An Exploration of the Syntactic Neighborhood of Mergesort

Iván Arcuschin Martin Monperrus

December 2015

1 Introduction

The syntactic neighborhood of a program p consists of all programs that differ from p by a small amount of changes [2]. In this paper, we consider the closest possible neighborhood: all programs that differ from p by a single character.

We are interested in the behavior of the programs that are in the the syntactic neighborhood: are they uncompilable programs, fail-fast programs, etc.? In particular, are there correct programs in the syntactic neighborhood?

Our key idea is to exhaustively explore and execute all programs that differ by one single printable character. This model represents a syntactic neighborhood. This research is very much exploratory because we have a very little understanding of this concept of "syntactic neighborhood" of programs. However, it is key for:

- automated program repair: are there patches in the syntactic neighborhood? how to define a syntactic neighborhood that maximizes the number of patches?
- automated program evolution and genetic programming: one kind of program evolution is to perform a sequence of steps in the successive syntactic neighborhoods.
- software security: many exploits consist of changing the program with code injection. Most exploits only enable to inject a small payload. It means that attackers leverage on the fact that many interesting programs exist in the syntactic neighborhood.

This model is not meant to represent a likely fault (whether a typo or a bit-flip) but rather is meant to improve our understanding of "syntactic neighborhood".

In this paper, we present the results of the complete exploration of the syntactic neighborhood of Mergesort implemented in two programming languages: C and Charity.

2 Experiment

2.1 Procedure

The experiment consists oi:

1. We take a program p
2. We create all possible programs that differ by a single printable character.
3. We generate n inputs for p
4. We execute the original programs and all mutants on all inputs
5. We compare the output of the mutated program against the output of the original program. This is bit-to-bit comparison, if the output is the same, the computation is fully correct for the considered input.

2.2 Character-level mutation

We explore the syntactic neighborhood by creating mutant programs at the level of single characters as follows Given the following set of printable chars,

{ !#$%&'()*+,-.:;¡=¿?@{|}~[\]^_‘0123456789abcdefghijklmnopqrstuvwxyz }

for each character in the original source code, we create one mutant per alternative character from the printable char set.

This procedure is given in the following algorithm:

```
Input: program p

Init mutants = []
Init source_code = base program split by end lines

# explore all lines
For line in source_code:
    # do not mutate preprocessor, assert statements or empty lines
    If line starts with '#', 'assert', or is empty:
        continue with next line
    # explore all chars in line
    For char in line:
        # do not change whitespaces
        If char is whitespace:
            continue with next char
        For replacement in printable_chars:
            If char == replacement:
                continue with next replacement
            # create new mutant
            Init new_mutant = source_code
            Put new_mutant[line] = source_code[line] with char replaced
            Add new_mutant to mutants

return mutants
```

Figure 1: Algorithm to Exhaustively Explore All Syntactically-Close Programs at the Character Level

We always use a minified version of the program, with the least number of white spaces (incl. tab). White spaces are not mutated: the rationale is that replacing any of them would obviously break tokenization.

After executing the mutated programs on all n inputs, we count the following cases

- Compiler error: mutants that fail to compile.
- Runtime error: mutants that compiled successfully, but ended with error the execution, i.e: Segmentation Fault. Infinite loops are broken by a 1s timeout and are considered as runtime errors.
- Incorrect output: mutants that compiled and executed successfully, but whose output do not match the output of the reference implementation on at least one input.
- Correct output: mutants that compiled and executed successfully, and the output is fully correct for all considered inputs.

Analysis Recall that this model is not meant to be fault model. Our goal is only to gain understanding of the concept of syntactic neighborhood. Yet, this syntactic neighborhood model has the following properties: 1) it contains the programs caused by programmer typos when the replaced character is physically close the original character on a given keyboard layout. 2) it

contains the programs slightly modified by transmission errors when programs are copied. over the network (as Javascript code in your browser).

To our knowledge, the closest neighborhood model in the literature is by Spinellis [3]: *"Random Character Substitution — RandCharSub: A single randomly chosen character (byte) within a randomly chosen token is substituted with a random byte"*. There are two main differences: first, we only consider printable bytes, second, and this is the most important, we exhaustively explore the syntactic neighborhood (as opposed to randomly).

2.3 Experiment #1: Results with Mergesort in C

Research question: What is the outcome of programs in the syntactic neighborhood of Mergesort in C?

We consider an implementation of Mergesort in C. The minified program contains 401 characters over 3 functions. we generate 20 integer arrays of size 10 with each integer randomly chosen in range [1,30]

There are 26408 character-level mutants. The huge majority of mutants do not compile. 478 programs yield a runtime error, 1190 programs produce an incorrect output. We observe 57 mutant programs (0.21%) that produce fully correct outputs for the 20 considered arrays.

	Compiler err.	Runtime err.	Incorrect out.	Correct out.	Total
Mergesort	24683 (93.47%)	478 (1.81%)	1190 (4.51%)	57 (0.21%)	26408 (100%)

Table 1: The profile of programs in the syntactic neighborhood of an implementation of Mergesort in C

2.4 Experiment #2: Comparison between C and Charity

Research question: What of programs in the syntactic neighborhood of a non Turing-complete language?

Our intuition, inspired from Wray [4] is the following:

> Assuming two programming languages A and B, if a language A is computationally richer then B, a small modification in a program written in A yields more valid behavior than a small modification in a program written in B.

We consider an implementation of Mergesort in the Charity language [1]. Charity an experimental purely functional non-Turing complete programming language, developed at the University of Calgary. The program has 390 characters. We also consider the same input arrays as for Experiment #1.

There are 25691 character-level mutants. There are no compiler errors since Charity is interpreted.

Finding #1: there are 27 (0.1%) mutated programs with fully correct output, which is half the number of correct mutated C programs.

Finding #2: there are only 28 (0.11%) mutated programs with yield an incorrect output. This is dramatically less than for the considered C program. This is a very good property: it is very easy for developers to spot errors thanks to runtime errors, on the contrary it is much harder to spot incorrect outputs. To that extent, the syntactic neighborhood of Charity programs is much less dangerous than the one of C programs.

	Compiler err.	Runtime err.	Incorrect out.	Correct out.	Total
Mergesort (C)	24683 (93.47%)	478 (1.81%)	1190 (4.51%)	57 (0.21%)	26408 (100%)
Mergesort (Charity)	-	25636 (99.78%)	28 (0.11%)	27 (0.10%)	25691 (100%)

Table 2: Comparison of C (Turing complete) and Charity (Non Turing-complete, stronger type-system)

Next we compute the number of locations, called spots, that can be modified at the character level with at least one equivalent mutated program. There are 3 such characters in the Charity program while there are 29 of them in the C program. Again, the syntactic neighborhood of Charity programs is more clear-cut than that of C programs.

2.5 Manual Analysis

Given the short amount of Charity mutants (27) that ended up with a fully correct output, we manually analyze them. They fall in 2 different cases:

1. Changing

   ```
   fold_cobTreex => x,(_, p) => mergelt_A p
   ```

   to

   ```
   fold_cobTreex => x,(a, p) => mergelt_A p
   ```

   Where the "_" was also successfully replaced to: $\{a, b, \ldots, o, q, \ldots, z\}$. In those 25 cases, this new variable is never used, which explains the semantic equivalence.

2. Changing

   ```
   build_m_tree l,(len,(delist:ff)),len)
   ```

   to

   ```
   build_m_tree l,(ten,(delist:ff)),len)
   ```

   or

   ```
   build_m_tree l,(len,(delist:ff)),ten)
   ```

   Where the "ten" is a special symbol in Charity, that represents the number 10. The programs are equivalent because the input arrays are of size 10. The other case is with the second "len".

2.6 Replication

The experimental material is available at:
`https://bitbucket.org/iarcuschin/syntactic-neighborhood-mergesort-artifact/`

3 Threats to Validity

General threats:

- Do those findings only hold for Mergesort? This can be solved by adding new programs.
- Do those findings only hold for the considered inputs? This can be solved by considering more and larger arrays as input.
- Are those findings based on Charity really due to non-turing completeness? This can be solved by adding other general purpose languages and non-turing complete programming languages.

References

[1] Robin Cockett and Tom Fukushima. *About charity*. Tech. rep. University of Calgary, 1992.

[2] A. Jefferson Offutt and J. Huffman Hayes. "A Semantic Model of Program Faults". In: *SIGSOFT Softw. Eng. Notes* 21.3 (1996), pp. 195–200.

[3] Diomidis Spinellis, Vassilios Karakoidas, and Panos Louridas. "Comparative language fuzz testing: programming languages vs. fat fingers". In: *Proceedings of the ACM 4th annual workshop on Evaluation and usability of programming languages and tools*. 2012, pp. 25–34.

[4] Stuart Wray. "Bugs with long tails". 2013.