved keyword `CHIMERA_ALL_FUNCTIONS` can be used to specify operators for all the functions, and this will override other directives. The reserved keyword `CHIMERA_ALL_OPERATORS` can be used to specify the application of all the available operators on a specific function.

```
CHIMERA_ALL_FUNCTIONS, Operator1, Operator2, ...
```

```
function_name , Operator1
function_name2 , Operator2 , Operator4
function_1 ,CHIMERA_ALL_OPERATORS
```

Use \\ to comment a line

4 Extend Clang-Chimera

This section presents a complete example of realization of a `mutation operator`, which is also the one included within the package. The mutation operator is an operator used for the mutation testing: the ROR operator (see `http://www0.cs.ucl.ac.uk/staff/m.harman/PastMScProjects2005/MaryumUmar.pdf`). The *Relational Operator Replacement* is a *method-level* (or *function-level* mutation operator) and it *replaces relational operators with other relational operators.*

Let's consider only the `>` operation, the others are very similar. A mutator that has to match the relational operation and changes it into others is needed, e.g. into ¡ and ¡= (two mutation type). We are supposing a single mutation per mutant, so the mutators are FOM (and therefore also the operator). In order to make a more complete example, let's also suppose that the mutation MUST NOT apply to operations inside the condition expression of a `for` statement.

The example exploits the build system and the main function already present in the package and it follows a bottom-up approach:

1. Create a `mutator`
2. Test a `mutator`
3. Create a `mutation operator` and register it

4.1 Create a `mutator`

The concept of `mutator` encapsulates the idea of modification of a part of the AST (AST pattern, which is ensemble of AST nodes). A mutator has:

- a match method: it is used to identify *where* to apply a modification (a *mutation*). It is the implementation of *matching rules*;
- a mutate method: it is used to actually modify the AST pattern. It is the implementation of `mutation rules`.

Being based on Clang's LibTooling, Clang-Chimera's `Mutator` class uses the means provided by it in order to ease the specification of the matching and mutation rules.

The `Mutator` class is defined in `Core/Mutator.h`, so to create a new `Mutator`, it is enough to include that header and create a class that inherits from the Mutator class.

The `Mutator` class is an abstract class, there are some pure and non-pure virtual methods to implement/override. In particular those methods implement the matching and mutation rules.

4.1.1 Matching Rules

The matching rules are implemented using two levels: a coarse and a fine grained.

The coarse grained level could be enough, it is based on the ASTMatchers. Indeed there are different types of AST matchers, when creating the new mutator is mandatory to specify which is its type, and to override the correspondent method. The method is in the form `get`*`SpecificType`*`Matcher` and it must return that specific type of matcher. So a specific `ASTMatcher` has to be implemented and it will be used to retrieve the correspondent `MatchFinder::MatchResult`, which is the type of node managed by the mutator methods.

For more details see the inner documentation.

4.1.2 Mutation Rules

Th mutation rules are implemented using the methods of a `Rewriter` object.

The implementation, which is well-commented, is in the files `include/Operators/Examples/Mutators.h` and `src/Operators/Examples/Mutators.cpp`

4.2 Test a `mutator`

Clang-Chimera has been created to be extensible. In order to easily and quickly tests new mutators, a mutator testing framework is provided. This testing framework is built upon Google Test.

Testing a mutator means to test its matching and mutation rules, indeed the effort was to find a way to quickly understand if those rules were correct and effective. While the mutation rules are difficult to automatically test, because there is not a very simple way to provide the oracle (due to problems in confronting the mutated code with the original one), the testing of the matching rules is straightforward. Its simplicity is due to the following assumption: *given a source code, each token, that can be the beginning of any syntax construct, has a unique location, which can be identified with a line and a column.* So, while it is possible to create an oracle to automatically check the matching rules, for the mutating rules they have to be manually checked, but the framework tries also to ease it.

The following are the steps to follow in order to create a test for a mutator:

1. Creation of a directory with the name equals to the identifier of the mutator to test. The directory is going to contain the test vectors and the oracle;

2. Creation of samples of C/C++ source code that will be used as test vectors, i.e. they will be analysed and mutated. Those samples **must contain functions**, whose code will be considered and they must be put in files with name `test_N.cpp`, where N starts from 0 and can assume an arbitrary value, with the constraint that if exists the Nth file must exist the $(N-1)$th file. The samples can not contain `include` directives, unless they are system headers;

3. For each `test_N` file, a `test_N_match.csv` file must be created. It has to contain a list of pair (line, column), which indicate, in an orderly manner, all the matches that should occur. For example let assume that in a file `test_0_match.csv` there is the following sequence:

   ```
   10,5
   16,3
   ```

   It means that executing the tests they should occur two matches, one at line 10, column 5 while the other at line 16, column 3.

   In order to make that work, it is important to use only spaces and **not** tabs in the sample codes.

4. Usage of the macro `CHIMERA_MUTATOR_MATCH_TEST(mutator_class_name, test_identifier)` to enable the test.

   For each Nth test, a correspondent file named `test_N_mutants.cpp` is created. Those files contain all the mutants that would be created by the mutator, in order to quickly observe the correctness of the mutation rules.

In the file `include/Testing/MutatorsTesting.h` there is the code that enables the test on the mutator we have previously created. It is recommended to insert all the test cases in that file, in order to avoid problems with redefinitions.

In `test` there is the directory `mutants`, which contains the directory with the oracle and the C/C++ samples. Indeed the argument to pass to the option `-execute-test` is `test/mutants`.

4.3 Create a `mutation operator` and register it

Differently from a mutator, a mutation operator is simply an object that is built and mutators are added to it.

In order to make Clang-Chimera aware of the new operator, it is enough to *register it* to a ChimeraTool, using the proper register function. It is registered using a smart pointer (an `::std::unique_ptr`). Indeed, typically it is used a function `get`*`OperatorName`*`Operator` which return the smart pointer.

In `include/Operators/Examples/Operators.h` and `src/Operators/Examples/Operators.cpp` there is the implementation of the ROR mutation operator. In `src/main.cpp` there is the instruction that *register* the mutation operator.

4.4 Build: Extend the build system using CMake

All the source codes are ready to be compiled. In order to allow the CMake build system of Clang-Chimera to correctly build the new operators/mutators it should be used the structure as in the example.

The new operators retrieving functions (`get`*`OperatorName`*`Operator`) must be made visible to the `src/main.cpp` file, in order to register them. In order to ease that, it is enough to include those definition in the file `include/Operators/Operators.h`.

The definitions of mutators and operators, can be spanned across multiple header files as one wishes, the important thing is to remember that **the inclusion relative path starts with** `include/`. So in our example `#include Operators/Examples/Operators.h` is the **correct** path to the file `Operators.h`.

Conversely, the implementation file should be put in a subdirectory of `src/Operators` with a `CMakeLists.txt` file, which will be automatically included by the one in parent directory. It is recommended to use the well-documented `CMakeLists.txt` file present in the `src/Operators/Examples` directory.

5 Share a Mutation Operator

Using the the CMake template to build new operators will create static libraries in the `lib/operators` directory.

So it's up to the creator to decide whether to release the source code or only the library.