MUSIC: Mutation Analysis Tool with High Configurability and Extensibility

Duy Loc Phan, Yunho Kim and Moonzoo Kim
School of Computing, KAIST
South Korea
Email: duyloc_1503@kaist.ac.kr, yunho.kim03@gmail.com, moonzoo@cs.kaist.ac.kr

***Abstract*—Although mutation analysis is important for various software analysis tasks, there exist few practical mutation tools for C programs. We have developed MUSIC (MUtation analySIs tool with high Configurability and extensibility) which generates mutants for modern complex real-world C programs conveniently. MUSIC provides various mutation operators including 10 new mutation operators such as string and function call mutation operators as well as 63 conventional mutation operators. Also, MUSIC supports a user to create a new mutation operator easily. Furthermore, MUSIC can select a domain and range of a mutation operator for various purposes.**

We have applied Milu, Proteum, and MUSIC to Siemens benchmark programs and a modern real-world C program cURL, and compared them in terms of applicability and a number of uncompilable mutants generated. In the experiment, MUSIC successfully generates mutants without any uncompilable mutants.

***Index Terms*—Mutation analysis, Practical mutation tool, C programs**

1. Introduction

Mutation analysis is important for various software analysis tasks such as evaluating quality of a test suite, fault localization [1]–[3], and test generation [4], [5]. However, there exist few practical mutation tools for C programs. Existing mutation tools for C programs often fail to generate useful mutants of a modern real-world C program. For example, Proteum [6] often fail to generate mutants for a modern C program because it does not support a recent C standard later than C89. For another example, Milu [7] generates many uncompilable mutants due to incorrect handling of types including `typedef`, `enum`, `const` type qualifier and an array type.

We have developed MUSIC (MUtation analySIs tool with high Configurability and extensibility) to generate mutants for modern complex real-world C programs that consist of multiple source files with complex compilation commands. MUSIC implements 63 mutation operators defined by Agrawal et al. [8] and 10 new mutation operators for strings and function calls to generate diverse mutants. MUSIC is designed to be extensible for a user to easily make a new mutation operator.

One salient feature of MUSIC is its fine-grained configuration of mutant generation, which can satisfy various purposes of mutation analysis. In other words, MUSIC allows a user to specify a target domain and range of a mutation operator and a target scope of mutation. For example, MUSIC can apply arithmetic mutation operator (OAAN) to only one of {+,-} and mutates it to only * between Line 100 and Line 200 of `target.c`.

We have applied Milu, Proteum, and MUSIC to Siemens benchmark programs [9] and a modern large program cURL [10]. We evaluate these tools in terms of applicability and a number of uncompilable mutants generated. For both Siemens benchmarks and cURL, MUSIC successfully generates mutants without any manual modification of the target programs and it generates no uncompilable mutant. In contrast, Proteum requires manual source code modification to generate mutants for Siemens benchmarks and fails to generate mutants for cURL. Milu generates many uncompilable mutants (34.18% and 75.31% of the mutants for Siemens benchmarks and cURL were uncompilable, respectively).

Our contributions are as follows:

1) We have developed MUSIC which is highly configurable and easy to extend for various mutation analysis purposes targeting modern complex real-world C programs.
2) MUSIC provides an extensive set of 73 mutation operators including 63 existing operators defined in Agrawal et al. [8] and 10 new operators which do not generate uncompilable mutants.
3) We have performed an experiment to evaluate Milu, Proteum and MUSIC on Siemens benchmarks and a modern real-world C program cURL and demonstrated that MUSIC has high applicability and generates no uncompilable mutant.

Sect. 2 explains MUSIC. Sect. 3 reports the experiment on Siemens benchmarks and cURL. Sect. 4 concludes this paper with future work.

2. MUSIC Technique

We have implemented MUSIC based on the modern compiler framework Clang/LLVM 4.0 [11]. MUSIC is written in 9,000 lines of C++ code, which consists of 178 header and source files, and 88 classes. MUSIC will be available at https://github.com/swtv-kaist/MUSIC.

Sect. 2.1 describes how a user applies MUSIC to a large complex project conveniently. Sect. 2.2 explains high configurability of MUSIC. Sect. 2.3 shows that a user can create his/her own new mutation operators easily with support of MUSIC. Sect. 2.4 explains how MUSIC avoids generating uncompilable mutants. The component architecture of MUSIC referred through the subsections is shown in Fig. 1.

2.1. Applicability

To generate mutants from a large complex project consisting of many directories and files with file-specific compilation commands, MUSIC utilizes a *compilation database*. This is because mutant generation often depends on specific compilation commands. For example, `util.c` in the left code of Fig. 2 illustrates such situation. Without compilation information, a mutation tool assumes that flag `UTIL` is *not* defined and generates an AST of the code as shown in the right part in Fig. 2, which fails to generate mutants on Line 4 even if a user actually compiles `util.c` with flag `UTIL` as true. MUSIC can utilize all such compilation information from a given compilation database and a user can apply MUSIC to complex large projects conveniently [1].

A compilation database is a collection of compilation commands for a set of files. MUSIC receives compilation database in a JSON format. Each entry in a compilation database has three fields:

- a file to which the compilation applies to
- a compilation command used
- a directory in which this command is executed

2.2. Configurability

MUSIC provides mainly four options to selectively generate mutants for multiple C source files as follows:

1) `-m` $mut\_op[:A[:B]]$: to select a mutation operator to apply (e.g., OAAN) and, optionally, a set of target token(s) to replace (e.g., A can be {+,*}) and a set of new token(s) to use (e.g., B can be {-,/}).

   mut_op can be one of the 73 pre-defined mutation operators as follows:

   - 63 mutation operators defined in Agrawal et al. [8]
   - three new string literal mutation operators [2]
   - seven new function call mutation operators [3]

   For example, OAAN mutates arithmetic operators (+, -, *, /, %) to other arithmetic operators. A user can specify a target domain of OAAN as $\{+\}$ and a target range as $\{*, /\}$ as shown in Fig. 3. Note that specified domain and range of `-m` must be *type-compatible* to a mutation operator (e.g., for OAAN, target domain and range cannot contain $<$ or $>>$). [4]
2) `-rs` mut_range_start: to specify a starting position of a target mutation range (i.e., a triple of a target file name, line number, column number)
3) `-re` mut_range_end: to specify an ending position of a target mutation range
4) `-l` max_num: to limit a maximum number of mutants generated per mutation point and mutation operator

   Some mutation operators may generate many mutants. For example, CCCR mutates constant literals to another constant literals in a target program, which can generate many mutants. If `-l` 10 is given, for each mutation location of CCCR, MUSIC arbitrarily generates at most 10 mutants of CCCR.

2.3. Extensibility

One advantage of MUSIC is that it supports a user to create his/her own *new* mutation operators conveniently. A mutation operator of MUSIC is defined as a rule to modify a target source file. Such rule specifies a target domain and range of a mutation operator as a set of tokens such that a mutation operator replaces tokens in a target domain with ones in a target range.

A mutation operator of MUSIC extends one of `ExprMutantOperator` which mutates C expressions and `StmtMutantOperator` which mutates C statements. These two classes extend an abstract class `MutantOperatorTemplate` which has a mutation operator name, its domain and range, and four utility functions (two for validating domain and range and two setter functions for domain and range). Also each mutation operator implements the following two core functions:

- `IsMutationTarget` function to check whether a current statement/expression should be mutated
- `Mutate` function to actually apply mutation

1. A user can easily generate a compilation database for a project by running CMake with the `-DCMAKE_EXPORT_COMPILE_COMMANDS` flag. Also, Gyp/Ninja or BEAR can be used for the purpose.

2. SRWS (String Remove White Space), SANL (String Add New Line) SCSR (String Constant for String constant Replacement)

3. FGSR (Function call returning scalar for Global Scalar variable Replacement), FLSR (Function call returning scalar for Local Scalar variable Replacement), FGTR (Function call returning structure for Global Structure variable Replacement), FLTR (Function call returning structure for Local Structure variable Replacement), FGPR (Function call returning pointer for Global Pointer variable Replacement), FLPR (Function call returning pointer for Local Pointer variable Replacement), FTWD (Function Call Twiddle Mutations)

4. For some mutation operators, MUSIC provides pre-defined values that a user can use to specify a target domain and range conveniently. For example -m CCCR:{1,2}:{MEDIAN,MAX} where MEDIAN is a median value and MAX is a maximum value among constants in a target program.

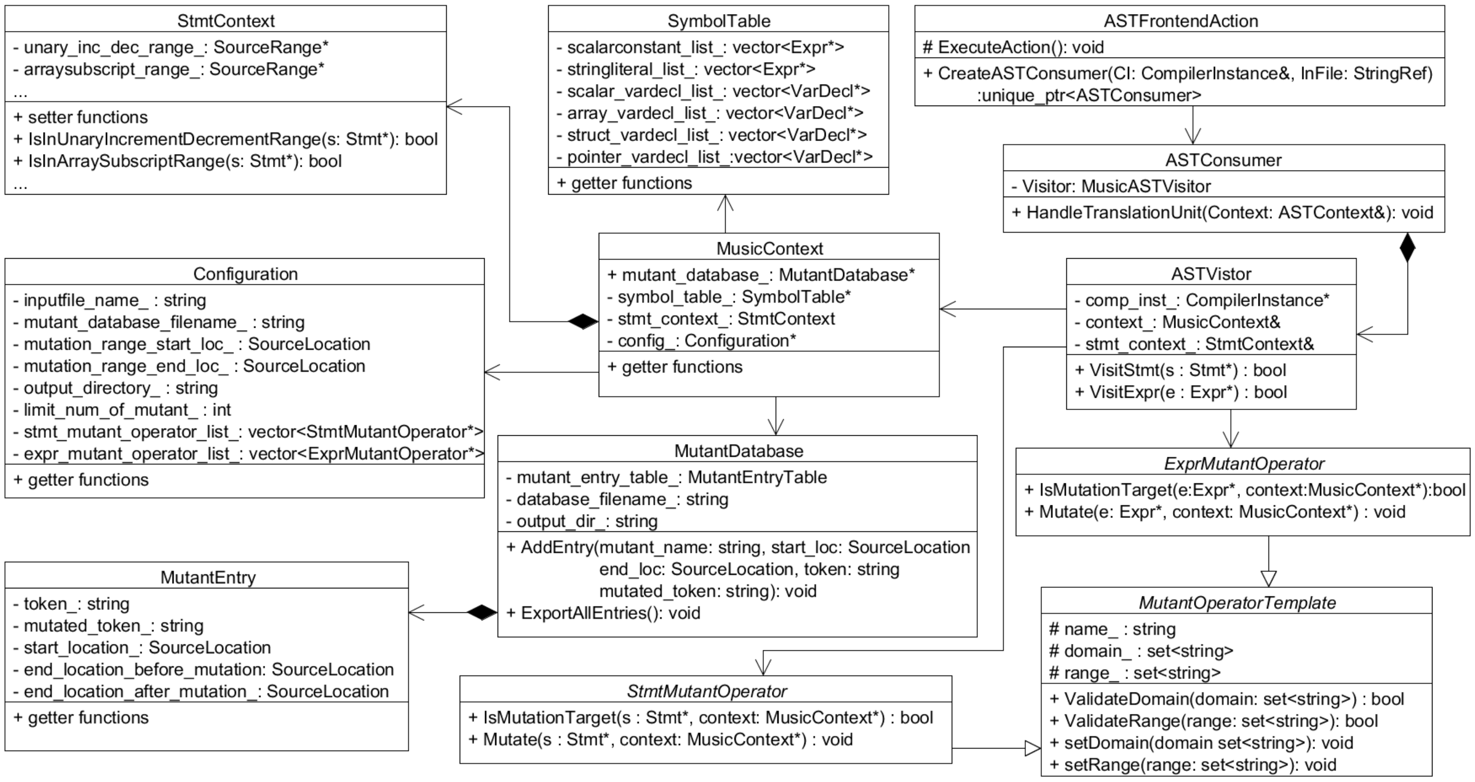
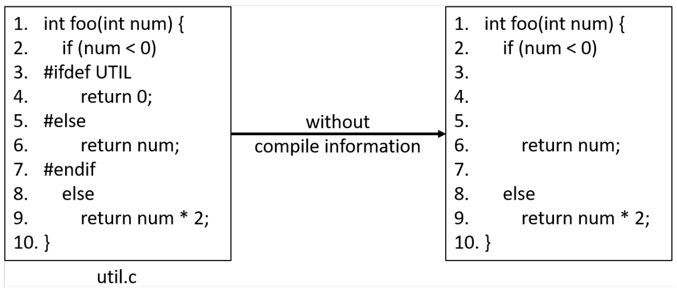

StmtContext
- unary_inc_dec_range_: SourceRange*
- arraysubscript_range_: SourceRange*
...
+ setter functions
+ IsInUnaryIncrementDecrementRange(s: Stmt*): bool
+ IsInArraySubscriptRange(s: Stmt*): bool
...
SymbolTable
- scalarconstant_list_: vector<Expr*>
- stringliteral_list_: vector<Expr*>
- scalar_vardecl_list_: vector<VarDecl*>
- array_vardecl_list_: vector<VarDecl*>
- struct_vardecl_list_: vector<VarDecl*>
- pointer_vardecl_list_:vector<VarDecl*>
+ getter functions
ASTFrontendAction
# ExecuteAction(): void
+ CreateASTConsumer(CI: CompilerInstance&, InFile: StringRef)
:unique_ptr<ASTConsumer>
ASTConsumer
- Visitor: MusicASTVisitor
+ HandleTranslationUnit(Context: ASTContext&): void
Configuration
- inputfile_name_ : string
- mutant_database_filename_ : string
- mutation_range_start_loc_ : SourceLocation
- mutation_range_end_loc_ : SourceLocation
- output_directory_ : string
- limit_num_of_mutant_ : int
- stmt_mutant_operator_list_: vector<StmtMutantOperator*>
- expr_mutant_operator_list_: vector<ExprMutantOperator*>
+ getter functions
MusicContext
+ mutant_database_: MutantDatabase*
- symbol_table_: SymbolTable*
- stmt_context_: StmtContext
- config_: Configuration*
+ getter functions
ASTVistor
- comp_inst_: CompilerInstance*
- context_: MusicContext&
- stmt_context_: StmtContext&
+ VisitStmt(s : Stmt*) : bool
+ VisitExpr(e : Expr*) : bool
MutantDatabase
- mutant_entry_table_: MutantEntryTable
- database_filename_: string
- output_dir_: string
+ AddEntry(mutant_name: string, start_loc: SourceLocation
end_loc: SourceLocation, token: string
mutated_token: string): void
+ ExportAllEntries(): void
ExprMutantOperator
+ IsMutationTarget(e:Expr*, context:MusicContext*):bool
+ Mutate(e: Expr*, context: MusicContext*) : void
MutantEntry
- token_: string
- mutated_token_: string
- start_location_: SourceLocation
- end_location_before_mutation: SourceLocation
- end_location_after_mutation_: SourceLocation
+ getter functions
MutantOperatorTemplate
# name_ : string
# domain_ : set<string>
# range_ : set<string>
+ ValidateDomain(domain: set<string>) : bool
+ ValidateRange(range: set<string>): bool
+ setDomain(domain set<string>): void
+ setRange(range: set<string>): void
StmtMutantOperator
+ IsMutationTarget(s : Stmt*, context: MusicContext*) : bool
+ Mutate(s : Stmt*, context: MusicContext*) : void

Figure 1. Simplified UML diagram of MUSIC


```
int foo(int num) {
   if (num < 0)
#ifdef UTIL
      return 0;
#else
      return num;
#endif
   else
      return num * 2;
}
```

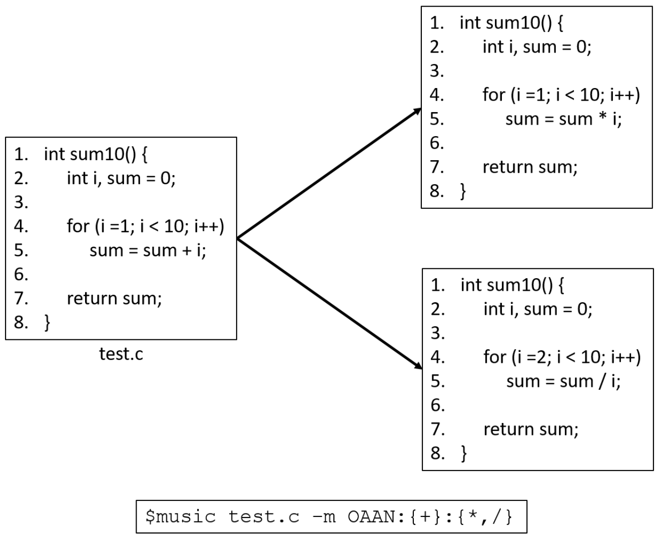
```
int foo(int num) {
   if (num < 0)



      return num;

   else
      return num * 2;
}
```

Figure 2. Example of code without proper compilation information


```
int sum10() {
   int i, sum = 0;

   for (i =1; i < 10; i++)
      sum = sum + i;

   return sum;
}
```

```
int sum10() {
   int i, sum = 0;

   for (i =1; i < 10; i++)
      sum = sum * i;

   return sum;
}
```
```
int sum10() {
   int i, sum = 0;

   for (i =2; i < 10; i++)
      sum = sum / i;

   return sum;
}
```
```
$music test.c -m OAAN:{+}:{*,/}
```

Figure 3. Usage of option -m to mutate + to * and / by modifying OAAN domain and range

For example, suppose that we would like to make a new mutation operator SANL (String mutation operator to Add New Line character) which mutates `StringLiteral` expressions by adding a newline character '`\n`' at the end of a target string. The domain of SANL is a set of strings in a target source file to mutate. Fig. 4 shows how function `SANL::IsMutationTarget` is defined.

For `Mutate` function, a goal is to add a new `MutantEntry` to `MutantDatabase`. `MutantEntry` contains a mutation operator name, start and end locations of target statement/expression, target token(s) to mutate and new tokens to replace target token(s). Figure 5 shows how function `SANL::Mutate` can be implemented.

In addition, MUSIC provides several utility classes to help a user build his/her own mutation operators conveniently. For example, suppose that a user wants to make a new mutation operator SCSR which mutates a string to another string in a target program. MUSIC provides `SymbolTable` class that con-

```
bool
IsMutationTarget(Expr *e, ...) {
  if (!isa<StringLiteral>(e))
    return false;

  if (!user_given_domain_.empty()) {
    return user_given_domain_.find(
        ConvertToString(e));
  }
  else true; // all strings are targeted
}
```

Figure 4. IsMutationTarget function for SANL

```
void
Mutate(Expr *e, MusicContext *context) {
  CompilerInstance *CI = \
      context->comp_inst_;
  SourceLocation start_loc = \
      e->getLocStart();
  SourceLocation end_loc = \
      GetEndLocOfExpr(e, CI);

  string token = ConvertToString(e);
  string new_token = \
      token.substr(token.size()-1)+"\\n\"";

  context->mutant_database_.AddMutantEntry(
      mutation_op_name_, start_loc, end_loc,
      token, new_token);
}
```

Figure 5. Example of making a new mutation operator SANL

tains categorized lists of statements/expressions of an entire target source file. For SCSR, a user can utilize `SymbolTable::stringliteral_list_` which is a list of `StringLiteral` expressions in the target source code file (i.e., a user can obtain strings to replace a target string by calling `ConvertToString` on each element of `stringliteral_list_`).

2.4. Mutant Compilability

MUSIC does not generate uncompilable mutants by utilizing type information. First, MUSIC avoids uncompilable mutant generation by utilizing type information of operands of target C operators. For example, Fig. 6 shows how MUSIC prohibits generating uncompilable mutants. Applying OAAN to mutate + to % on Line 5 will generate an uncompilable mutant because % should take only integer operands but the second operand of % (i.e., `f`) is a floating number. MUSIC prevents this mutation by analyzing types of operands in a target expression (i.e., `arr[1]+f`).

Second, MUSIC avoids uncompilable mutant generation by utilizing type information of target variables (including contexts of target variables which are stored in `StmtContext` class). For example, while parsing an expression containing a target variable `a` with unary increment (i.e., `a++`), decrement (i.e., `a--`), address-of (i.e., `&a`) or

```
 1: int foo() {
 2:   float f = -1.0; int a = 1; int arr[2];
 3:   arr[0] = a;
 4:   scanf("%d", &arr[a]);
 5:   int sum = arr[1] + f;
 6:
 7:   if (arr[1] < 0) {
 8:     done:
 9:     return (int) f;
10:   }
11:
12:   if (sum < 0)
13:     goto done;
14:
15:   return sum;
16: }
```

Figure 6. Example source code for avoiding uncompilable mutants

dereference operator (i.e., `*a`), MUSIC does not mutate a target variable `a` to a constant. For another example, if VSRR (Scalar Variable Replacement) mutates an integer variable `a` (used as an index of an array) to a floating variable `f` at Line 4 of Fig. 6, the generated mutant will not be compilable because an array index cannot be a non-integer type. Thus, MUSIC does not mutate `a` to `f`. .

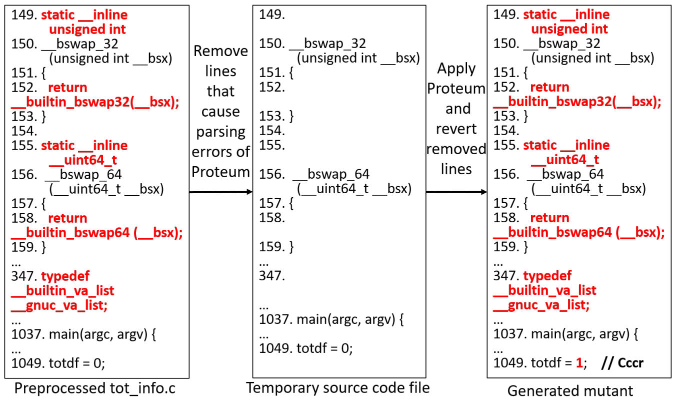
Third, MUSIC utilizes information about `goto`, `switch` statements to prevent uncompilable mutants violating C syntax. For example, SSDL (Statement Deletion) should not be applied to if-statement on Line 7 of Fig. 6 because removal of Line 8 will cause a compile error due to missing target label statement of `goto` at Line 13. For `switch` statements, MUSIC checks all `case` labels' values to prevent uncompilable mutants caused by the duplicated case label error.

3. Case Studies: Siemens Benchmarks and cURL

We evaluate applicability and efficiency (i.e., a number of uncompilable mutants generated) of MUSIC by applying MUSIC to the seven Siemens benchmark programs in Subject Infrastructure Repository (SIR) [9] and a large real-world modern C program cURL [10]. Also, we compare MUSIC with Milu and Proteum which are popular mutation tools for C programs.

We target Siemens benchmark programs because they have various C language constructs including integer and floating-point arithmetic, `struct`s, pointers, memory allocations, loops, `switch` statements and complex conditional expressions. For this reason, these programs have been widely studied in testing and debugging literature. Siemens benchmark programs are 312.6 LoC long on average (see the second column of Table 1).

cURL is a command line tool and library for transferring data through various network protocols including HTTP, FTP, IMAP, etc. We choose cURL because cURL is a very popular open-source project which has 6,700 stars in GitHub. To maintain mutant generation time reasonably, we



```
149. static __inline
       unsigned int
150. __bswap_32
       (unsigned int __bsx)
151. {
152.   return
__builtin_bswap32(__bsx);
153. }
154.
155. static __inline
       __uint64_t
156.  __bswap_64
       (__uint64_t __bsx)
157. {
158.   return
__builtin_bswap64 (__bsx);
159. }
...
347. typedef
__builtin_va_list
__gnuc_va_list;
...
1037. main(argc, argv) {
...
1049. totdf = 0;
```

```
149.

150. __bswap_32
       (unsigned int __bsx)
151. {
152.

153. }
154.
155.

156.  __bswap_64
       (__uint64_t __bsx)
157. {
158.

159. }
...
347.


...
1037. main(argc, argv) {
...
1049. totdf = 0;
```

```
149. static __inline
       unsigned int
150. __bswap_32
       (unsigned int __bsx)
151. {
152.   return
__builtin_bswap32(__bsx);
153. }
154.
155. static __inline
       __uint64_t
156.  __bswap_64
       (__uint64_t __bsx)
157. {
158.   return
__builtin_bswap64 (__bsx);
159. }
...
347. typedef
__builtin_va_list
__gnuc_va_list;
...
1037. main(argc, argv) {
...
1049. totdf = 1;    // Cccr
```

Remove lines that cause parsing errors of Proteum
Apply Proteum and revert removed lines
Preprocessed tot_info.c
Temporary source code file
Generated mutant

Figure 7. Source code modification of tot_info to apply Proteum

build cURL to support only HTTP protocol and perform mutant generation on only the cURL command line tool, not library. cURL with only HTTP protocol support is 12753 LoC long.

3.1. Applicability

MUSIC clearly shows better applicability than Milu and Proteum. We could apply MUSIC to cURL easily because it takes multiple preprocessed or unpreprocessed C files as input using compilation database (Sect. 2.1). In contrast, Milu and Proteum take only a *single preprocessed* C source file as input. In other words, to apply Milu and Proteum, a user has to manually handle complex compilation information (including macro definitions , header files in separate directories, and so on) for each C file one by one, which causes significant manual overhead for large projects.

Moreover, Proteum often fails to generate mutants for C programs compatible with recent C99 or C11 standards (for example, the system header files of Siemens benchmarks are compatible with C99 standards). A manual workaround for this problem is as follows:

1) A user identifies statement(s) s_f of a target program that make Proteum fail.
2) A user modifies s_f to s' so that Proteum can process a target program without failure.
3) A user generates mutants and then revert s' of every mutant to s_f.

We had to modify five lines for each of `printtokens`, `printtokens2`, `totinfo`, and one line for each of the remaining four Siemens benchmark programs. As an example, Fig. 7 shows how we modify the preprocessed source file of `totinfo`. The five lines colored with red in the leftmost box cause parsing errors of Proteum due to inline keyword (Lines 149 and 155), built-in functions (Lines 152 and 158), and built-in type (Line 347). We generate a temporary source file by removing the problematic lines (see the middle box of Fig. 7) and apply Proteum to the temporary file. After Proteum generates mutants, we revert the removed lines in each of the mutants (see the right box of Fig. 7).

For cURL, Proteum fails to generate a mutant even after we have modified more than 20 lines in the preprocessed `tool_main.c` source file which contains `main()` of cURL. For Milu, we had to manually generate 28 preprocessed source files and apply Milu to each of the preprocessed source files separately. Therefore, MUSIC shows higher applicability to a large real-world modern C program such as cURL than Milu and Proteum.

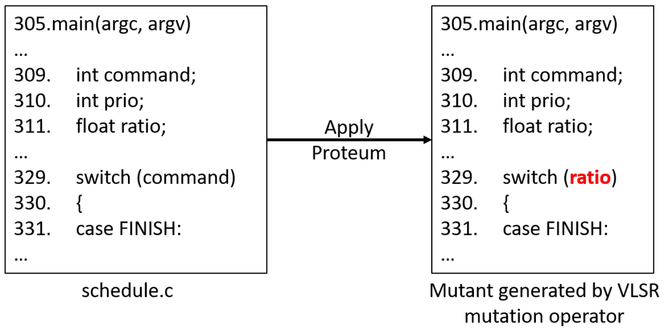
TABLE 1. NUMBER OF MUTANTS GENERATED BY MILU, PROTEUM AND MUSIC ON SIEMENS C BENCHMARK PROGRAMS

Target Program	LOC	Mutation Tool	#Gen. Mutants	#Uncomp. Mutants	%Uncomp. Mutants
printtokens	343	Milu	3077	995	32.34
		Proteum	4273	200	4.68
		MUSIC	11274	0	0.00
printtokens2	355	Milu	2424	523	21.58
		Proteum	4680	162	3.46
		MUSIC	4791	0	0.00
replace	513	Milu	3927	353	8.99
		Proteum	10872	509	4.68
		MUSIC	9925	0	0.00
schedule	296	Milu	1310	640	48.85
		Proteum	2241	103	4.60
		MUSIC	2365	0	0.00
schedule2	263	Milu	1919	915	47.68
		Proteum	2950	114	3.86
		MUSIC	3033	0	0.00
tcas	137	Milu	874	271	31.01
		Proteum	2872	74	2.58
		MUSIC	3415	0	0.00
totinfo	281	Milu	2381	996	41.83
		Proteum	6390	122	1.91
		MUSIC	10486	0	0.00
Average	**312.6**	**Milu**	**2273.1**	**670.4**	**33.18**
		Proteum	**4896.9**	**183.4**	**3.68**
		MUSIC	**6469.9**	**0**	**0.00**

3.2. Efficiency

3.2.1. Total Number of Mutants Generated. The forth column of Table 1 shows numbers of mutants generated by Milu, Proteum and MUSIC. Milu supports only 28 mutation operators so it usually generates much less mutants than Proteum (78 mutation operators) and MUSIC (73 mutation operators). As a result, on average, Milu, Proteum, and MUSIC generate 2273.1, 4896.9 and 6469.9 mutants on Siemens benchmark programs, respectively.

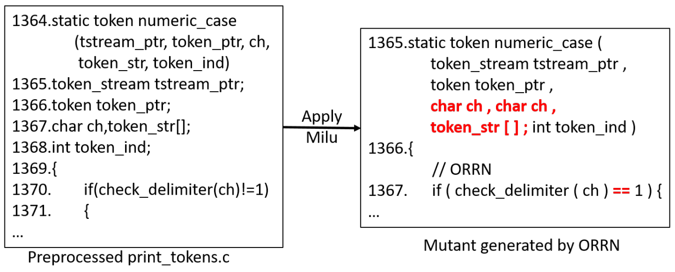
For Siemens benchmark programs, MUSIC generates 1573.0 more mutants than Proteum, on average. The difference is mainly caused by the way CCCR (Constant for Constant Replacement) and CCSR (Constant for Scalar Replacement) are implemented in each tool. MUSIC's CCCR and CCSR mutate a constant and a variable to a constant in a target program including a global constant as defined in Agrawal [8], respectively. However, Proteum's CCCR and CCSR do *not* mutate a constant or a variable to a global constant.



```
305.main(argc, argv)
...
309.    int command;
310.    int prio;
311.    float ratio;
...
329.    switch (command)
330.    {
331.    case FINISH:
...
```

```
305.main(argc, argv)
...
309.    int command;
310.    int prio;
311.    float ratio;
...
329.    switch (ratio)
330.    {
331.    case FINISH:
...
```

Figure 8. Example of uncompilable mutants generated by Proteum



```
1364.static token numeric_case
          (tstream_ptr, token_ptr, ch,
           token_str, token_ind)
1365.token_stream tstream_ptr;
1366.token token_ptr;
1367.char ch,token_str[];
1368.int token_ind;
1369.{
1370.     if(check_delimiter(ch)!=1)
1371.     {
...
```

```
1365.static token numeric_case (
          token_stream tstream_ptr ,
          token token_ptr ,
          char ch , char ch ,
          token_str [ ] ; int token_ind )
1366.{
          // ORRN
1367.     if ( check_delimiter ( ch ) == 1 ) {
...
```

Figure 9. Example of uncompilable mutants generated by Milu

For cURL, Milu and MUSIC generates 41470 and 1035088 mutants respectively.

3.2.2. Number of Uncompilable Mutants. The fifth column of Table 1 shows that MUSIC generates no uncompilable mutants for Siemens benchmark programs. In contrast, 33.18% and 3.68% of all mutants generated by Milu and Proteum are uncompilable, respectively (the sixth column of Table 1).

An example of an uncompilable mutant generated by Proteum is shown in Fig. 8. Proteum's VLSR (Local Scalar Variable Replacement) mutates a condition variable `command` on Line 329 to a floating variable `ratio`. Since switch statement cannot take a floating variable, the generated mutant is uncompilable.

Fig. 9 shows an example of an uncompilable mutant generated by Milu. When applying Milu to mutate function `numeric_case` in `printtokens`, all 170 generated mutants were uncompilable due to the syntax errors occurred in function definition: redefinition of parameter `ch` and inclusion of semicolon in list of function parameters.

For cURL, MUSIC generates no uncompilable mutant while Milu generates 31232 ones (i.e., 75.31% of the generated mutants are uncompilable). We found that this large number of uncompilable mutants is caused by incorrect handling of types including `typedef`, `enum`, `const` type qualifier and an array type.

4. Conclusion and Future Work

We have presented MUSIC which can generate diverse mutants for various mutation analysis purposes on modern real-world C programs. Through the experiments on Siemens benchmark programs and cURL, we have demonstrated that MUSIC has high applicability and generates no uncompilable mutant.

As future work, we will apply MUSIC to more large C projects to evaluate its applicability further. Also, since mutation analysis is actively used for various software analysis tasks, we plan to provide more diverse mutation operators in MUSIC. MUSIC will be publicly available at https://github.com/swtv-kaist/MUSIC

References

[1] M. Papadakis and Y. L. Traon, "Metallaxis-fl: mutation-based fault localization," *Softw. Test., Verif. Reliab.*, vol. 25, no. 5-7, pp. 605–628, 2015.

[2] S. Moon, Y. Kim, M. Kim, and S. Yoo, "Ask the mutants: Mutating faulty programs for fault localization," in *ICST*, 2014.

[3] S. Hong, B. Lee, T. Kwak, Y. Jeon, B. Ko, Y. Kim, and M. Kim, "Mutation-based fault localization for real-world multilingual programs," in *ASE*, 2015, pp. 464–475.

[4] Y. Kim, S. Hong, B. Ko, D. Phan, and M. Kim, "Destructive software testing: Mutating a target program to diversify test exploration for high test coverage," in *ICST*, 2018.

[5] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," *IEEE Transactions on Software Engineering*, vol. 38, pp. 278–292, 2011.

[6] J. C. Maldonado, M. E. Delamaro, S. C. P. F. Fabbri, A. da Silva Simão, T. Sugeta, A. M. R. Vincenzi, and P. C. Masiero, *Proteum: A Family of Tools to Support Specification and Program Testing Based on Mutation*. Springer US, 2001, pp. 113–116.

[7] Y. Jia and M. Harman, "Milu : A customizable, runtime-optimized higher order mutation testing tool for the full c language," in *TAIC-PART*, 2008, pp. 94–98.

[8] H.Agrawal, R.A.DeMillo, B.Hathaway, W.Hsu, W.Hsu, E.W.Krauser, R.J.Martin, A.P.Mathur, and E.Spafford, "Design of mutant operators for the c programming language," Purdue University, Tech. Rep. SERC-TR-120-P, 1989.

[9] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Software Engineering*, vol. 10, no. 4, pp. 405–435, Oct. 2005.

[10] "cURL: A command line tool and library for transferring data with URLs," https://curl.haxx.se/.

[11] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *CGO*, ser. CGO '04, 2004, pp. 75–.