Syntax Is All You Need: A Universal-Language Approach to Mutant Generation

SOURAV DEB, Northern Arizona University, United States
KUSH JAIN, Carnegie Mellon University, United States
RIJNARD VAN TONDER, Mysten Labs, United States
CLAIRE LE GOUES, Carnegie Mellon University, United States
ALEX GROCE, Northern Arizona University, United States

While mutation testing has been a topic of academic interest for decades, it is only recently that "real-world" developers, including industry leaders such as Google and Meta, have adopted mutation testing. We propose a new approach to the development of mutation testing tools, and in particular the core challenge of *generating mutants*. Current practice tends towards two limited approaches to mutation generation: mutants are either (1) generated at the bytecode/IR level, and thus neither human readable nor adaptable to source-level features of languages or projects, or (2) generated at the source level by language-specific tools that are hard to write and maintain, and in fact are often abandoned by both developers and users. We propose instead that source-level mutation generation is a special case of *program transformation* in general, and that adopting this approach allows for a single tool that can effectively generate source-level mutants for essentially *any* programming language. Furthermore, by using *parser parser combinators* many of the seeming limitations of an any-language approach can be overcome, without the need to parse specific languages. We compare this new approach to mutation to existing tools, and demonstrate the advantages of using parser parser combinators to improve on a regular-expression based approach to generation. Finally, we show that our approach can provide effective mutant generation even for a language for which it lacks any language-specific operators, and that is not very similar in syntax to any language it has been applied to previously.

CCS Concepts: • **Software and its engineering → Dynamic analysis**; **Software testing and debugging**.

Additional Key Words and Phrases: Software Testing, Mutants, Mutation Generation

ACM Reference Format:
Sourav Deb, Kush Jain, Rijnard van Tonder, Claire Le Goues, and Alex Groce. 2024. Syntax Is All You Need: A Universal-Language Approach to Mutant Generation. *Proc. ACM Softw. Eng.* 1, FSE, Article 30 (July 2024), 21 pages. https://doi.org/10.1145/3643756

1 INTRODUCTION

Mutation testing, though introduced in the late 1970s [22, 22, 23], has long been a topic of academic interest. With the recent adoption of mutation testing by bellwether software industry companies including Google [31], Meta [4], and Amazon [24], interest in using mutation testing for real-world projects has grown widely in practice.

Authors' addresses: Sourav Deb, Northern Arizona University, Flagstaff, AZ, United States; Kush Jain, Carnegie Mellon University, Pittsburgh, PA, United States; Rijnard van Tonder, Mysten Labs, Palo Alto, CA, United States; Claire Le Goues, Carnegie Mellon University, Pittsburgh, PA, United States; Alex Groce, Northern Arizona University, Flagstaff, AZ, United States.

ACM 2994-970X/2024/7-ART30
https://doi.org/10.1145/3643756

Indeed, interest in mutation testing has become widespread enough in the open source community that a popular (more than 300 stars on GitHub) repository, "Awesome Mutation Testing," lists more than 40 tools, covering almost twenty languages.[1] Some of these tools are the products of academic research; others are essentially the hobby projects of developers interested in enabling mutation testing for their favorite language.

This diversity of tools highlights the growth of widespread interest in mutation testing. However, it also demonstrates various ways that bringing mutation testing to typically polyglot modern projects is cumbersome. Each language (or IR, at minimum) requires its own underlying mutation and testing framework. Mutation tools are not trivial to develop; the smallest single-language tools we examined were nearly 4K LOC, and the mean size of a mutation testing tool was over 20K non-comment source lines.

Newer languages, including ones with considerable interest in the development community, often lack mutation testing tools. Even languages with a long history may long lack a working mutation testing tool; e.g., it was not until 2014 that Haskell had such a tool [21]. Furthermore, the Haskell mutation tool is no longer supported,[2] and it appears that no successor tool has appeared in the ensuing years.[3] Haskell developers (until now, see below) have no option for performing true mutation testing. Similarly, while the R language is used by more than 2 million statisticians and data scientists, the only mutation testing tool for R is a prototype-in-progress that supports a very limited set of mutation operators (https://github.com/sckott/mutant); the last update to the tool was made in late 2021.

Indeed, abandonment is a feature of the sad history of many mutation testing tools, exacerbating the challenges of its uptake. The page listing abandoned mutation testing tools (https://github.com/theofidry/awesome-mutation-testing/blob/master/abandoned.md) lists almost half as many tools as are under active maintenance. Each tool represents considerable development effort, and was popular enough to be noted in the "Awesome Mutation Testing" resource. Many of the abandoned repositories have more than 20 stars, suggesting inconvenienced users, especially for tools targeting evolving languages. While this is a minor concern for languages such as C that are relatively stable, it means that a mutation tool may fail to parse C++ 2020, and is particularly problematic for faster-moving languages like Rust.

A great deal of research effort has been devoted to devising clever schemes for reducing the computational burden of mutation testing over the years (e.g., [19, 28, 34]); in practice, however, the recent adoption of mutation testing arguably relies more on the availability of vast (cloud) computing resources than any particular approach from the literature, beyond heuristic restrictions on the mutants used (e.g., one per line, or only some simple operators). On the other hand, almost no research has addressed perhaps the most central issue of mutation testing: in the absence of a working mutation testing tool for the language in which one is developing software, one cannot obtain the benefits of mutation testing. The absence of such tools is due to the high human cost of developing and maintaining mutation testing tools. In particular, new languages may have unstable grammars, and few tools to aid developers in parsing and manipulating source code. Ideally, mutation testing would be available for *any* newly proposed programming language, as soon as that language has any community of users at all. In the present state-of-the-art, however, it may be years before even rudimentary tools appear, and they may "disappear" shortly there-after. In fact, the difficulties of parsing code make bytecode or IL-level mutation a popular recent choice

[1] https://github.com/theofidry/awesome-mutation-testing

[2] The last commits or package updates were in 2015, and the primary author confirms that changes to `haskell-src-ext` since introduced render it unusable at present.

[3] There is FitSpec [5] which is maintained, but FitSpec does not perform traditional mutation testing or produce source code (or bytecode) mutants; it rather performs black-box type-aware mutation of function return values.

in tool development, despite the advantages of source-based mutation for users [17], including the ability to read and understand mutants (which is often essential if one aims to add a test to kill a mutant) and to define mutants in terms of source-level language (or even project/library-specific features). Moreover, even arguably less desirable bytecode mutation approach is not available for languages that do not compile to, e.g., Java or LLVM bytecode– e.g., Haskell and R.

Developing a large number of mutation testing tools, each requiring its own development community, workflow for mutation, and effort to keep up with language changes, may, however, be unnecessary. The core problem of mutation testing, *mutant generation*, is simply a particular instance of the problem of *program transformation*. Program transformation considers the problem of taking a program, P, and producing a program, P' that is the result of applying some function f to P (i.e., $P' = f(P)$). In mutant generation, we consider a family of functions f where each $f_{o,loc}$ is an instantiation of a *mutation operator* o (as classically, if often informally, defined in the literature) at a particular program location loc in P. The inclusion of loc in the specification of f indicates that the problem is a specialization of general program transformation in that all classic mutation operators we are aware of are *local*: they only affect a small portion of a program, generally a contiguous string of characters or tokens.

Moreover, as this suggests, mutation testing can be treated as an almost entirely *syntactic* transformation problem; in fact, in their classic introduction to software testing, Ammann and Offutt refer to mutation testing as "syntax-based testing" [3]. While, e.g., refactoring may need to "understand" a program to some extent, mutation can often operate with essentially no context beyond a single line of code.

We further propose that in fact, for the most part, mutant generation need not even "know" the syntax of a target language, in terms of a complete grammar. Instead, mutation testing can operate to a large extent in a *universal* manner, where programs are transformed at the level of *local patterns*. That is, each o and loc can be defined without reference to, e.g., the context-free grammar of a programming language, but only with respect to a position in P treated as a string, and a transformation on a string defined in terms of, e.g., regular expressions with capture and replacement. Treating mutant generation in this way has two major benefits:

(1) A single tool can provide mutant generation for almost any programming language, without needing to parse that language, and rules for one language can be re-used for another language. This reduces the development and maintenance effort for mutation testing tools by orders of magnitude. Moreover, it removes the need for many maintenance activities related to parsing code; such changes need to be made only when a change to the language would introduce new mutation operators or modify existing ones. The local nature of operators tends to make the second case rare.
(2) The existing literature on multi-lingual or language agnostic program transformation can be applied to the problem of mutant generation, making it easy to express mutation operators that operate across many languages, and efficient to generate valid mutants even in the absence of language-specific development effort.

It is true that this approach to mutant generation does pay a cost in terms of generating invalid mutants that must be rejected at the compilation stage of the mutation testing process. However, as observed by Pizzoleto et al. in their systematic review of mutation testing [32] cost reduction approaches, mutation testing consists of four stages: 1) execution of a test suite on the un-mutated program, 2) generation of mutants, 3) execution of the test suite on the mutants, and 4) analysis (by a human, usually) of the mutants. It is only steps 3 and 4 that are "very costly," as Pizzoletto et al. note. Our approach increases the costs of the second step only, and, as we discuss below, can help reduce costs of steps 3 and 4.

The contributions of this paper are therefore:

(1) The description of a novel approach to mutant generation based on generalized description of multi-lingual program transformations that allows a single tool to handle mutant generation for essentially all programming languages. This approach also makes mutation testing almost trivial to extend to most new languages and even allows developers to write project-specific rules easily without having to modify a mutation tool's implementation.
(2) A proposal to use parser parser combinators to improve the efficiency and expressive power of that approach, and an evaluation of the gains thus achieved.
(3) A comparison of an implementation of mutant generation based on this approach with existing tools for four important programming languages. The single-language tools range from approximately 3,800 LOC to nearly 60,000 LOC, and, we emphasize, each handles one programming language. Our tool, which provides comparable core mutation testing functionality, supports over a dozen languages using only about 2,200 LOC of Python and less than 500 lines for rules defining mutation operators over a dozen specific languages (and which can effectively mutate an essentially unlimited number of other languages). Our approach enables a smaller tool than any other tool we examined (by over 1000 LOC) to support a rich set of operators for all languages covered by other tools, plus additional languages.
(4) Far from being limited by its implementation size and universality, our tool *generates a larger number of valid, non-equivalent mutants* for every language we examined than any competing, language-specific tool. That these additional mutants are meaningful, rather than obscure, is further supported by the fact that our mutation scores are similar to those for other tools.
(5) A small case study showing that our approach produces useful results, even without the definition of language-specific rules, for Haskell, a language arguably well outside the language paradigms we considered when developing the universal mutation rules.

Our tool implementing these ideas, the *universalmutator*, is available on GitHub at https://github.com/agroce/universalmutator, and can be installed via Python pip. We welcome contributions and improvement suggestions. Our tool is the official recommended tool for mutation of Solidity programs (https://docs.soliditylang.org/en/latest/resources.html).

2 A UNIVERSAL-LANGUAGE SYNTAX-BASED APPROACH

The key insight of this paper is that *most mutation operators proposed in the literature do not in fact require parsing of source code.* Consider, for example, one of the most commonly used mutation operators, replacement of arithmetic operations. We do not need to parse a program containing the string `''x + y''` in order to replace that string with `''x - y''`. Instead, this change is guaranteed to be effected if we simply search the program, represented as a string, for all occurrences of `''+''`, and produce for each such occurrence, a mutant where the character is replaced by `''-''`. Many languages express arithmetic as infix operators between expressions; if expressed syntactically, mutations for them can be expressed once, trivially implementing a large fraction of the classic Mothra [25] mutation operators. In a *syntactical* (but not semantic) sense, we claim, there is almost a "universal" (very abstract, never implemented) programming language, of which almost all particular programming languages can be treated as mere dialects.

The obvious objection to this simplistic approach is that unless we parse the code to identify arithmetic expressions, some of these replacements will be invalid; the universal language is not actually real! E.g., in a C++ program we will produce mutants like `x-+` replacing `x++`. However, most of these instances can be avoided by slightly more judicious choice of search target, e.g., instances of `''+''` where the preceding or following character is not another `''+''`. This may still

in some cases produce invalid mutants, but rejecting such mutants is easy, since they will fail to compile. Of course, we pay the price of compiling each mutant at generation time.

On the positive side, this argues that we can replace a complex implementation requiring parsing and analysis with one that only requires string search-and-replace. Such an implementation on face can be done using much less code. Moreover, defining new mutation operators is not a matter of understanding a parsed program representation, but is essentially a matter of providing two strings, where the second is a proposed replacement for the first. This simplifies operator specification from implementation of tree transformations into a minimal list.

Of course, simple string replacement cannot handle many interesting mutations. Consider the classic case of statement deletion, one of the most widely used and important mutation operators [7]. This cannot be concisely specified with string pairs. However, if we allow the use of *regular expression match and replace pairs*, we can express this operator, with some precision:

```
(^\s*)(\S+[^{}]+.*)\n ==> \1/*\2*/\n
```

In Python regular expression syntax, this expresses the replacement of strings with a possibly-empty amount of white space followed by a character pattern indicating a line of source code, with the same source code, preserving indentation, turned into a comment. While much more expressive than simple string replacement, regular expressions are familiar to most developers.

2.1 Hierarchies of Operator Applicability

Regular expressions suggest a mechanism for textual transformation customized to multiple target languages, which could then share a single engine to apply rules to source files without the complexity of parsing. However, we observe an additional property of mutation operators and mutation testing that speaks to the potential of a language-agnostic approach: many of the most widely used mutation operators in the literature are *universal*. For example, consider the case of replacing instances of ‘‘+’’ with ‘‘-’’. This mutation is not tied to any particular language, but applies to *almost all programming languages in use.* Others are not quite so widely applicable, but still apply to a large number of specific languages, e.g., many programming languages share C's logical operators, though important exceptions like Python and Lisp-family languages do not.

Implementing a tool for a language L therefore, generally need not require writing rules for all mutation operators to be applied to L programs, but instead can proceed by process of 1) identifying L's place in a *hierarchy* of language *families*, and then 2) identifying additional operators needed for L itself. The first of these tasks is often trivial, as in practice there are only a few basic syntactic forms for languages, at the level needed to describe transformation rules for operators. In fact, ignoring the second step will often provide "good-enough" mutation testing, in that, for example, most of the Mothra rules and statement deletion can be entirely handled without descending to the specific-language level at all.

As an example, consider mutation of Java code. Many typical Java mutants can be generated by the kind of "universal" rules (e.g., arithmetic operator replacements) considered above. Other mutation operators suitable for Java source are provided by considering the set of "C-like" languages that use the logical operators and control constructs common to e.g., Java, C, and C++ (`if`, `while`, etc.). Statement deletion at the line level can also be implemented by using the common comment notation for such languages. Implementing Java mutation, given universal and C-like rules, may require no more than a handful of additional rules.

The left-hand-side of Table 1 shows examples of universal, C-like, and language-specific rules for a few languages, taken directly from our implementation (see Section 3). In all cases, the form of a rule is: `regexp ==> replacement`, using Python regular expression and replacement syntax.

Table 1. Mutation operator rule examples, using regular expressions (left) and the Comby language (right), based on parser parser combinators.

Regexp	Comby
Universal	
`< ==> >`	`:[a]<:[b] ==> :[a]>:[b]`
`< ==> ==`	`:[a]<:[b] ==> :[a]==:[b]`
`< ==> <=`	`:[a]<:[b] ==> :[a]<=:[b]`
C-Like	
`else ==>`	`else ==>`
`if (\(.*\)) ==> if (!\1)`	`if (:[cond]) ==> if (!(:[cond]))`
`if (\(.*\)) ==> if (1==1)`	`if (:[cond]) ==> if (1==1)`
Language-Specific	
`synchronized ==>`	`synchronized ==>`
`(^\s*)(\S+.*\n) ==> \1pass`	`:[ s]:[expr]:[lf [\n]] ==> :[s]pass:[lf]`
`any_of ==> all_of`	`any_of ==> all_of`

Such rules are often non-trivial to write and read (and especially to debug); we show below that using parser parser combinators often simplifies and clarifies matters.

We first show universal rules drawn from the large set of transformations for less-than comparisons. The C-like rules are somewhat more complex: for removing elses (which always leaves valid code in C-like languages, but with the else clause always executing) and for replacing if conditions with, respectively, a negation and a constantly true expression. Finally, we show language-specific rules, for, respectively, Java, Python (statement deletion, including a `pass` and respecting indentation level), and C++. The last rule was contributed by a user using our tool to perform mutation testing on the Bitcoin core implementation. For a C++ program, all of the universal rules and C-like rules will be applied, plus the last language-specific rule. For a Python program, however, only the universal and Python-specific rules apply.

2.2 Beyond Regular Expressions with Parser Parser Combinators

While regular expressions have been shown to be capable of driving mutant generation, we propose going beyond regular expressions using a lightweight parser-driven approach that attunes mutation operators to language-general syntax-aware transformations. Such operators can perform tasks that go beyond regular transforms and ensure syntactically well-formed outputs (e.g., transformations must preserve balanced parentheses or braces). More generally, operators can be made sensitive to coarse-grained context-free grammar properties. In short, these operators go beyond the expressivity of regular expressions to generate more *varied* and more *valid* mutants.

Parser parser combinators [35] is a recent approach enabling syntax-transformation for multiple languages, and (we show) an apt choice for mutation testing operators. This key idea is to replace code constructs common to many languages (e.g., multi-line code blocks delineated by braces) with simple declarative patterns. Using parser combinators for coarse syntax matching avoids the complexity of integrating heterogeneous rewrite tools for mutation testing (e.g., one for Java, one for C++) and human effort to implement language-specific mutation operators (e.g., where the user has to learn to write operators for each language tool). The mechanization and expressive properties of parser parser combinators are covered in prior literature [35]; in this paper we demonstrate that

their application is uniquely effective for universal source-based mutation testing. In particular, we build mutation operators with Comby (https://comby.dev), the tool that implements parser parser combinators to declaratively match and rewrite source code syntax.

Comby is especially suited for matching multi-line code blocks and disambiguating code from comments and strings—constructs that otherwise confound and complicate regular expression patterns. Comby supports over 40 languages and implements a generic parser for additional languages to recognize their syntactic constructs. The following is a brief overview of Comby definitions and syntax for mutation operators implemented in our approach:

- `:[hole]` syntax matches source code assigned to a variable *hole*. Holes match all characters (including whitespace) lazily up to its suffix (analogous to `.*?` in regex), but only *within* its level of balanced delimiters. For example, `{:[hole]}` will match all characters inside balanced braces. By default, parentheses and square brackets are also treated as balanced delimiters, as applicable to most languages.
- `":[x]"` matches the body of a well-quoted string. Unlike `:[x]` (without quotes), the quoted variety implies that a data string may be any value, including strings that contain unbalanced parentheses, like `"item)"`.
- `:[hole:e]` matches words like `hello` and contiguous well-balanced expression-like syntax, like `print("hello world")` or `(a + b)`. It stops matching at whitespace boundaries, and so does not match a string like `a + b`. It also does not match unbalanced code syntax like `foo)` in typical languages where expressions are expected to be well-balanced.
- `:[[hole]]` matches alphanumeric characters in source code (similar to `\w+` in regex).
- When variable *hole* occurs multiple times in a match template, matched values must be equal.
- Non-whitespace characters are matched literally.
- Contiguous whitespace (e.g., spaces, newline) match contiguous whitespace in the source code. That is, match templates are sensitive to the *presence* of whitespace, but *not* exact layout (i.e., the number of spaces may not correspond exactly between template and source code).
- Matching is insensitive to comments. Comments are parsed as whitespace when matching non-hole syntax.

The right-hand-side of Table 1 shows the same regular-expression rules implemented as Comby patterns. While some simple rules are unchanged, others are either more succinct or provide more context as to applicability (e.g., only less than between valid holes should be mutated). This shows the further major benefit of using Comby: once a user has learned the simple concepts above, it is often much easier to read—and especially *write*—mutation operator definitions in Comby than using regular expressions. The added precision comes not only "for free" but in fact yields a benefit for the "end-user-programmer" of mutations.

Comby provides mechanisms to further define custom match syntax and behavior beyond these standard patterns. Comby patterns can also embed regular expressions, subsuming their expressive power to enable regex-based mutation operators intermixed with context-free syntax properties.

3 IMPLEMENTATION

We implemented our approach in a mutation testing tool, written in about 2,200 lines of Python. The rules defining mutation operators for a set of languages including C, C++, Python, Java, Lisp, Swift, Rust, Solidity, Fe, Vyper, JavaScript, and Go, required another 436 lines; this is the total for Comby and regexp-based rules, so each version requires only 218 lines of rule definitions. The tool makes use of parser parser combinators via the `comby-python` module.

While many tools provide a GUI front-end, or integrate to a particular testing library, we chose to focus on an easily configured CLI, allowing a user to specify the commmands used to build mutants

or run tests. While less convenient in the case where there is a good fit between a tool's integrations and a project's environment, we found this flexibility useful even when limiting considerations to a single language. For example, the widely used PIT system is strongly tied to certain versions of JUnit, and mutating projects using a different testing infrastructure, or an older JUnit, can be difficult. Our approach also is suitable for use in CI and other automated environments. The tool attempts to automatically identify which rule sets to use in mutating a source file, but can also be instructed to run fewer or more rule sets, and allows users to supply their own custom rule sets.

For the most part we expect users to provide build commands, but do support automatic compilation and comparison of mutants for Python, Solidity, Fe, Vyper, Swift, Rust, and Java, including Trivial Compiler Equivalence [29] checks for redundancy of mutants. These default handlers can always be over-ridden if they do not work with a particular build setup (for Python, the approach almost always works, but in the case of multi-file projects in other languages, it usually requires some manual setup).

The tool also provides a number of utilities, including mechanisms for incorporating code coverage into mutant selection, integration with a Python automated testing tool, prioritization of unkilled mutants to show dis-similar mutants early in a ranking (similar to the FPF approach to identifying unique bugs found in fuzzing [6]), and tools for pruning a class of mutants based on some criteria (e.g., removing all mutants of logging statements from a set of generated mutants).

3.1 Supported Languages and Operators

For comparison, Google has used its substantial resources to provide mutant generation for C++, Java, Python, Javascript, Go, Typescript, and Common Lisp [30], all of which we also support. The operators Google uses for these languages (based on the original Mothra operators [25]) are a subset of those provided by our implementation, with the caveat that block removal is only supported when using Comby; regular expression mutation is limited to single-line statement deletion. For Lisp-like languages, the difficulty of identifying statements vs. value-returning function calls makes use of block deletion somewhat impractical. The majority of the Google Mothra operators can be implemented as universal rules for all languages; only specializing logical operators and implementing block/statement deletion even requires language identification. In addition to these core operators, we provide a number of additional operators for every language offered.

There is generally a one-to-one correspondence between Comby and Regexp rules in our implementation. The number of Comby rules (which is a small multiple of the number of "operators" being defined) for languages discussed below, at each hierarchical level in our implementation, is: 75 (universal), 33 (Python), 16 (C-like), 14 (C++), 7 (Java), and 2 (Rust).

The primary limitation of our tool is that it does not offer operators that require semantic knowledge of the program beyond the identification of roles offered by Comby. The only operator from the mutation testing literature that we are aware of that as a consequence cannot be supported by our approach is variable replacement. However, we are aware of no widely-used tool or even recent study of mutation testing that makes use of this type of operator, suggesting that it is difficult for other methods to implement, and is not in demand by users.

One concern with our approach is that every time the program changes, the cost of invalid mutants must be paid again. However, the locality of mutants and source changes in fact means this is seldom required. Because a mutant is a small source change, updating a mutant to reflect even large changes to a source file is nearly cost-free, and if the original mutant was valid, the new mutant will usually also be valid. New mutant generation is only required for new lines of code and modified lines of code. Our implementation allows the use of the `git merge-file` algorithm to control the modification of most mutants, and uses our mechanism for selective mutant generation to force mutation where there is a conflict in a "mutant patch."

4 EXPERIMENTAL EVALUATION

- RQ1: How does the use of parser parser combinators modify the efficiency (in terms of invalid mutants) and effectiveness (in terms of mutation score and equivalent mutants) of the universal approach to mutant generation?
- RQ2: How does the universal approach, using regular expressions or parser parser combinator rules, compare to existing single-language mutation tools, in terms of number of generated mutants, mutation scores, equivalent mutants, and other evaluation measures used in the literature?

Both research questions were addressed by generating and running mutants for a set of four programming languages: C++, Java, Python, and Rust. The criteria for language selection was as follows: C++, Java, and Python are among the most widely used programming languages at present. Furthermore, C++, unlike C, can be notoriously hard to parse, making development of mutation tools that use an AST and properly consider mutants of, e.g., code inside a template, difficult. Java is the language in which mutation testing has been studied most extensively in recent literature, and PIT is almost certainly the most popular and widely used mutation tool. Python long suffered from a lack of working and maintained mutation tools, despite a large number of abandoned efforts, though at present it is served by multiple maintained, AST-based tools. Finally, Rust is an "emerging language" with few tools targeting it. Table 4 gives an overview of tools we identified for each language as working tools capable of single-file mutant generation.

To evaluate both RQ1 and RQ2 we assembled a set of 24 source files, six from each language (about 9,000 total non-comment LOC). We first selected, using the GitHub search API and manual examination, six GitHub projects for each language meeting a set of criteria:

(1) The project must have a minimum of 1,000 stars.
(2) The percentage of source languages other than the target language must be small.
(3) There must be an executable test suite where all tests pass.
(4) The project must be relatively easy to build, without a large set of external dependencies.
(5) In addition, for Java, we required use of maven.

Files were then selected by picking for each project, the largest source file that was less than 20KB in size (we restricted our study to smaller source files because we had to manually examine mutants for equivalence, and determining context of changes for larger files proved extremely time-consuming). If that file was not covered by the test suite, the next closest to 20KB in order was chosen. Finally, if no source file under 20KB existed or was covered by tests, the smallest file larger than 20KB covered by tests was selected.

We additionally present a small case study, applying our implementation to a program in a language for which it has *no* rules, Haskell. The case study concerns a fairly trivial example, but was chosen due to being the only program for which data was available as to the effectiveness of the tool we compared against (since that tool no longer works).

4.1 Regular Expressions vs. Parser Parser Combinators

While the primary advantages of using parser parser combinators as a basis for mutant generation are in expressiveness and ease of use, particularly rule readability, a second advantage is a reduction in the number of invalid mutants generated.

Table 2 and Table 3 compare generated mutants for Regex and Comby modes of our tool. Invalid mutants are mutants that the tool produces, but which fail to compile. The cost of each such mutant is usually small, since compilation generally fails early for trivial reasons (e.g., `break` or `continue` outside a loop), but reducing the number of such mutants directly reduces the cost of initial mutant generation, and generally indicates a better "understanding" of the source being mutated.

Table 2. Regex vs. Comby for C++ (top), Java (middle), and Rust (bottom). LOC describes the project files considered. Muts. is total number of mutants generated for those files. Invalid mutants are those that don't compile. Mut. Score refers to the Mutation Score.

C++

Project	LOC	Regex			Comby		
		Muts.	% Valid	Mut. Score	Muts.	% Valid	Mut. Score
ompl	525	2533	23.49%	0.05	2174	25.94%	0.07
libgraphqlparser	137	741	21.19%	0.90	440	29.77%	0.89
rethinkdb_rebirth	471	2246	24.27%	0.12	2009	24.49%	0.10
xoreos	732	3876	62.41%	0.01	3297	69.15%	0.01
libxmljs	630	3989	22.09%	0.62	4008	19.71%	0.64
tiny-diff.-simulator	195	1282	58.27%	0.05	1214	63.43%	0.04

			Valid	Invalid	% Valid	Mut. Score
Overall	Regex	Mean	890.67	1553.83	35.28%	0.29
		Std.Dev.	787.62	956.31	19.48%	0.38
	Comby	Mean	837.83	1352.50	38.75%	0.29
		Std.Dev.	745.78	1058.33	21.66%	0.37

Java

Project	LOC	Regex			Comby		
		Muts.	% Valid	Mut. Score	Muts.	% Valid	Mut. Score
pebble	279	1014	8.48%	0.81	752	11.30%	0.80
spring-hateoas	397	2479	10.00%	0.33	1384	12.43%	0.37
javacc	363	1888	28.70%	0.23	1276	38.71%	0.21
coffee-gb	463	1927	11.63%	0.21	798	21.80%	0.23
egads	252	2454	25.43%	0.71	1163	51.33%	0.71
apk-parser	324	1583	22.17%	0.07	1199	27.77%	0.08

			Valid	Invalid	% Valid	Mut. Score
Overall	Regex	Mean	345.83	1545.00	17.74%	0.39
		Std.Dev.	203.86	467.98	8.73%	0.30
	Comby	Mean	309.17	786.17	27.22%	0.40
		Std.Dev.	202.44	235.24	15.59%	0.29

Rust

Project	LOC	Regex			Comby		
		Muts.	% Valid	Mut. Score	Muts.	% Valid	Mut. Score
cargo release	346	2014	42.25%	0.85	2401	70.39%	0.80
passarine	376	2294	9.29%	0.57	1691	10.23%	0.52
typos	463	1920	20.10%	0.76	1565	23.39%	0.77
ord	533	2858	26.66%	0.73	2312	27.81%	0.83
bazuka	285	1554	29.02%	0.62	1322	30.41%	0.73
strum	90	428	11.92%	0.16	368	13.58%	0.22

			Valid	Invalid	% Valid	Mut. Score
Overall	Regex	Mean	452.33	1392.33	23.21%	0.62
		Std.Dev.	309.05	656.76	12.16%	0.24
	Comby	Mean	554.00	1055.83	29.30%	0.65
		Std.Dev.	592.63	508.30	21.62%	0.25

Table 3. Regex vs. Comby for Python. LOC describes the project files considered. Muts. is total number of mutants generated for those files. Invalid mutants are ones that fail module load. We use TCE to check for redundant Python mutants, shown under "Red."

Project	LOC	Regex				Comby			
		Muts.	Red.	% Valid	Mut. Score	Muts.	Red.	% Valid	Mut. Score
keract	402	3984	729	41.53%	0.28	3067	533	47.96%	0.26
dtw	118	2106	567	46.20%	0.52	1835	639	47.19%	0.55
notion-sdk-py	189	1038	82	18.89%	0.57	1019	122	19.23%	0.60
atlassian-python-api	236	1511	207	19.99%	0.64	1319	351	25.25%	0.62
ESD	698	6268	1642	29.46%	0.07	3874	614	44.18%	0.07
fiber	358	1565	163	31.08%	0.62	1466	241	31.24%	0.64

			Red.	Valid	Invalid	% Valid	Mut. Score
Overall	Regex	Mean	565.00	913.00	1267.33	31.19%	0.45
		Std.Dev.	584.79	708.93	813.35	11.06%	0.23
	Comby	Mean	416.67	839.33	840.67	35.84%	0.46
		Std.Dev.	211.62	628.79	418.99	12.28%	0.24

The mean (and median) percentage of mutants that are valid is higher for Comby for all languages. On average, even for a single modestly sized file, using parser parser combinators saves 100-600 failed compilations.

The number of valid mutants is usually slightly smaller for Comby, but Comby actually produces a larger mean (but not median) number of mutants for Rust, due to one project where Comby produces a much larger number of valid mutants. The small gain in valid mutants for Regexp is partly due to Regexp (unfortunately) mutating some comments, and partly due to a reported, but not yet fixed, bug in the Python front-end to Comby, affecting an operator that swaps literal list items. As discussed below, mutation scores and mutant equivalence rates are very similar for both modes; mean mutation scores differed by at most 0.3, and project scores by more only for Rust.

For Python, TCE on Python bytecode was possible. The number of redundant (equivalent to the original bytecode, or the bytecode of an already-generated mutant) mutants was overall similar for both Regex and Comby, though varied widely by file.

In sum, in addition to providing a major advantage in expressiveness (the ability to easily define complex multi-line mutations that preserve structure, e.g. going beyond statement deletion to block deletion), parser parser combinators offer a notable gain in the efficiency of initial mutant generation, with no apparent loss of overall effectiveness.

4.2 Universal Source-Based Mutation vs. Previous Approaches

We then attempted to compare the quality of the approximately 15K valid mutants generated by our tool using Comby to mutants generated by competing tools for the relevant languages. Table 4 shows the set of tools compared. We attempted to identify well-maintained and commonly used open source tools for each language chosen. The tools cover a number of paradigms; for the most part we did not choose bytecode based tools, since these often make it difficult or impossible to mutate a single source file. PIT, however, provides this functionality and is perhaps the paradigmatic bytecode based mutation tool. Only Dextool, for C++, implements the *mutant schema* [34] approach

Table 4. Overview of Mutation Testing Tools. Our tool has significantly fewer lines of code than the single-language tools.

Language	Tool	URL	LOC
Python	Mutmut	https://github.com/boxed/mutmut	3870
	cosmic-ray	https://github.com/sixty-north/cosmic-ray	4599
Java	PIT	https://github.com/hcoles/pitest	59577
	LittleDarwin	https://github.com/aliparsai/LittleDarwin	22359
Rust	Cargo-Mutants	https://github.com/sourcefrog/cargo-mutants	7020
C++	Dextool	https://github.com/joakim-brannstrom/dextool	38611
ALL	Universalmutator	https://github.com/agroce/universalmutator	2244

often advocated in the research literature, where a single metaprogram is produced and, e.g., environment variabes are used to control which particular version of a program actually executes. Using a schema has the major advantage of dramatically reducing the time required to compile mutants. We speculate, however, that the fact that the approach is somewhat complex to implement for some kinds of mutants, and furthermore fails if even one invalid mutant is generated, has dissuaded most developers of non-academic tools from adopting it.

The most striking thing about this table, and a strong support for the central argument of this paper, is that our implementation, which covers *all the languages considered plus a number of additional languages*, is *the smallest of the tools by a margin of over 1,000 LOC*, and can be extended to handle additional languages with minimal effort. Each currently supported language requires about 15 rules on average, though these must be implemented as both Regex and Comby rules if both modes are to be used for the language.

The central question is: does using a universal approach produce notably less effective mutation testing? Or, to put it another way, does the substantial effort required to develop and maintain a mutation tool for a single language pay off in obvious advantages in mutation testing results? Tables 5-8 show the results of mutation testing for the generated mutants for our tool, in Comby mode, and the above-listed tools for each of the four studied languages.

We do not report the time taken for mutation generation and testing. Dextool uses mutant schema, and PIT performs bytecode mutation, so both are much faster for compiling mutants, but other tools took similar amounts of time to compile mutants. Mutant execution time was also generally similar, except for tools that produced very few mutants. We expect that our approach is generally slower for initial mutant generation, due to a higher number of invalid mutants, but that this cost is relatively small compared to the time required to compile and execute valid mutants for almost all projects. An informal study of large Java projects by the author of the PIT gradle plugin [1] suggests that even a successful compile of a typical project takes somewhere between one-sixth to one-third the time required for executing tests, and failed compiles are usually faster than successful ones, since invalid mutants almost always fail during parsing, and so code generation, optimization, and linking are not performed.

We estimated mutant equivalence when possible by examining a sample of 30 randomly selected unkilled mutants for each file. For some tools, human inspection of the individual unkilled mutants was not possible. However, we were of course able to estimtate the percent of unkilled mutants that were equivalent for our two modes for all files, and these numbers can be put in the perspective of rates reported in the literature. Reports of equivalent mutant percentages in the literature sometimes

report the percent of *all* mutants (including killed mutants, which obviously cannot be equivalent) that are equivalent. We follow Schuler and Zeller [33] in reporting the percentage of *unkilled* mutants that were found to be equivalent. Schuler and Zeller found about 45% of unkilled mutants of Java programs to be equivalent. Other reports of lower rates, e.g, 9% [27] or 7% [29], report the percent of *all* mutants, and so map to much higher rates of equivalence over *unkilled* mutants. We suggest that as a fraction of unkilled mutants, a rate of 30% or fewer equivalent mutants is good, and a rate of 10% or below is very good.

Table 5. C++ (Comby vs. Dextool). We cannot measure equivalent mutants for Dextool.

Project	Comby			Dextool	
	Muts.	Mut. Score	% Eqv.	Muts.	Mut. Score
ompl	564	0.07	26.7%	278	0.08
libgraphqlparser	131	0.90	33.3%	52	0.96
rethinkdb_rebirth	492	0.10	12.7%	25	0.36
xoreos	2280	0.01	3.3%	414	0.02
libxmljs	790	0.64	6.7%	830	0.46
tiny-differentiable-simulator	770	0.04	0%	61	0.21
Mean	837.83	0.29	13.67%	276.67	0.33
Std.Dev.	745.78	0.37	13.50%	311.56	0.38

Table 6. Java (Comby vs. PIT vs. LittleDarwin). LittleDarwin and PIT produce no equivalent mutants.

Project	Comby			PIT		LittleDarwin	
	Muts.	Mut. Score	% Eqv	Muts.	Mut. Score	Muts.	Mut. Score
pebble	85	0.81	3.3%	20	0.95	8	1.00
spring-hateoas	172	0.37	10.0%	114	0.28	38	0.39
javacc	494	0.21	3.3%	143	0.21	82	0.78
coffee-gb	174	0.23	0%	36	0.89	29	0.38
egads	597	0.71	20.0%	107	0.81	85	0.87
apk-parser	333	0.08	3.3%	89	0.16	33	0.09
Mean	309.17	0.40	7.9%	84.83	0.55	45.83	0.59
Std.Dev.	202.44	0.29	7.3%	47.60	0.37	30.93	0.35

Previous studies comparing mutation tools for Java [2, 9, 20] found that results for individual projects often varied considerably across tools without indicating a clear advantage for one tool over another. Our primary criteria for comparing tools is whether there is an overall pattern of clearly less-effective mutation for our approach. We only report results for Comby, as Comby and Regexp approaches produced nearly identical mutation scores, and had similar equivalence rates. In fact, Regexp equivalence rates were somewhat lower over all than those shown here. Full results for Regexp mutation execution are included in our replication package.

4.2.1 C++. We found only one tool that allowed single-file mutation and worked reliably for C++. Dextool generally produced fewer mutants than either mode of our tool, and for some projects produced an extremely small number of mutants. E.g., for the `tiny-differentiable-simulator`

Table 7. Python (Comby vs. Mutmut vs. CosmicRay)

Project	Comby			Mutmut			CosmicRay		
	Muts.	Mut. Score	% Eqv	Muts.	Mut. Score	% Eqv	Muts.	Mut. Score	% Eqv
keract	1471	0.26	6.7%	482	0.72	20.0%	921	0.27	3.3%
dtw	866	0.55	0%	270	0.44	3.3%	713	0.46	27.7%
notion-sdk-py	196	0.60	13.3%	63	0.65	0%	52	0.44	13.9%
atlassian-python-api	333	0.62	30.0%	174	0.71	13.3%	106	0.74	3.5%
ESD	1712	0.07	6.7%	720	0.08	13.3%	1013	0.06	6.7%
Fiber	458	0.64	23.3%	222	0.57	46.7%	320	0.81	20.0%
Mean	839.33	0.46	13.3%	321.83	0.53	16.1%	520.93	0.46	12.34%
Std.Dev.	628.80	0.24	11.4%	239.04	0.24	16.7%	417.50	0.28	9.57%

Table 8. Rust (Comby vs. Cargo-Mutants)

Project	Comby			Cargo-Mutants		
	Muts.	Mut. Score	% Eqv.	Muts.	Mut. Score	% Eqv.
cargo release	1690	0.80	0%	5	0.40	0%
passarine	173	0.52	26.7%	0	0.00	0%
typos	366	0.77	10.0%	6	0.83	0%
ord	643	0.83	20.0%	6	1.00	0%
bazuka	402	0.73	16.7%	0	0.00	0%
strum	50	0.22	6.7%	2	1.00	0%
Mean	554.00	0.64	13.3%	3.17	0.54	0.00%
Std. Dev	592.63	0.25	9.7%	2.86	0.47	0.00%

project, Dextool only produced a total of 61 mutants, and gave a much higher mutation score than our two modes. Our tools produced more than 600 additional unkilled mutants, and none of the inspected mutants were equivalent. Similarly, Dextool only found 2 unkilled mutants for the `libgraphqlparser` project. We doubt that the set of test weaknesses in the code is adequately represented by such a small number of mutants when our tools were able to produce significantly more unkilled mutants. Even assuming Dextool's mutants are never equivalent (possible though not necessarily likely) and supposing that a third of our generated mutants are equivalent, it seems likely that Dextool is missing important mutants in some cases. Overall, mean mutation scores were similar for Dextool and our tool, while median mutation scores were much lower for our tool, which produced many more unkilled mutants for some projects. While our equivalence rate was somewhat high for one project, it was generally good compared to results in the literature. For C++, there seems to be no reason to suspect that the universal approach is producing sub-par mutation testing.

4.2.2 Java. For Java, we compare to PIT, arguably the standard for mutation testing tools in real-world usage (and a tool very frequently used in the literature) as well as LittleDarwin, a source-based tool. Our tool produces lower but comparable mutation scores, and usually acceptable or excellent equivalence rates. LittleDarwin produces *no* equivalent mutants, but is a highly conservative

tool, only implementing a small set of operators, and not including statement deletion. PIT also produced no equivalent mutants; the limited set of operators applicable at the bytecode level tends to produce non-equivalent mutants by nature. Our tools always produced more non-equivalent unkilled mutants than PIT by a large margin. Our equivalence rates were uniformly low, with a median of only 3.3% equivalent mutants.

4.2.3 Python. Our comparison of Python mutation tools again showed that there is no evident disadvantage to using our approach, in terms of mutation testing results. Comby, with one exception, achieved a very good equivalence rate. Again, discarding equivalent mutants, our tool produced more mutants, and more unkilled mutants, than other tools, but still had a similar overall mutation score (the only clear outlier in scores is Mutmut's troublingly high score for the keract project).

4.2.4 Rust. Finally, for Rust, we observed that the Cargo-Mutants tool simply does not generate many mutants for most source files. Mutation scores were broadly similar to ours, though lower, and none of the generated mutants we examined were equivalent, but for two projects Cargo-Mutants did not produce *any compilable mutants* even though our tool was able to produce a large number of non-equivalent mutants. Our tool did produce a moderately high percentage of equivalent mutants for two projects, but these numbers are still well within the range of expected results in past inspections of equivalent mutants. We also note that for Rust the cost for invalid mutants is unusually low, due to the availability of `cargo check` which very quickly checks that code compiles, without actually compiling it, making our approach more attractive. Some C++ compilers also offer fast syntax check facilities, e.g., `-fsyntax-only` in g++.

4.2.5 Discussion: Summary of Tool Comparisons. Our tools produce somewhat lower mutation scores than other tools, and higher (though still generally good) equivalence rates than some tools, but clearly fall into the same broad similarity of behavior reported for high-quality mutation tools in the literature comparing Java mutation tools. The tools with very low or zero equivalence rates, furthermore, are very conservative in mutation generation, and often simply don't produce many mutants. In fact, inspection of the non-comment source lines as measured by CLOC as compared to results as shown in Tables 2, Tables 6 and 8 shows that for many programs, LittleDarwin and Cargo-Mutants generated *much less than one mutant per actual line code.* Such a conservative approach is likely to produce few or no actionable (unkilled and non-equivalent) mutants for a program with even a moderately high-adequacy test suite. But such suites are arguably the ones where mutation testing offers the most promise, by allowing the detection of very subtle remaining holes in tests for high-criticality software.

We speculate that our somewhat lower scores and possibly higher equivalence rates derive from the simple fact that *we generate substantially more* mutants than most tools, even after excluding equivalent mutants. In fact, for every single language studied, our tool generated the largest number of mutants, without a compensatingly higher equivalence rate, or a very much lower mutation score. Our mean number of generated valid, non-equivalent mutants (estimating based on our sample) was more than a factor of two larger than the nearest competing tool for all languages except Python, and was still more than 300 mutants larger than any other tool for Python.

The ease of defining mutation operators, plus our interest in generating large numbers of mutants and prioritizing the most interesting, in order to make mutation testing useful for projects with extremely high quality test suites, or even using projects using formal verification (as in the work of Groce et al. [13]), has encouraged us to take a "let a thousand flowers bloom" approach to choosing mutation operators. For example, our approach provides many more control-flow mutations than most tools, due to our observation that test suites sometimes fail to check for loop execution counts other than one and zero. Few other tools introduce `break` and `continue` statements into programs,

```
1  qsort :: [ Int] -> [ Int ]
2  qsort [] = []
3  qsort (x: xs ) = qsort l ++ [x] ++ qsort r
4      where l = filter (< x) xs
5            r = filter ( >= x) xs
```

Fig. 1. Example from MuCheck Paper [21]

or transform one into the other. These mutants are often equivalent (in fact, they give rise to a large number of our equivalent mutants reported above), but *also sometimes expose subtle weaknesses in a high quality test suite*, in our experience. In general, we take our philosophy of mutation from the proposal by Gopinath et al. [10, 11] that expanding the set of mutation operations and classes of such operations, in the long run, is more important than using only a small set of "good" operators, in that the end-goal of mutation testing is to identify weaknesses in test suites, not simply to estimate a single score. The price paid for this expansiveness in allowing ways to detect weaknesses in testing is a larger number of mutants to execute, and a slightly larger number of equivalent mutants. However, this is not a fundamental limitation or forced choice. We propose as future work to split the operators for each level of the hierarchy into core operators and a set of "aggressive" operators that includes our full current set of operators.

In fact, this brings us back to a point made in the introduction. While our approach introduces some unavoidable costs in the (relatively cheap, compared to execution and analysis) compilation stage of mutation testing, it may offer *more* flexibility in addressing execution and analysis stages. Defining limited custom sets of operators to run is trivial in our setting, making it easy to execute only, e.g., statement mutations, or, in a numeric-computation intensive codebase, only arithmetic operators. Expanding the hierarchical structure to include "famillies" of mutation operators is trivial, and gives users great control over mutation. Our approach produces source copies of all mutants, making it easy to apply techniques like Predictive Mutation Testing [36] that work with source code. This also makes it easy to integrate, as we already have done, heuristic aids to mutant *analysis* such as ranking of unkilled mutants by similarity and "interstingness," an approach that has already proven useful in using mutation testing to improve static analysis tools [12].

4.3 Haskell Case Study

Figure 1 shows a simple Haskell program, the motivating example in the paper presenting the MuCheck [21] mutation testing tool for Haskell. As noted in the introduction, to our knowledge MuCheck is the first (and perhaps only) mutation tool for Haskell code. MuCheck has not been updated since 2015. To see how our approach fares when mutating code in a language for which 1) it has no specific rules and 2) the assumption of a simple C-like syntax does not hold, we applied our implementation to this example.

Using regular expressions, and only the universal rules, our tool generates 10 valid mutants and 34 invalid mutants of the code (22.73% valid mutants). Comby produces an identical 10 valid mutants, but 65 invalid mutants (13.33% validity). Both tools produce a large number of invalid mutants here due to including rules such as adding `break` and `continue` that are sufficiently widespread in applicability to be included in our "universal" set but do not apply to Haskell; we believe that Comby's awareness of Haskell syntax in this case actually gives it *more* opportunities to apply such "inappropriate" mutants.

These mutants produce a mutation score of 0.8 for the two properties defined in the original MuCheck paper. In other words, like MuCheck, our tool is able to determine that the properties

provided are insufficient. In particular, while they check idempotence and that the result is sorted, they do not check that the result of `qsort` is a permutaton of the input list. The MuCheck paper only provides mutation scores for the two properties independently. For idempotence, MuCheck gives a mutation score of 0.84, and our implementation yields a score of 0.8. Checking only sortedness, MuCheck yields a score of 0.61, while our approach produces an even lower score of 0.4. Our lower scores are not due to equivalent mutants: adding a property to check that the sorted list is a permutation of the original list kills all our mutants. Our implementation does not provide the small set of Haskell-specific mutations provided by MuCheck (e.g., type-aware function replacement). It nonetheless produces clearly useful mutants, even for a language arguably radically different from those for which the universal rules were developed.

4.4 Comparison with GPT 4

```
[0..] ==> [1..]
IO () ==> IO String
"Current TODO list:" ==> "Updated TODO list:"
"Invalid command: " ==> "Unrecognized command: "
```

Listing 1. **Example of GPT-4 Mutants for a Simple Haskell Program**

```
head :[list] ==> last :[list]
tail :[list] ==> init :[list]
(:[expr1]) && (:[expr2]) ==> (:[expr1]) || (:[expr2])
(:[expr1]) >>= (:[expr2]) ==> (:[expr1]) >> (:[expr2])
```

Listing 2. **GPT-4 Comby Haskell Rules**

To understand how our approach compares with state of the art large language models (LLM), we prompted GPT-4 to generate a mutation testing tool for Haskell. Surprisingly, the generated program had no syntax errors, and even ran to produce a mutant. However, the program only implemented the inverting conditionals operation and some basic module mutations, a small subset of all the operators our tool implements, despite our tool being unaware of Haskell's existence. Asking the LLM to expand the set of rules produces more operators; however the number of mutants generated by such an approach is still a small subset of what our approach is capable of doing, even for a language we do not technically support.

Another approach is to have LLMs generate mutants for a given program. We experiment with prompting GPT-4 to generate mutants for a simple Haskell TODO-list management program. listing 1 shows the mutants that GPT-4 generated for this program (formatted as before ==> after). Notice that some of the mutants generated make sense, and resemble bugs a developer might make (replacing 0 with 1 or using the wrong data type). However, GPT-4 also generated mutants changing strings for all outputs to the console. These are not fruitful mutants; the messages (potentially) could be checked, but the arbitrary nature of such strings makes them poor mutants (which is why no mutation tools we are aware of implement such an operator).

While LLMs are not able to generate fully working language specific mutation testing tools from scratch, they are capable of generating rules in our formats that are specific to a given language. For example, prompting GPT-4 to generate Haskell rules for our tool generates the set of rules seen in listing 2. The generated rules are both valid Comby syntax and language-specific. E.g., replacing `head` with `tail` is a useful Haskell-specific rule. This suggests that while extending other tools to include new rules often involves writing complex code to manipulate ASTs, our tool may be

extensible simply by requesting an LLM to implement a natural language description of the desired rule(s), since GPT-4 already knows how to use Comby (and write regular expressions).

5 THREATS TO VALIDITY

Our study only examined 24 total source files, from 24 different projects, across four different languages, a total of about 9,000 lines of non-comment source code. None of these files were very large, though they were of sizes common to many source files in projects across GitHub, and none were less than about 100 non-comment lines of code. More importantly, there is no established way to compare mutation tools. Previous studies of the topic have generally simply expected that a useful tool must generate a significant number of non-equivalent unkilled mutants, and yield mutation scores that are broadly similar to those of other tools, both tests that our tool passes with ease (and some tools we compared to do not always pass).

6 RELATED WORK

Many approaches have been proposed to tackle the *computational* cost of mutation, including weak-mutation, meta-mutation, mutation-sampling, and predicting which mutants will be killed [19, 26, 34, 37]. Approaches to reducing the cost of mutation analysis were categorized as *do smarter*, *do faster*, and *do fewer* by Offutt et al. [28]. The *do smarter* approaches include space-time trade-offs, weak mutation analysis, and parallelization of mutation analysis. The *do faster* approaches include mutant schema generation, code patching, and other methods to make mutants run faster. Finally, the *do fewer* approaches try to reduce the number of mutants examined, and include selective mutation and mutant sampling. None of these approaches focus on the cost in *human* time to develop and maintain mutation testing tools. In fact, the complexity and sophistication of some of these approaches imposes a daunting barrier to those who would develop "good" mutation tools for a new language.

Hariri et al. compared C mutation approaches at the source and compiler IR levels [17] and found that overall source level mutation was better, producing fewer mutants overall, but matching the IR approach in the important measures of surface and minimal mutants and overall mutation score. Numerous studies compare Java mutation tools [9, 20], including a recent article for a more general audience in Communications of the ACM [2] (perhaps indicating a growing interest in practical mutation testing), which showed that users considered active maintenance, support for a variety of testing frameworks, and support for recent Java versions as the most important features in a Java mutation tool. The approach proposed in this paper by its nature tends to promise all three of these key factors without imposing burdens on maintainers. More recently, there has been some effort to create a language agnostic mutation testing tool [8]. However, this tool uses Wodel, a new (and hard-to-read, compared to Comby) DSL for expressing operators, and requires users to define a set of operators for each language, without sharing of rules. Groce et al. originally presented an initial prototype of our approach in a short tool paper, lacking parser-parser combinator support or a non-trivial empirical evaluation [14]. The universalmutator was used in previous work proposing mutation-based evaluation and improvement of static analysis tools [12], as a basis for the mutation approach (using Comby) adopted in a paper on improving turn-key compiler fuzzing [16], and as a basis for incorporating mutation generation in fuzzing [15], as well as in other work simply using it as a mutation tool.

7 CONCLUSIONS AND FUTURE WORK

We conclude that, by using a hierarchical, source-based approach, either relying on a pure-text regular expression definition of mutation operators or a richer parser parser combinator transformation definition, it is possible to implement effective, easily extensible mutation testing for essentially *all*

commonly used programming languages, and likely most not-so-commonly-used programming languages, in very few lines of code. Our approach fundamentally relies on the insight that program mutation is just a restricted instance of the general problem of program transformation, where transformations are highly syntactic (rather than semantics-aware) and highly localized.

As future work, we plan to give users more control over which mutation operators are used, with controls common to multiple languages. Our approach is a good fit to modern large software projects, which are often multilingual [18]: a single mutation tool can be used for an entire project.

8 DATA AVAILABILITY

We have made a full replication package available at https://figshare.com/s/7ab4bc9a156fec248528. See the README.md file included for details. The package contains Python scripts to run our data collection and experiments, as well as full source code for our tool.

9 ACKNOWLEDGMENTS

A portion of this work was funded by the National Science Foundation under NSF awards CCF-2129388, CCF-1910067, and CCF-2129446. We also thank all contributors to the Universalmutator and Comby projects.

REFERENCES

[1] [n. d.]. How fast (or slow) mutation testing really is? https://solidsoft.wordpress.com/2017/09/19/how-fast-or-slow-mutation-testing-really-is/.

[2] Domenico Amalfitano, Ana C. R. Paiva, Alexis Inquel, Luís Pinto, Anna Rita Fasolino, and René Just. 2022. How Do Java Mutation Tools Differ? *Commun. ACM* 65, 12 (nov 2022), 74–89. https://doi.org/10.1145/3526099

[3] Paul Ammann and Jeff Offutt. 2016. *Introduction to software testing*. Cambridge University Press.

[4] Moritz Beller, Chu-Pan Wong, Johannes Bader, Andrew Scott, Mateusz Machalica, Satish Chandra, and Erik Meijer. 2021. What It Would Take to Use Mutation Testing in Industry - A Study at Facebook. In *International Conference on Software Engineering: Software Engineering in Practice*. IEEE, 268–277. https://doi.org/10.1109/ICSE-SEIP52600.2021.00036

[5] Rudy Braquehais and Colin Runciman. 2016. FitSpec: Refining Property Sets for Functional Testing. In *Proceedings of the 9th International Symposium on Haskell* (Nara, Japan) *(Haskell 2016)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/2976002.2976003

[6] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. 2013. Taming Compiler Fuzzers. In *Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 197–208. https://doi.org/10.1145/2491956.2462173

[7] Lin Deng, Jeff Offutt, and Nan Li. 2013. Empirical evaluation of the statement deletion mutation operator. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE, 84–93.

[8] Pablo Gómez-Abajo, Esther Guerra, Juan de Lara, and Mercedes G. Merayo. 2021. Wodel-Test: A Model-Based Framework for Language-Independent Mutation Testing. *Softw. Syst. Model.* 20, 3 (jun 2021), 767–793. https://doi.org/10.1007/s10270-020-00827-0

[9] Rahul Gopinath, Iftekhar Ahmed, Mohammad Amin Alipour, Carlos Jensen, and Alex Groce. 2017. Does choice of mutation tool matter? *Software Quality Journal* 25 (2017), 871–920.

[10] Rahul Gopinath, Iftekhar Ahmed, Mohammad Amin Alipour, Carlos Jensen, and Alex Groce. 2017. Mutation reduction strategies considered harmful. *IEEE Transactions on Reliability* 66, 3 (2017), 854–874.

[11] Rahul Gopinath, Mohammad Amin Alipour, Iftekhar Ahmed, Carlos Jensen, and Alex Groce. 2016. On The Limits of Mutation Reduction Strategies. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. 511–522.

[12] Alex Groce, Iftekhar Ahmed, Josselin Feist, Gustavo Grieco, Jiri Gesi, Mehran Meidani, and Qihong Chen. 2021. Evaluating and Improving Static Analysis Tools Via Differential Mutation Analysis. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*. 207–218. https://doi.org/10.1109/QRS54544.2021.00032

[13] Alex Groce, Iftekhar Ahmed, Carlos Jensen, Paul E McKenney, and Josie Holmes. 2018. How verified (or tested) is my code? falsification-driven verification and testing. *Automated Software Engineering Journal* 25, 4 (2018), 917–960.

[14] Alex Groce, Josie Holmes, Darko Marinov, August Shi, and Lingming Zhang. 2018. An Extensible, Regular-expression-based Tool for Multi-language Mutant Generation. In *International Conference on Software Engineering: Companion*

Proceeedings (Gothenburg, Sweden) *(ICSE 2018)*. ACM, New York, NY, USA, 25–28. https://doi.org/10.1145/3183440.3183485

[15] Alex Groce, Goutamkumar Tulajappa Kalburgi, Claire Le Goues, Kush Jain, and Rahul Gopinath. 2022. Registered report: First, fuzz the mutants. In *International Fuzzing Workshop, ser. FUZZING*, Vol. 22.

[16] Alex Groce, Rijnard van Tonder, Goutamkumar Tulajappa Kalburgi, and Claire Le Goues. 2022. Making no-fuss compiler fuzzing effective. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction* (Seoul, South Korea) *(CC 2022)*. Association for Computing Machinery, New York, NY, USA, 194–204. https://doi.org/10.1145/3497776.3517765

[17] Farah Hariri, August Shi, Vimuth Fernando, Suleman Mahmood, and Darko Marinov. 2019. Comparing Mutation Testing at the Levels of Source Code and Compiler Intermediate Representation. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. 114–124. https://doi.org/10.1109/ICST.2019.00021

[18] Shin Hong, Byeongcheol Lee, Taehoon Kwak, Yiru Jeon, Bongsuk Ko, Yunho Kim, and Moonzoo Kim. 2015. Mutation-Based Fault Localization for Real-World Multilingual Programs (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 464–475. https://doi.org/10.1109/ASE.2015.14

[19] Samuel J. Kaufman, Ryan Featherman, Justin Alvin, Bob Kurtz, Paul Ammann, and René Just. 2022. Prioritizing Mutants to Guide Mutation Testing. In *Proceedings of the International Conference on Software Engineering (ICSE)*.

[20] Marinos Kintis, Mike Papadakis, Andreas Papadopoulos, Evangelos Valvis, and Nicos Malevris. 2016. Analysing and Comparing the Effectiveness of Mutation Testing Tools: A Manual Study. In *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 147–156. https://doi.org/10.1109/SCAM.2016.28

[21] Duc Le, Mohammad Amin Alipour, Rahul Gopinath, and Alex Groce. 2014. MuCheck: An Extensible Tool for Mutation Testing of Haskell Programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (San Jose, CA, USA) *(ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 429–432. https://doi.org/10.1145/2610384.2628052

[22] Richard J. Lipton, Richard A DeMillo, and Frederick G Sayward. 1978. Hints on test data selection: Help for the practicing programmer. *Computer* 11, 4 (1978), 34–41.

[23] Aditya P Mathur. 2012. *Foundations of Software Testing*. Addison-Wesley.

[24] Giorgio Natili. [n. d.]. Mutation Testing at Scale. https://slides.com/giorgionatili/mutation-testing-at-scale.

[25] A Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H Untch, and Christian Zapf. 1996. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 5, 2 (1996), 99–118.

[26] A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H. Untch, and Christian Zapf. 1996. An Experimental Determination of Sufficient Mutant Operators. *ACM Trans. Softw. Eng. Methodol.* 5, 2 (apr 1996), 99–118. https://doi.org/10.1145/227607.227610

[27] A Jefferson Offutt and Jie Pan. 1997. Automatically detecting equivalent mutants and infeasible paths. *Software testing, verification and reliability* 7, 3 (1997), 165–192.

[28] A Jefferson Offutt and Roland H Untch. 2001. Mutation 2000: Uniting the orthogonal. In *Mutation testing for the new century*. Springer, 34–44.

[29] Mike Papadakis, Yue Jia, Mark Harman, and Yves Le Traon. 2015. Trivial Compiler Equivalence: A Large Scale Empirical Study of a Simple Fast and Effective Equivalent Mutant Detection Technique. In *International Conference on Software Engineering*.

[30] Goran Petrovic and Marko Ivankovic. 2018. State of mutation testing at google. In *International Conference on Software Engineering: Software Engineering in Practice*, Frances Paulisch and Jan Bosch (Eds.). ACM, 163–171. https://doi.org/10.1145/3183519.3183521

[31] Goran Petrovic, Marko Ivankovic, Bob Kurtz, Paul Ammann, and René Just. 2018. An Industrial Application of Mutation Testing: Lessons, Challenges, and Research Directions. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 47–53. https://doi.org/10.1109/ICSTW.2018.00027

[32] Alessandro Viola Pizzoleto, Fabiano Cutigi Ferrari, Jeff Offutt, Leo Fernandes, and Márcio Ribeiro. 2019. A systematic literature review of techniques and metrics to reduce the cost of mutation testing. *Journal of Systems and Software* 157 (2019), 110388.

[33] David Schuler and Andreas Zeller. 2013. Covering and Uncovering Equivalent Mutants. *Software Testing, Verification and Reliability* 23, 5 (2013), 353–374. https://doi.org/10.1002/stvr.1473 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.1473

[34] Roland H. Untch, A. Jefferson Offutt, and Mary Jean Harrold. 1993. Mutation Analysis Using Mutant Schemata. *SIGSOFT Softw. Eng. Notes* 18, 3 (jul 1993), 139–148. https://doi.org/10.1145/174146.154265

[35] Rijnard van Tonder and Claire Le Goues. 2019. Lightweight Multi-Language Syntax Transformation with Parser Parser Combinators. In *Conference on Programming language Design and Implementation (PLDI '19)*.

[36] Jie Zhang, Ziyi Wang, Lingming Zhang, Dan Hao, Lei Zang, Shiyang Cheng, and Lu Zhang. 2016. Predictive Mutation Testing. In *International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 342–353. https://doi.org/10.1145/2931037.2931038

[37] Jie Zhang, Ziyi Wang, Lingming Zhang, Dan Hao, Lei Zang, Shiyang Cheng, and Lu Zhang. 2016. Predictive Mutation Testing. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 342–353. https://doi.org/10.1145/2931037.2931038

Received 2023-09-27; accepted 2024-01-23