The Journal of Open Source Software

MARTINI: The Little Match and Replace Tool for Automatic Code Rewriting

Alister Johnson [1¶], **Camille Coti**[2], **Allen D. Malony**[1], **and Johannes Doerfert** [3]

1 University of Oregon, Eugene, OR, USA **2** Université du Québec à Montréal, Montréal, QC, Canada **3** Argonne National Laboratory, Lemont, IL, USA ¶ Corresponding author

DOI: 10.21105/joss.04590

Software

- Review
- Repository
- Archive

Editor: Daniel S. Katz

Reviewers:

- @wimvanderbauwhede
- @tokuso2

Submitted: 06 July 2022
Published: 13 August 2022

License
Authors of papers retain copyright and release the work under a Creative Commons Attribution 4.0 International License (CC BY 4.0).

In partnership with

JOSS Special Issue for Euro-Par 2022 Artefacts
10.1007/978-3-031-12597-3_2

Summary

Rewriting code for cleanliness, API changes, and new programming models is a common, yet time-consuming task. Localized or syntax-based changes are often mechanical and can be automated with text-based tools, like Unix's sed. However, non-localized or semantic-based changes require specialized tools that usually come with complex, hard-coded rules that require expertise in compilers. This means existing techniques for code rewriting are either too simple for complex tasks or too complex to be customized by non-expert users; in either case, developers are often forced to manually update their code instead.

This work presents MARTINI, a new approach to code rewriting built on the Clang compiler, which exposes complex and semantic-driven rewrite capabilities to users in a simple and natural way. Rewrite rules are expressed as a pair of parameterized "before-and-after" code snippets in the source language, one to describe what to match and one to describe what the replacement looks like. Through this novel and user-friendly interface, programmers can automate and customize complex code changes which require a deep understanding of the language without any knowledge of compiler internals.

Statement of Need

Once rewrites span code ranges or require semantic reasoning, text-based tooling is inadequate or requires complex implementations (for example, tracking balanced parentheses with extended regular expressions). Traditionally, this is where compiler-based tooling comes in (Quinlan & Liao, 2011). The compiler's frontend has parsing and semantic analysis capabilities that allow more complete understanding of the source code and, consequently, semantic-based rewriting over most arbitrary code ranges. However, developing and customizing such tooling requires a deep understanding of the compiler and its rewriting infrastructure (if one exists), which restricts the developer pool drastically (Murai et al., 2018; Takizawa et al., 2014). Previously, as long as the number of rewrites was small and customization was not required, hard-coded rules in a compiler-based tool were sufficient. Today, however, language standards are changing more rapidly and new parallel programming models are constantly being developed.

Programmers who wish to keep their applications up-to-date must use a streamlined refactoring process (Wright et al., 2013). For instance, testing a new programming model is an intriguing and often difficult proposition; parts might be a simple matter of replacing one API with another, but most often complex changes have to be made as well, especially if the new model has any kind of parallelism. Other rewriting tasks, such as adding instrumentation or error-checking asserts, are just as time-consuming and important. These changes often follow patterns, however, and if programmers are able to capture those patterns in some way, these tasks seem like they *should* be able to be automated.

Example

As an example, consider the "simple" rewriting task done by the `clang-tidy` rule "modernize-use-nullptr". This rule replaces constants, like `NULL` and `0`, assigned to pointer variables with the C++11 `nullptr` keyword, which is both safer and more readable. Figure 1 illustrates the changes `clang-tidy` can perform. The first replacement, where a is initialized to `0`, could be done with a text-based tool, like sed, although constructing a generic regular expression to match arbitrary types and variable names could be tricky. However, the other two replacements are difficult if not impossible to handle without semantic context. The physical distance between the type of the variable and the `0`-literal can cross file boundaries, and most languages allow for various other complexities. This semantic context is out of reach for purely text-based tools. Being built on the Clang compiler, MARTINI has the context needed to emulate most of `clang-tidy`'s "modernize-use-nullptr" rule using just the three matcher-replacer pairs in Figure 2. While `clang-tidy`'s rule is over 100 lines of complex code that requires extensive knowledge of Clang internals, MARTINI can be understood by the average programmer.

```
int* test() {
 int* a = 0;
 double b, *c;
 b = 0;
 c = 0;
 return 0;
}
```

(a) Example snippet in which the `0`-literal is used for pointer and non-pointer values.

```
int* test() {
 int* a = nullptr;
 double b, *c;
 b = 0;
 c = nullptr;
 return nullptr;
}
```

(b) The same snippet with the `0`-literal replaced by `nullptr` in all pointer contexts.

Figure 1: Example to showcase the "modernize-use-nullptr" `clang-tidy` rewrite rule, which replaces `0`-literal pointers with `nullptr`.

Related Work

Most previous work in automatic code rewriting relies on compiler experts directly working with the code's abstract syntax tree (AST). ClangMR (Wright et al., 2013) and the Clang Transformer library (Clang Developers, n.d.) are similar code rewriting tools implemented in Clang, but both of these use AST matchers (with a few additions) as a user interface. MARTINI is designed to provide similar functionality but be usable by non-compiler experts.

Other code rewriting frameworks include the ROSE compiler (Quinlan & Liao, 2011), Xevolver (Takizawa et al., 2014) (built on ROSE), and the Omni source-to-source compiler (Murai et al., 2018). These tools provide more low-level interfaces than ours, and thus more precise control over rewriting, but at the cost of requiring users to be compiler experts. Neither of these are very user-friendly, as users have to directly describe AST manipulations and use syntax specific to each tool. MARTINI, on the other hand, only requires knowledge of C++ and the semantics of a few new attributes.

The most similar work to our own is Nobrainer (V. Savchenko et al., 2019; V. V. Savchenko et al., 2020), which also uses C/C++ code snippets and AST matchers to match application code and describe how to modify it, and inspired some of our user interface. As the Nobrainer project has existed for longer than ours, it supports more of the C++ standard. However, Nobrainer uses more restrictive and specialized syntax and generally enforces more restrictions on transformations than we do. They do this to ensure their transformations are (type-)safe, but we opted to take a more lenient approach. For example, it's easy to imagine cases where users may want to change the type of an expression, such as testing multiple precision arithmetic, but this is nearly impossible in Nobrainer. Nobrainer's design philosophy is to make matchers as specific as possible and force users to add generality - they assume all names in a matcher are literals unless they are specified as parameters and do their best to enforce safety.

Our philosophy is almost precisely the opposite: our matchers are as general as possible and users must add specificity, e.g., with literals, and we allow users to define any transformations they wish with minimal restrictions on safety. Nobrainer is also, sadly, not open-source.

```
template <typename T>
[[clang::matcher("nullptr-decl")]]
auto null_match() {
  [[clang::matcher_block]]
  T* var = 0;
}
```

(a) Matcher for initializing a pointer-typed variable to a 0-literal.

```
template <typename T>
[[clang::replace("nullptr-decl")]]
auto null_replace() {
  [[clang::matcher_block]]
  T* var = nullptr;
}
```

(b) Replacement using `nullptr` for the matcher in (a).

```
template <typename T>
[[clang::matcher("nullptr-asgn")]]
auto null2_match() {
  T* var = nullptr;
  [[clang::matcher_block]]
  var = 0;
}
```

(c) Matcher for a 0-literal assignment to a pointer-typed variable.

```
template <typename T>
[[clang::replace("nullptr-asgn")]]
auto null2_replace() {
  T* var = nullptr;
  [[clang::matcher_block]]
  var = nullptr;
}
```

(d) Replacement using `nullptr` for the matcher in (b).

```
template <typename T>
[[clang::matcher("nullptr-ret")]]
T* null3_match() {
  [[clang::matcher_block]]
  return 0;
}
```

(e) Matcher for a pointer-typed return statement using a 0-literal.

```
template <typename T>
[[clang::replace("nullptr-ret")]]
T* null3_replace() {
  [[clang::matcher_block]]
  return nullptr;
}
```

(f) Replacement using a `nullptr` for the matcher in (e).

Figure 2: MARTINI's equivalent to `clang-tidy`'s "modernize-use-nullptr" rule, using abbreviated MARTINI syntax.

References

Clang Developers. (n.d.). *clang::tooling::Transformer class reference*. https://clang.llvm.org/doxygen/classclang/_1/_1tooling/_1/_1Transformer.html

Murai, H., Sato, M., Nakao, M., & Lee, J. (2018). Metaprogramming framework for existing HPC languages based on the Omni compiler infrastructure. *International Symposium on Computing and Networking Workshops (CANDARW)*. https://doi.org/10.1109/CANDARW.2018.00054

Quinlan, D., & Liao, C. (2011). The ROSE source-to-source compiler infrastructure. *Cetus Users and Compiler Infrastructure Workshop @PACT*.

Savchenko, V. V., Sorokin, K. S., Bronshtein, I. E., Volkov, A. S., Kachanov, V. V., Pankratenko, G. A., Ermakov, M. K., Markov, S. I., Spiridonov, A. V., & Aleksandrov, I. V. (2020). NOBRAINER: A tool for example-based transformation of C/C++ code. *Programming and Computer Software*, *46*(5). https://doi.org/10.1134/S0361768820040052

Savchenko, V., Sorokin, K., Pankratenko, G., Markov, S., Spiridonov, A., Alexandrov, I., Volkov, A., & Sun, K. (2019). Nobrainer: An example-driven framework for C/C++

code transformations. In N. Bjørner, I. Virbitskaite, & A. Voronkov (Eds.), *Perspectives of system informatics*. Springer International Publishing. https://doi.org/10.1007/978-3-030-37487-7/_12

Takizawa, H., Hirasawa, S., Hayashi, Y., Egawa, R., & Kobayashi, H. (2014). Xevolver: An XML-based code translation framework for supporting HPC application migration. *2014 21st International Conference on High Performance Computing (HiPC)*. https://doi.org/10.1109/HiPC.2014.7116902

Wright, H. K., Jasper, D., Klimek, M., Carruth, C., & Wan, Z. (2013). Large-scale automated refactoring using ClangMR. *International Conference on Software Maintenance*. https://doi.org/10.1109/ICSM.2013.93