An Extensible, Regular-Expression-Based Tool for Multi-Language Mutant Generation

Alex Groce
School of Informatics, Computing, and Cyber Systems, Northern Arizona University, USA
agroce@gmail.com

Josie Holmes
School of Informatics, Computing, and Cyber Systems, Northern Arizona University, USA
Josie.Holmes@nau.edu

Darko Marinov
Department of Computer Science, University of Illinois at Urbana-Champaign, USA
marinov@illinois.edu

August Shi
Department of Computer Science, University of Illinois at Urbana-Champaign, USA
awshi2@illinois.edu

Lingming Zhang
Department of Computer Science, The University of Texas at Dallas, USA
lingming.zhang@utdallas.edu

ABSTRACT

Mutation testing is widely used in research (even if not in practice). Mutation testing tools usually target only one programming language and rely on parsing a program to generate mutants, or operate not at the source level but on compiled bytecode. Unfortunately, developing a robust mutation testing tool for a new language in this paradigm is a difficult and time-consuming undertaking. Moreover, bytecode/intermediate language mutants are difficult for programmers to read and understand. This paper presents a simple tool, called `universalmutator`, based on regular-expression-defined transformations of source code. The primary drawback of such an approach is that our tool can generate invalid mutants that do not compile, and sometimes fails to generate mutants that a parser-based tool would have produced. Additionally, it is incompatible with some approaches to improving the efficiency of mutation testing. However, the regexp-based approach provides multiple compensating advantages. First, our tool is easy to adapt to new languages; e.g., we present here the first mutation tool for Apple's Swift programming language. Second, the method makes handling multi-language programs and systems simple, because the same tool can support every language. Finally, our approach makes it easy for users to add custom, project-specific mutations.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**;

KEYWORDS

mutation testing, multi-language tools, regular expressions

ICSE '18 Companion, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5663-3/18/05.
https://doi.org/10.1145/3183440.3183485

ACM Reference Format:
Alex Groce, Josie Holmes, Darko Marinov, August Shi, and Lingming Zhang . 2018. An Extensible, Regular-Expression-Based Tool for Multi-Language Mutant Generation. In *ICSE '18 Companion: 40th International Conference on Software Engineering , May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3183440.3183485

1 INTRODUCTION

Mutation testing [7, 25] is a widely used technique, originally proposed as a way for test engineers to understand omissions in their testing, and now routinely used in research, e.g., to evaluate novel software testing methods [2, 4, 8] and to improve software fault localization [20, 23, 27].

Unfortunately, despite the considerable utility of mutation testing, effective mutation testing tools are surprisingly rare. Java alone has widely available, easily used mutation tools, but the most popular and useful of these, PIT [6], operates only at the bytecode level, making it difficult for users to read and understand the generated mutants. While there are multiple tools for C mutation, they often crash on surprisingly simple programs, and both Milu (https://github.com/yuejia/Milu) and Proteum (https://github.com/magsilva/proteum) have not been updated since mid-2016. In fact, the simple Prolog-based tool developed by Andrews [5] is still widely used in research and practice, despite not even working with recent versions of the Prolog compiler.

The `universalmutator`[1] is a tool for mutation testing that aims to expand the applicability of mutation testing by making it easy to extend effective mutant generation to new programming languages. The tool's design is informed by four basic principles:

(1) In practice, most source-code mutants of interest can be produced without actually parsing the language of a source code file. Most useful mutations are approximately equivalent to a set of string manipulations that can be defined by a regular expression to match and a replacement string. This makes it possible to avoid the onerous work of building a parser for every language to be mutated.
(2) Modern optimizing compilers are better at detecting invalid and (more importantly) equivalent and redundant mutants

[1] https://github.com/agroce/universalmutator

than any tool that can be produced with a reasonable amount of effort. Reducing the set of mutants to be tested to useful mutants is most easily approached by leveraging Trivial Compiler Equivalence [16, 26].

(3) Because of the theoretical and practical limits of mutation reduction strategies [10, 12], the effectiveness of random sampling [11], and the need for novel mutants to correlate to important classes of software defect [9, 17, 21], it is essential for a mutation tool to make adding new kinds of mutations easy, and to support custom, project-specific mutants defined by developers expert in their own code base.

(4) Mutants are used in a wide variety of contexts; while simplicity limits the efficiency of our tool, we aim to make it easy to integrate the tool into a variety of software testing and engineering tasks. This aim leads us to a UNIX-like "little tools" philosophy, where rather than having one monolithic program for mutant generation, execution, and presentation of results, we provide separate tools for mutant generation, mutant execution, exclusion of mutants based on code coverage, and so forth, using simple text files as a means of communication between these tools.

Such an approach has limitations, of course. In general, it will usually produce more invalid mutants, or generally uninteresting mutants (e.g., modifying part of a constant string), and it will be difficult to apply many of the methods proposed for speeding up mutation testing, such as mutant schemata [28] or "forking" approaches [13, 29]. However, we believe that having a somewhat inefficient mutation tool is better than having no tool at all for the many languages currently lacking tools. Moreover, the more sophisticated techniques for making mutation testing more efficient also make it more difficult to allow custom mutation operators, discouraging project-specific approaches or experimentation with novel operators, e.g., for concurrency or functional programming.

The design of `universalmutator` is simple. After determining which language a source file is written in, it applies appropriate regular-expression-based mutation rules, contained in `.rules` files (a number of which are provided with the tool). In most cases, a source file will be mutated based on multiple rule files: there is a set of "universal" mutations rules, common to all languages, a set of rules for C-like languages, and then (often) a set of rules for each specific language. For instance, mutating a Swift program uses the files `universal.rules`, `c_like.rules`, and `swift.rules`.

Figure 1 shows the set of rules that are applied in common to all languages. Some of these rules are irrelevant to some languages (for instance, Python does not use && as a logical operator), but in such cases the presence of the rule is harmless. The core mutations defined here include various operator replacements, numeric and string constant replacements (introducing several additions to the set used by Andrews [5], such as replacing any string by the empty string), and the insertion of new `break` statements. The universal rules also include an example of a mechanism for annotating that a line should never be mutated. This is a feature not (to our knowledge) present in most tools, but can be useful: first, some lines may be known to relate to untested code (e.g., `WARNING` messages that are never checked by tests); second, in rare instances a statement

```
DO_NOT_MUTATE ==> DO_NOT_MUTATE
\+ ==> -
\+ ==> *
\+ ==> /
\+ ==> %
-([^>]) ==> +\1
-([^>]) ==> *\1
-([^>]) ==> /\1
-([^>]) ==> %\1
([^/*])([^/*]) ==> \1+\2
([^/*])([^/*]) ==> \1-\2
([^/*])([^/*]) ==> \1/\2
([^/*])([^/*]) ==> \1%\2
([^/])/([^/]) ==> \1+\2
([^/])/([^/]) ==> \1-\2
([^/])/([^/]) ==> \1*\2
([^/])/([^/]) ==> \1%\2
% ==> +
% ==> -
% ==> *
% ==> /
!= ==> ==
!= ==> <=
!= ==> >=
!= ==> >
!= ==> <
== ==> !=
== ==> <=
== ==> >=
== ==> >
== ==> <
>= ==> ==
>= ==> !=
>= ==> <
>= ==> >
<= ==> ==
<= ==> !=
<= ==> <
<= ==> >
< ==> >
< ==> ==
([^-])> ==> \1<
([^-])> ==> \1==
-([^>]) ==> \1
(\D)(\d+)(\D) ==> \g<1>0\3
(\D)(\d+)(\D) ==> \g<1>1\3
(\D)(\d+)(\D) ==> \g<1>-1\3
(\D)(\d+)(\D) ==> \1\2+1\3
(\D)(\d+)(\D) ==> \1\2-1\3
&& ==> ||
\|\| ==> &&
! ==>
([^&])&([^&]) ==> \1|\2
([^|])\|([^|]) ==> \1&\2
(^\s*)(\S+.*)\n ==> \1\2\n\1break;\n
".+" ==> ""
```

Figure 1: Universal mutation rules for all languages

may be dangerous to mutate, such as a network or file system operation, but there is interest in mutating other aspects of computation. The example rule file makes the structure of a mutation rule clear: the left hand side of a mutation is a regular expression (regexp), using Python's notation for regexps; the right hand side (across the ==>) is a Python regular expression replacement string, which can use \1, \2, etc. to include groups from the regexp that matched. Under this scheme, most mutations are simple to define, and the only major limitation for common mutation operators is that the approach does not handle deletion of multiple-line statements.

2 EXAMPLE USAGE

The tool can easily be installed using Python's `pip` package management system:

```
% pip install universalmutator
```

The module is pure Python and depends on no modules other than built-in Python libraries, so it should be usable on almost any

system that supports Python 2.7+. Mutation with TCE of a particular language, of course, may depend on having that language's compiler available (e.g., `swiftc` is used to compile swift code).

To mutate a source file, use the command:

```
% mutate <sourcefile>
```

If the source file extension is one of those supported (currently `.c`, `.cpp`, `.java`, `.py`, and `.swift`), the tool will automatically determine the rules to use and generate a set of mutants. For example, if `<sourcefile>` is `avltree.c`, it will generate `avltree.mutant.0.c`, `avltree.mutant.1.c`, and so forth. For Swift and Python files, the tool will also attempt to compile each mutant, and only generate mutant files for mutants that (1) compile without syntax errors and (2) do not produce equivalent object files to either the original source code or another already-generated mutant. Because the use of a linker, include paths, and compiler directives makes it difficult to automatically check C and C++ files for successful compilation[2], the tool also supports a mode where the user provides a compilation command to be used to check for valid mutants, e.g.:

```
% mutate foo.c --cmd "clang -c MUTANT -I ."
```

where `MUTANT` will be replaced by the name of a generated temporary file for the mutant, clang will be used as the compiler, and the current directory will be used as an include path. Constructing a TCE checker to compare C and C++ object files is non-trivial, due to build complexities and object formats that include timestamps, so we allow users to write Python handlers that perform that task in a project- and compiler-specific way.

To analyze the generated mutants, provide the tool both the name of the source file to be mutated and the command to run to process the mutants. For example, the complete process for performing mutation analysis on the TSTL [15, 18] Python testing tool's AVL tree example from scratch is:

```
$ pip install tstl
$ git clone https://github.com/agroce/tstl
$ cd tstl/examples/AVL
$ tstl avlnew.tstl
$ mutate avl.py
$ analyze_mutants avl.py "ulimit -t30; tstl_rt -t 10"
```

The `mutate` tool will generate 799 valid mutants, 493 invalid mutants (e.g., changing an `if` into a `pass` in a failed statement deletion), and 17 redundant mutants equivalent either to one of the valid mutants or the original `avl.py`. Running `analyze_mutants` will produce a pair of files, `notkilled.txt` and `killed.txt`, containing names of mutants for which the `tstl_rt` command (which runs TSTL's random tester, with a timeout of 10 seconds, here run with a ulimit of 30 seconds to catch mutants producing non-termination) returned an error code of zero or non-zero, respectively. Ten seconds of random testing kills 386 of the mutants, but 413 survive.

The `analyze_mutants` tool optionally takes a third argument, the name of a file containing mutants to ignore. This lets users write their own code to prune mutants without having to remove the mutant files. The `universalmutator` provides an additional tool, `check_covered <sourcefile> <coveragefile> <outfile>` that scans all mutants of sourcefile, determines their location (recall that for us all mutants will correspond to a single line of `sourcefile`), and then outputs a file (`outfile`) containing all mutants not present in `coveragefile`. The expected format for `coveragefile` is a list of line numbers, delimited by whitespace, starting from 1; the optional `--tstl` argument also allows the tool to process internal coverage reports from the `tstl` testing tools. Of the 413 surviving mutants, 67 are in code not covered during even 5 minutes of random testing, mostly test code not part of the actual AVL class.

[2]This difficulty is a significant contrast to languages like Python or Swift, where even a file that is part of a large project is often easily re-compiled by itself, without any extra options or configuration.

3 EXTENDING THE MUTATOR

The parser-free nature of `universalmutator` makes it easy to use in ways that might not easily work with a more traditional mutation tool. For example, it can handle files that are "almost" in some known-to-the-tool language just as well as files actually in the language, which would usually be rejected by a parser-based tool. Model checking and testing tools, such as TSTL and Spin [19], embed other languages in test harnesses (Python for TSTL, C for Spin's PROMELA language). In these cases one just mutates the appropriate file, after telling the mutator what language it is "written in." Some mutants may be invalid, but these will just be rejected by the language tools and discarded.

However, in some cases simply using an existing language's rules to mutate something close to that language is not all that is needed. One may want to add a new mutant, for a project, or one may need to define basic mutation operators for a novel programming language with unusual syntax. In both cases, the solution is simple: just write a new `.rules` file, and either add it to the `mutate` call, or, if there is a real need to go outside the box, and the `universal` mutation rules no longer apply, use the language `none` and explicitly note the rule file(s) to be used.

Because of the regexp style of defining operators, one can write a quick, clumsy rule if one wants to try an experimental operator out and see how well it works, or refine it further. For instance, one might think that changing `while` loops into `if` statements is a promising idea for a mutation operator[3]. This does not represent a common human error, probably, but can check test suites to make sure they are good at investigating loop iterations beyond running zero or one or more times. A quick-and-dirty sketch of the rule might look like:

```
while ==> if
```

This version of the rule will largely suffice. It will, of course, replace `while` strings in function names, but perhaps this is not enough of a problem to matter. If one does eventually decide that the rule needs refining (for instance, when working with a large project where there are dozens of methods on a timed event class named with the pattern `whileFoo`), one can ensure that `while` is a word on its own, using Python's regexp:

```
while(\W) ==> if\1
```

This rule forces the character after the `while` to be something that shows the `while` is not part of a valid identifier, in most languages. We capture it using the parentheses and reproduce the same character in the replacement using the `\1` back-reference.

Another use for the same mechanism is to *avoid* mutating code. For instance, imagine testing a large project with many warning messages, always consisting of the use of a `WARNING` macro. The system tests do not make use of these warnings, and so they end up as a

[3]We added this one to the universal rules file, after writing about it here.

Table 1: Java mutation results

Subject	PIT			Major			universalmutator		
	Gen	Kill	MS	Gen	Kill	MS	Gen	Kill	MS
Triangle	45	44	97.78%	130	130	93.53%	188	184	97.87%
FizzBuzz	111	92	82.88%	244	202	82.79%	203	176	86.70%

large set of equivalent mutants, modifying the strings, deleting the statements, and modifying numeric constants or arithmetic operators in the warnings. Annotating all of these with DO_NOT_MUTATE is inconvenient and hurts program readability. Instead, one can just add a nowarnings.rules to every mutate call:

```
WARNING ==> DO_NOT_MUTATE
```

4 EVALUATION

As a simple evaluation of the capabilities of our approach, we compared the mutants generated by universalmutator to those generated by Andrews' tool [5], used in papers on mutation-based model checking of C programs [14] and mutations as a tool for improving Linux kernel test suites [3]. The mutants generated by our tool were a superset of those generated by Andrews' tool for the 8 C files used in these papers. Most importantly, the actual mutants used to drive fixes to the model-checking harnesses or the Linux kernel test suite were produced by both tools.

We also applied our tool to an ongoing effort to test the pyfakefs file system [1], using TSTL [15, 18], and confirmed that our mutants were equally useful for test suite evaluation and improvement as the mutants produced by muupi [24], a much more complex, AST-based tool specialized for Python.

Finally, we compared our results to PIT (ver. 1.2.5-SNAPSHOT) [6] and Major (ver. v1.3.2) [22] for both the classic pedagogical Triangle example and an interview-quiz GitHub project, FizzBuzz[4]. We used all the default mutation operators for both PIT and Major. Specifically, here are the commands we used to perform mutation testing using universalmutator for Triangle:

```
$ mkdir mutants
$ mutate src/main/java/triangle/Triangle.java --mutantDir mutants
...
$ analyze_mutants src/main/java/triangle/Triangle.java "mvn test" \
  --mutantDir mutants
```

Table 1 shows the experimental results. For each tool on each subject, we show the number of generated mutants (column "Gen"), the number of killed mutants (column "Kill"), and the mutation score (column "MS"). From the table, we can observe that all three tools provide similar mutation scores, demonstrating usefulness of our tool for Java, even without any ability to parse Java (or any Java-specific rules beyond the erasure of the synchronized keyword) or handle multi-line constructs.

5 CONCLUSION

While most mutation tools either attempt to parse a source file or operate at the bytecode level, this paper proposes another possibility: we present universalmutator, a multi-language tool based on simple regular-expression defined transformations of source as un-parsed text. The advantages of the approach are its simplicity of extension, ability to work on files that do not actually parse, and the ease with which it supports adding mutation support for new programming languages. As future work, we plan to extend the ease of use of the tool, further integrate it with popular build environments, and add new languages, such as Rust. universalmutator is open-source and publicly available on GitHub at https://github.com/agroce/universalmutator.

[4]https://github.com/EnterpriseQualityCoding/FizzBuzzEnterpriseEdition

ACKNOWLEDGMENTS

We thank Farah Hariri for discussions on mutation testing. This work was partially supported by National Science Foundation grants CCF-1409423, CCF-1421503, and CCF-1566589.

REFERENCES

[1] 2011. pyfakefs implements a fake file system that mocks the Python file system modules. https://github.com/jmcgeheeiv/pyfakefs. (June 2011).
[2] I. Ahmed, R. Gopinath, C. Brindescu, A. Groce, and C. Jensen. 2016. Can testedness be effectively measured?. In *FSE*.
[3] I. Ahmed, C. Jensen, A. Groce, and P. E. McKenney. 2017. Applying mutation analysis on kernel test suites: an experience report. In *MUTATION*.
[4] M. A. Alipour, A. Shi, R. Gopinath, D. Marinov, and A. Groce. 2016. Evaluating non-adequate test-case reduction. In *ASE*.
[5] J. H. Andrews, L. C. Briand, and Y. Labiche. 2005. Is mutation an appropriate tool for testing experiments?. In *ICSE*.
[6] H. Coles. 2010. PIT Mutation Testing. http://pittest.org/. (2010).
[7] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. 1978. Hints on test data selection: Help for the practicing programmer. *Computer* 4, 11 (1978).
[8] M. Gligoric, A. Groce, C. Zhang, R. Sharma, A. Alipour, and D. Marinov. 2013. Comparing non-adequate test suites using coverage criteria. In *ISSTA*.
[9] M. Gligoric, S. Khurshid, S. Misailovic, and A. Shi. 2017. Mutation testing meets approximate computing. In *ICSE NIER*.
[10] R. Gopinath, I. Ahmed, M. A. Alipour, C. Jensen, and A. Groce. 2017. Mutation reduction strategies considered harmful. *IEEE Trans. Rel.* 66, 3 (2017).
[11] R. Gopinath, A. Alipour, I. Ahmed, C. Jensen, and A. Groce. 2015. How hard does mutation analysis have to be, anyway?. In *ISSRE*.
[12] R. Gopinath, M. A. Alipour, I. Ahmed, C. Jensen, and A. Groce. 2016. On the limits of mutation reduction strategies. In *ICSE*.
[13] R. Gopinath, C. Jensen, and A. Groce. 2016. Topsy-Turvy: a smarter and faster parallelization of mutation analysis. In *ICSE*.
[14] A. Groce, I. Ahmed, C. Jensen, and P. E. McKenney. 2015. How verified is my code? Falsification-driven verification. In *ASE*.
[15] A. Groce and J. Pinto. 2015. A little language for testing. In *NFM*.
[16] F. Hariri, A. Shi, H. Converse, S. Khurshid, and D. Marinov. 2016. Evaluating the effects of compiler optimizations on mutation testing at the compiler IR level. In *ISSRE*.
[17] F. Hariri, A. Shi, O. Legunsen, M. Gligoric, S. Khurshid, and S. Misailovic. 2018. Approximate transformations as mutation operators. In *ICST*.
[18] J. Holmes, A. Groce, J. Pinto, P. Mittal, P. Azimi, K. Kellar, and J. O'Brien. 2017. TSTL: the template scripting testing language. *STTT* 20, 1 (2017).
[19] G. J. Holzmann. 2003. *The SPIN model checker: primer and reference manual*. Addison-Wesley Professional.
[20] S. Hong, B. Lee, T. Kwak, Y. Jeon, B. Ko, Y. Kim, and M. Kim. 2015. Mutation-based fault localization for real-world multilingual programs. In *ASE*.
[21] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser. 2014. Are mutants a valid substitute for real faults in software testing?. In *FSE*.
[22] R. Just, F. Schweiggert, and G. M. Kapfhammer. 2011. MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler. In *ASE*.
[23] X. Li and L. Zhang. 2017. Transforming programs and tests in tandem for fault localization. *Proc. ACM Program. Lang.* 1, OOPSLA (2017).
[24] X. Liu. 2016. muupi mutation tool. https://github.com/apepkuss/muupi. (September 2016).
[25] J. Offutt and A. Abdurazik. 2000. In *Mutation 2000: mutation testing in the twentieth and the twenty first centuries*.
[26] M. Papadakis, Y. Jia, M. Harman, and Y. Le Traon. 2015. Trivial compiler equivalence: a large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In *ICSE*.
[27] M. Papadakis and Y. Le Traon. 2015. Metallaxis-FL: mutation-based fault localization. *Softw. Test., Verif. Reliab.* 25, 5-7 (2015).
[28] R. H. Untch, A. J. Offutt, and M. J. Harrold. 1993. Mutation analysis using mutant schemata. In *SEN*, Vol. 18.
[29] B. Wang, Y. Xiong, Y. Shi, L. Zhang, and D. Hao. 2017. Faster mutation analysis via equivalence modulo states. In *ISSTA*.