



Fuzzing as Editor Feedback

Marcel Garus  

Hasso Plattner Institute, University of Potsdam, Germany

Jens Lincke  

Hasso Plattner Institute, University of Potsdam, Germany

Robert Hirschfeld  

Hasso Plattner Institute, University of Potsdam, Germany

Abstract

Live programming requires concrete examples, but coming up with examples takes effort. However, there are ways to execute code without specifying examples, such as fuzzing. Fuzzing is a technique that synthesizes program inputs to find bugs in security-critical software.

While fuzzing focuses on finding crashes, it also produces valid inputs as a byproduct. Our approach is to make use of this to show examples, including edge cases, directly in the editor.

To provide examples for individual pieces of code, we implement fuzzing at the granularity of functions. We integrate it into the compiler pipeline and language tooling of *Martinaise*, a custom programming language with a limited feature set. Initially, our examples are random and then mutate based on coverage feedback to reach interesting code locations and become smaller.

We evaluate our tool in small case studies, showing generated examples for numbers, strings, and composite objects. Our fuzzed examples still feel synthetic, but since they are grounded in the dynamic behavior of code, they can cover the entire execution and show edge cases.

2012 ACM Subject Classification Software and its engineering → Development frameworks and environments; Software and its engineering → Empirical software validation; Software and its engineering → Functionality

Keywords and phrases Fuzzing, Example-based Programming, Babylonian Programming, Dynamic Analysis, Code Coverage, Randomized Testing, Function-Level Fuzzing

Digital Object Identifier 10.4230/OASICS.Programming.2025.8

Supplementary Material *Software (Source Code)*: <https://github.com/MarcelGarus/martinaise> [5], archived at `swb:1:dir:8a3acf542aa172ae7ccc3a1384b6fd5303d42f61`

1 Introduction

Concrete examples help programmers understand abstract source code. Several practices, such as unit tests, technical documentation, and tutorials, use examples to illustrate the meaning of programs. Examples can also increase the confidence that a program behaves correctly for edge cases. However, coming up with good examples for edge cases is often overlooked and happens later, if at all.

Bugs introduced by unhandled edge cases can be difficult to fix because the original context, such as familiarity with the code and intent, is lost. Static analyses (such as linters and type checkers) and dynamic analyses (such as tests and fuzzing) can proactively find bugs. One of these techniques, fuzzing, automatically tests code with synthetic inputs to find crashes and has found many security-critical bugs in open-source projects.

We propose using fuzzing to generate examples while editing code. We built a prototype that works in a standard IDE for a custom language with a limited feature set, making it easy to fuzz. Examples found through fuzzing automatically appear in the editor. Even though the synthetic examples are often not typical and may even be unpleasant, these examples illustrate edge cases, help cover the code base, and can help find bugs, e.g., by illustrating uncaught errors.



© Marcel Garus, Jens Lincke, and Robert Hirschfeld;
licensed under Creative Commons License CC-BY 4.0

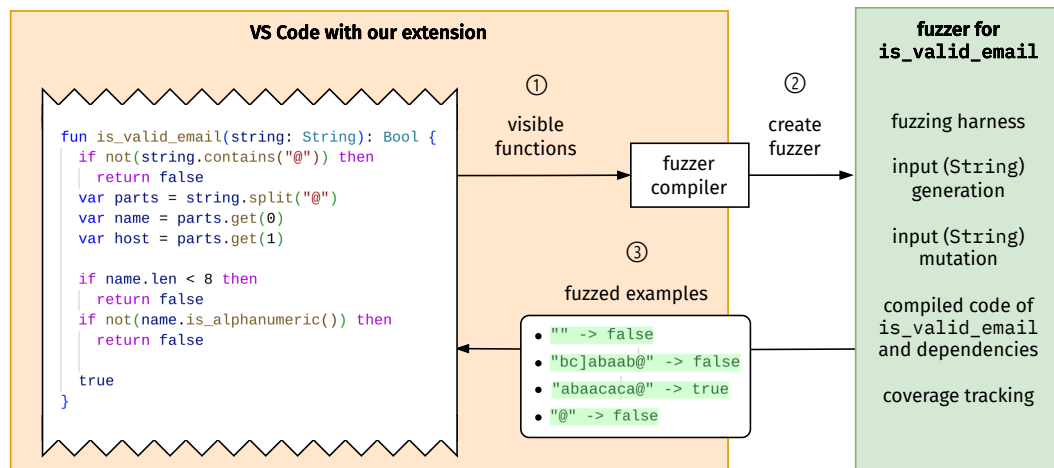
Companion Proceedings of the 9th International Conference on the Art, Science, and Engineering of Programming (Programming 2025).

Editors: Jonathan Edwards, Roly Perera, and Tomas Petricek; Article No. 8; pp. 8:1–8:15



OpenAccess Series in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



■ **Figure 1** Our extension tracks which functions are visible (1), creates and runs specialized fuzzers for them (2) and uses examples with different coverage to showcase the entire function behavior (3).

Our contributions are:

- We use techniques from fuzzing and property-based testing to find examples, not just bugs.
- We built an IDE extension that continuously fuzzes the visible code and shows examples.
- We fuzz individual functions with structured inputs, enabling our tool to quickly cover the entire code base.
- We minimize examples to make them less overwhelming to developers. We discuss the shortcomings of this approach for finding helpful examples.

The rest of the paper is structured as follows: Section 2 briefly introduces property-based testing and fuzzing and motivates the need for examples as editor feedback. Section 3 introduces our approach that fuzzes functions and shows the result directly in the editor. Section 4 describes the implementation for the Martinian language and the Visual Studio Code editor. Section 5 evaluates the quality and performance of our tool with small case studies. Section 6 discusses ideas for future work and how to adapt the approach to other languages. Section 7 concludes.

2 Background and Related Work

Over the years, the software engineering industry has adopted several practices that help write robust code. Static tools such as type checkers or linters analyze the code without executing it. Dynamic tools such as unit tests or debuggers execute the code to find other errors. Some dynamic tools, such as property-based testing and fuzzing, reduce the amount of setup required by automatically generating inputs.

2.1 Property-Based Testing

Property-based testing frameworks generate random inputs. Your tests receive those inputs as arguments and can then test properties of your code that should be true for any input [16]. For example, to test a function that calculates the maximum of a list of numbers, you could write a test that receives a list of numbers as input, calls the maximum function, and verifies properties of the result. Properties of the maximum function could be that the original list has to contain the result, or that the result is larger than or equal to every list item.

Property-based testing frameworks exist for many languages, such as QuickCheck¹ for Haskell. They usually work in two phases:

1. In the exploration phase, the framework runs the test with increasingly more complex inputs. If one of those inputs causes the test to fail, it enters the shrinking phase.
2. In the shrinking phase, the framework tries smaller versions of the failing input. It repeatedly shrinks the input until any further reduction causes the test to succeed. Finally, it reports the smallest input that still fails the test.

To generate correctly-typed inputs, property-based testing frameworks usually require developers to implement code for generating inputs, although this can sometimes be automated.

Compared to unit tests (which only test a single input-output pair), property-based tests are more abstract and can test code more thoroughly without much more effort.

2.2 Fuzzing

Fuzzing is the testing of programs with random inputs to find crashes. Historically, fuzzing evolved from an outside-in view, analyzing the robustness of other people's programs [14]. Today, fuzzing makes security-critical software, such as firmware, operating systems, browsers, and applications, more robust² [26, 24, 27].

In this technique, a component called *fuzzer* generates bytes, runs the program with those inputs, and receives feedback about the program's behavior. Because fuzzing works with unstructured bytes, it is usually more feedback-driven than property-based testing out of necessity. State-of-the-art fuzzers can be differentiated in the amount of feedback they receive from the running program [3, 22, 19]:

Blackbox fuzzing The fuzzer only receives feedback about the program's externally visible behavior, such as whether it crashed and how long it was executed. The limited amount of feedback makes it difficult to reach interesting code, especially if the program validates its input.

Whitebox fuzzing The fuzzer accesses the program code (at the source or machine code level) and analyzes its structure. For example, it can use symbolic execution and a constraint solver to generate inputs that reach target locations. This produces high-quality inputs but introduces a considerable overhead for generating inputs, especially for code with complicated control flow.

Greybox fuzzing The fuzzer uses simple coverage feedback to judge the quality of inputs. For example, inputs leading deep into the program's logic are perhaps more interesting than ones only executing a few instructions. Greybox fuzzers mutate promising inputs – for example, by inserting or removing a few bytes from the input byte sequence – gradually exploring all code. Greybox fuzzing balances the quantity and quality of the input generation and is, therefore, more effective at finding bugs than the other two variants.

The most widely used open-source fuzzers are based on American Fuzzy Lop (AFL) [26], a Greybox fuzzer. Since its inception, several papers [2, 1, 3, 10, 25, 6, 4] have proposed additions to make fuzzing more effective, but most Greybox fuzzers follow AFL's general architecture:

¹ <https://hackage.haskell.org/package/QuickCheck>

² Some high-profile projects where fuzzing found bugs: Mozilla Firefox, Apple Safari, iOS kernel, sqlite, Linux ext4, Tor, PHP, OpenSSL, OpenSSH, LibreOffice, libpng, curl, GPG, OpenCV, zstd, MySQL, ... See also: <https://github.com/google/oss-fuzz>

8:4 Fuzzing as Editor Feedback

1. The fuzzer maintains a pool of inputs. Initially, this pool is empty or filled with examples explicitly given by the developer.
2. The fuzzer either generates a random byte sequence or picks and slightly mutates one from the pool.
3. The fuzzer runs the program, sending the input bytes into its standard input (stdin). The fuzzer observes the coverage and whether the program crashed.
4. If the input executes code that was not reached before, the input is added to the input pool. If the input crashes the program, it is reported.
5. Go to step 2.

While fuzzing and property-based testing evolved from different origins, their use cases overlap: Fuzzing tests the property that the program doesn't crash (although assertions in the code can reduce any property to a crash). A unique appeal of fuzzing among dynamic tools is that it requires no setup.

2.3 Motivation

Fuzzing and property-based testing are usually used as after-the-fact analyses to find bugs. This distances them from the development workflow, both temporally and spatially.

Using property-based testing requires conscious effort from the developer. They have to implement/derive code for generating typed values, and they have to manually write tests for properties.

Using fuzzing requires less setup. However, it is usually not integrated into the editor, but invoked from the command line. Fuzzing works on entire programs, generates inputs at the byte level, and can take a long time to find bugs deep within your code (far from the main function).

Both tools yield insights apart from a potential crash report: They create examples covering all the different parts of the program and showing edge cases. Making these examples readily available during programming sessions rather than as an after-the-fact analysis has not yet been explored. Perhaps the developer can see these examples directly while writing code?

2.4 Examples as Editor Feedback

Other works explore showing live examples next to the source code, a practice sometimes called Babylonian Programming [18, 17]. For example, in *Inventing on Principle* [21], Bret Victor presents an editor that lets you specify example values for function inputs in a pane next to the source code. The editor will then show the concrete values of all variables and how they evolve as the function executes. Other works of live examples in your code include Live Literals [20], the Babylonian Programming Editor [18], or projection boxes [8].

A major difference among these tools is how developers specify examples. Rauch et al. [18] distinguish between two methods:

Implicit examples are provided in the source code, thus requiring code modification.

Explicit examples are provided using a special syntax or separate UI.

Making the computer come up with examples adds a third method. Mattis et al. [12] explore using large language models (LLMs) to generate examples and tests automatically. However, the model sometimes hallucinates function arguments, especially if code is not sufficiently tested or used by other code – the exact cases where examples would be helpful.

A fundamental limitation of LLMs is that they approximate the runtime behavior of code internally [15, 9]. On the other hand, examples found through fuzzing are grounded in the actual semantics of the code.

Fuzzing is a largely untapped source of examples. In this paper, we bring those examples to the editor. Our prototype shows annotations based on these examples.

3 Fuzzed Function Examples

We built an editor extension that uses fuzzing to generate examples for functions in a statically typed language. As you browse and edit code, our extension shows example inputs for functions, derived through function-level fuzzing.

As soon as a function becomes visible in the editor, our extension creates a specialized fuzzer for that function, as seen in Figure 1. Because this functionality is integrated into the compiler, it can automatically derive code that allows the fuzzer to work with typed arguments, for example, to generate new values. If you prefer, you can customize this code. Our extension then runs the generated fuzzer, which uses coverage feedback to explore the whole execution space, similar to other Greybox fuzzers. It also uses coverage to select and print examples that show different behaviors of the function. The extension reads the fuzzer’s reports and transforms them into hints that are shown in the editor.

Our tooling wants to not only find crashes, but also provide examples that help developers understand the function behavior, including edge cases. However, “helpfulness” is difficult to formalize and optimize. So, like traditional fuzzers and property-based testing frameworks, our prototype minimizes inputs. Compared to raw, random examples, these smaller, less complex ones are usually easier to comprehend but still highlight edge cases.

In traditional fuzzing starting from the `main` function, arguments to inner functions are filtered by the calling code, but it is more computationally expensive or even impossible to reach some functions. Fuzzing individual functions instead of the whole program can result in examples that may never actually occur in practice, highlighting edge cases early and driving developers to specify their assumptions and code more defensively.

4 Implementation for Visual Studio Code in Martinaire

We want to explore whether fuzzing-based editor tooling is useful. Rather than creating a generic solution that works on the lowest common denominator of programming languages and editors, we limit our scope to a single language and editor.

4.1 Martinaire

We designed Martinaire³, a custom programming language that has some specific limitations that make it easy to fuzz:

- It has a simple, static type system and favors concrete types over abstract interfaces.
- It has no first-class functions capturing their lexical context, only top-level functions.
- It has a small compiler and virtual machine, which enable complete code instrumentation.
- It has basic tooling support for Visual Studio Code (VS Code), a popular code editor.

In section 6, we discuss how our approach can be adapted to real-world languages.

³ <https://marcelgarus.dev/martinaire>

4.2 Working With Arguments

The fuzzer needs to be able to generate, mutate, and analyze the function arguments. Our prototype creates code that allows the fuzzer to work with the argument types, but the behavior can be overridden if necessary.

The fuzzer can generate random values of a desired type:

- For structs, it generates a random value for each field.
- For enums, it chooses a random variant and generates a random payload.

Like many property-based testing frameworks, the code for generating values accepts a desired complexity. This allows the fuzzer to test a function with increasingly larger inputs and prevents infinite recursion for recursive types.

Only being able to generate values is not enough. Greybox fuzzers use evolution to be effective at exploring code. They randomly mutate the input and keep those inputs that cover new ground as the new baseline.

- For structs, the fuzzer changes a random field.
- For enums, the fuzzer either chooses a new variant or changes the payload.

This mutation works with a temperature: The higher the temperature, the more the mutated value differs from the original one. This allows the fuzzer to widen or narrow the search space of new inputs.

4.3 Tracking coverage

Fuzzing requires feedback about the code coverage. Similar to existing fuzzers like AFL, our fuzzer *instruments* the generated byte code to track the coverage: It rewrites conditional jumps, the only form of data-dependent control flow, so that they update a global coverage bitset. After running some code, this bitset automatically indicates which branches were chosen.

Instructions with external side effects also get replaced during instrumentation. Instructions that output data (such as printing or writing a file) get omitted. If a function reads external data (from the standard input or a file), the function is not fuzzed.

4.4 Fuzzing

Given the functionality for generating inputs, judging their complexity, tracking coverage, and mutating them, we can now implement Greybox fuzzing. We have demands that are somewhat different from those of existing Greybox fuzzers: We want to find crashes but also interesting inputs. The editor should show representative examples as soon as possible.

Our process is very similar to how AFL works [26]: The fuzzer maintains a pool of examples. Initially, these are randomly generated. The fuzzer picks an example, mutates it, runs it, and compares the coverage to the original one:

- If it covers less code, it is discarded.
- If it covers the same code but is smaller, the example replaces the current example.
- If it covers new code, it is added to the pool for further exploration.

This process results in a set of small examples that execute different parts of a function's code.

4.5 Fuzzing for Editor Tooling

We want to show examples in the editor based on fuzzing. As a proof of concept, we implemented that functionality for the Visual Studio Code IDE⁴ (VS Code). We created a VS Code extension that tracks the opened files and the scroll position, checks which functions are visible, and spawns fuzzers for these functions. As these fuzzers report their progress, the extension continuously updates the examples shown in the editor.

Currently, editing the code invalidates the fuzzed examples in that file. When writing code, the visible functions are continuously fuzzed.

5 Evaluation

To illustrate how our tool works, here is a function that averages a list of numbers:

```
fun average(list: List[Int]): Int { [] panics [0] -> 0
  list.sum() / list.len
}
```

The red and green hints at the end of the first line automatically appear when our extension is activated. The red hint informs us that the function crashes when applied to an empty list (this happens because it divides by zero). The green hint tells us that the average of a list containing a zero is zero.

Our tool can work with custom types and tries to find example inputs that cover the entire function. For the following evaluation function for mathematical terms, our tool implements code for generating and mutating random terms and produces examples displayed after the function definition:

```
enum Term {
  number: Int,
  add: Operands,
  subtract: Operands,
  multiply: Operands,
  divide: Operands,
}
struct Operands { left: &Term, right: &Term }

fun eval(term: Term): Int { 6 - 3 / 0 panics 5 * 0 -> 0 2 / 4 -> 0 0 + 3
  switch term
  case number(num) num
  case add(op) op.left.eval() + op.right.eval()
  case subtract(op) op.left.eval() - op.right.eval()
  case multiply(op) op.left.eval() * op.right.eval()
  case divide(op) op.left.eval() / op.right.eval()
}
```

As not all examples fit on the screen, here are all the examples you can see when scrolling to the right:

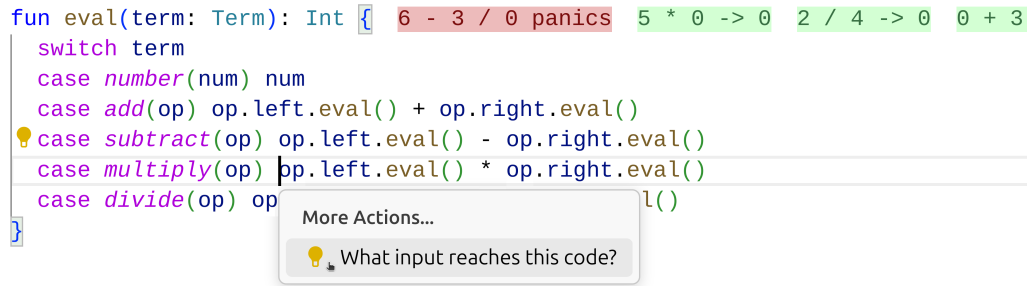
```
6 - 3 / 0 panics 5 * 0 -> 0 2 / 4 -> 0 0 + 3 - -3 -> 6 0 + 0 -> 0 0 -> 0
```

⁴ <https://code.visualstudio.com/>

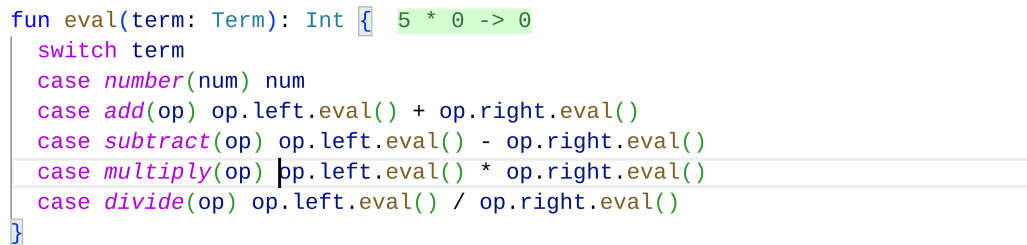
8:8 Fuzzing as Editor Feedback

The tool shows five examples of the five branches inside the function and another example that crashes the function.

In VS Code, one can use code actions to perform refactorings. Figure 2 shows how to use a code action to filter the examples to ones reaching a particular location. This does not change the fuzzing behavior, such as the directed fuzzing described by Böhme et al. [1]; it only changes which inputs are displayed.



(a) Code action asking for examples.



(b) Filtered fuzzing examples.

■ **Figure 2** Asking for examples that reach a given code location.

5.1 Quality of Examples

Coverage feedback is enough to explore functions with a straightforward control flow. Here is a function where the fuzzer successfully gives one example for each way the function can return:

```
fun is_valid_email(string: String): Bool {  "@" -> false  "" -> false  "b
  if not(string.contains("@")) then
    return false
  var parts = string.split("@")
  var name = parts.get(0)
  var host = parts.get(1)

  if name.len < 8 then
    return false
  if not(name.is_alphanumeric()) then
    return false
  true
}
```


These are all examples:

```
"@" -> false   "" -> false   "bc]abaab@" -> false   "abaacaca@" -> true
```

However, achieving full coverage on *any* function is impossible. For example, this function only returns an integer if the SHA-256 hash of the input matches an expected value:

```
var hash = "5E884898DA28047151D0E56F8DC6292773603D0D6AABBDD62A11EF721D1542D8"

fun secret_number(password: String): Maybe[Int] { "" -> none
    if sha_256(password) == hash then {
        | some(42)
    } else {
        | none[Int]()
    }
}
```

While, in theory, the fuzzer could generate and show an example input that reaches the conditional branch, doing so in a reasonable time frame requires breaking SHA-256.

In practice, our fuzzer already struggles with simpler code. For example, it does not reach the inner code in the following if clause:

```
if url.starts_with("https://") then {
    | ...
}
```

State-of-the-art fuzzers can fuzz this code because they also track the number of loop iterations, so inputs with a different prefix of "https" (such as "hello" and "htttt") will have different coverage in the string comparison code.

Apart from finding inputs that crash functions, the examples discovered by our tool can also be used to understand how a function behaves without looking at its implementation. However, while code coverage and input complexity generally correlate with how helpful examples are, we recognize that it is not a perfect proxy. In everyday use, we identify two major limitations when using our tool to understand existing code.

First, minimizing examples can be counterproductive. Previously, we showed that our tool gives [0] as an example for the **average** function. While a list containing a single number achieves full code coverage, it does not give the developer confidence that the implementation is correct. [1, 2, 3] would be a better example.

Second, the examples our tool shows are often unnatural. Here, our tool greets someone with an empty name:

```
fun greet(name: String): String { "" -> "Hello, !"
    | "Hello, {name}!"
}
```

While this is the expected behavior of fuzzing, in our experience, such unnatural examples introduce some mental overhead. This problem is difficult to address in a general way because the perceived complexity of values depends on the context: Even if 0 is perceived as less complex than 3 in isolation, most humans perceive [1, 2, 3, 4] as less complex than [1, 2, 0, 4].

■ **Table 1** Better examples over time for an email address checking function.

time (ms)	runs	screenshot of the examples in the editor
0	0	
989	1	<code>" -> false</code>
991	28	<code>" -> false</code> <code>"&EC]vFI>mWmFhh}@qwt-8&\$~%kA" -> false</code>
1002	82	<code>" -> false</code> <code>"&EC]vFI>mWmFhh}@qwt-8&\$~%kA" -> false</code> <code>"IFj@(`~B]G^6l[k</code>
2155	1140	<code>" -> false</code> <code>"lC]vFbmh@qta8Myme" -> false</code> <code>"IFj@(`~B]G^6l[kH[]H^aS:2?</code>
2177	1334	<code>" -> false</code> <code>"lCKvcafh@h" -> true</code> <code>"lC]vcafh@h" -> false</code> <code>"IFj@(`~B]G^</code>
2592	2409	<code>" -> false</code> <code>"lCKvcafh@h" -> true</code> <code>"bd]ccaab@" -> false</code> <code>"IFj@(`~B]G^</code>
3450	3794	<code>" -> false</code> <code>"bd]abaab@" -> false</code> <code>"lCKvcafh@h" -> true</code> <code>"IFj@(`~B]G^</code>
3664	4236	<code>" -> false</code> <code>"lCKvcafh@h" -> true</code> <code>"bc]abaab@" -> false</code> <code>"IFj@(`~B]G^</code>
4906	5272	<code>"@" -> false</code> <code>" -> false</code> <code>"lCKvcafh@h" -> true</code> <code>"bc]abaab@" -> false</code>
7183	9367	<code>"@" -> false</code> <code>" -> false</code> <code>"bc]abaab@" -> false</code> <code>"abaacaca@" -> true</code>

5.2 Performance

Fuzzing takes time. Compared to traditional fuzzing, an advantage of fuzzing on the granularity of functions as opposed to entire programs is that the fuzzer does not have to reach potentially crashing code indirectly through the `main` function. For example, if the entry part of the code is guarded with code that is difficult to pass such as the SHA-256 code from section 5.1, our tool can still fuzz internally used functions in isolation.

Unlike type-checking or linting, the output of fuzzing changes over time. Table 1 shows how examples for the email address check from section 5.1 change on a desktop computer⁵. Every line shows a timestamp, the number of function executions that were performed, and a screenshot of the examples. In the beginning, there are no examples. After the function is compiled, our tool quickly discovers four random-looking examples that cover all the code. Then, it shrinks them. The table only shows a subset of all updates – in total, the fuzzer produces many more intermediate versions, most of which differ only by a single character from the previous version. In the end, you are left with four small, representative examples of the function behavior.

Because of this continuous refinement, the feedback feels reasonably fast in practice. Even for large functions, the initial examples appear quickly – it just takes longer for the examples to stabilize.

5.3 Displaying Examples

Apart from the quality of examples, we also want to look at how and where examples are displayed. Currently, our tool formats examples as text and shows them next to the function signature. For many values, there are more appropriate representations than formatted text, such as visualizations. It may make sense to expand our tool to other editors that do not have VS Code’s constraints like the ones discussed by Rauch et al. [18].

⁵ The computer has an AMD Ryzen™ 7 5800X × 16 and 32 GiB of RAM. It runs Ubuntu 24.04.1 LTS.

There are some practical limitations of our implementation: We currently don't order examples. Also, for functions that mutate the input or produce a side effect, input-output pairs do not help understand their behavior.

6 Future Work

We showed that fuzzing can be a new source of information in the editor. There are many paths for further exploration.

6.1 Ideas for Improvements

Better performance

Implementing fuzzing for a real-world language with an optimizing compiler will improve the performance. Using a just-in-time compiled language or an incremental compiler could improve the latency between editing the code and examples updating. The same incremental caching could be used to only fuzz functions if they change.

Better fuzzing

State-of-the-art fuzzers track more information, such as the number of iterations in a loop; this makes it possible to explore more code. Various other aspects of fuzzing research could also be adapted, such as using directed fuzzing [1] when asking for examples reaching a particular location.

Fuzz with more context

Currently, functions are fuzzed in complete isolation. Rather than fuzzing functions individually or indirectly through the `main` function, we could only record examples that passed through a fixed level of other functions. These indirectly found examples are potentially more natural because the code in outer functions restricts which values reach inner functions. Alternatively, the fuzzer could start with function arguments retrieved from tests using example mining [7].

More natural examples

The usual objective of fuzzing – finding a small input that breaks the code – does not quite match our use case of finding helpful examples. Recent developments in LLMs make it practical to generate natural examples automatically [12]. Some approaches [23, 13] try to integrate LLMs into the input generation and mutation parts of fuzzing to guide the fuzzer to sensible inputs in a more targeted way. Using that technique to maximize how natural examples look might lead to helpful and correct examples grounded in actual execution. Parallel to this work, Mattis et al. created a framework to more rigorously evaluate examples in multiple dimensions [11] such as complexity, exceptionality, abstractness, or interactivity. We could evaluate our approach using this newly created framework.

Relate examples to coverage

We could clarify the relationship between examples and their coverage by moving examples to unique code locations they reach or greying out unreached code when clicking on an example.

Prioritize examples

Currently, examples are unordered. An intentional ordering might first show an example covering a happy path and examples covering crashes and only then display examples that reach niche code locations. Alternatively, examples reaching the cursor position could be considered more relevant, leading to a reordering of examples as you move the cursor through the code.

Interactive examples

Currently, examples are only displayed as text. Examples should be the starting point for visualizations, debugging sessions, unit tests, and other tools.

6.2 Generalizing to Other Languages

We chose Martinaise because it allowed us to investigate using fuzzing in the editor without focusing on more advanced challenges posed by real-world languages. These are ways in which our prototype could be adapted to different languages:

Dynamically Typed Languages

In dynamically typed languages, functions generally do not have type annotations, so the fuzzer can only rely on coverage data to guide it to valid (read: non-crashing) inputs. Because inputs with unexpected shapes are likely to cause functions to crash, crashes are less meaningful than in statically typed languages. However, passing examples can be a very valuable replacement for types.

Languages With Pure Functions

Mathematical functions cannot modify their inputs or have side effects. Some programming languages model their code similarly: They favor immutable data structures and functions without side effects (“pure functions”).

In these languages, fuzzing would not need to protect against mutations of inputs or side effects. Communicating the insights from fuzzing is more straightforward for these languages.

Languages With First-Class Functions

Some languages support functions as values. To fuzz higher-order functions (functions expecting functions), the fuzzer needs to generate a function. Mocking a function as a map from inputs to outputs enables the fuzzer to slightly mutate the function’s behavior, enabling evolution, and display the function to the developer.

7 Conclusion

Concrete examples can be valuable while editing abstract code and are needed in live programming, but coming up with examples takes effort and usually happens after writing code. However, there are ways to execute code without specifying examples: Fuzzing is a technique that synthesizes program inputs to find bugs in security-critical software.

In this paper, we used fuzzing to show examples in the editor. Our tool generates examples by starting with random inputs, which it then mutates based on coverage feedback to reach interesting code locations. While fuzzing focuses on finding crashes, it also produces valid inputs as a byproduct that can serve as examples for developers. Aesthetic qualities of inputs are of no concern to fuzzers, but are important when showing examples to the user.

Fuzzing the whole program just to get examples for the functions visible in the editor takes too long. By fuzzing individual functions instead of entire programs, we can display examples next to function signatures and do not have to find a path from the program entry point, making the process faster. This way, examples can be generated reasonably quickly, but we also show examples that a function would never be called with.

We evaluate our tool in small case studies, showing generated examples for numbers, strings, and composite objects. Similar to other tools such as type checking or linting, our tool gives reasonably quick feedback, but the results take a moment to stabilize as the fuzzer explores more code and refines examples. Unlike examples created by humans or large language models, fuzzed examples feel synthetic, but they are grounded in the dynamic behavior of code, cover the entire execution, and test edge cases.

References

- 1 Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, Dallas Texas USA, October 2017. ACM. doi:10.1145/3133956.3134020.
- 2 Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based Greybox Fuzzing as Markov Chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, Vienna Austria, October 2016. ACM. doi:10.1145/2976749.2978428.
- 3 Peng Chen and Hao Chen. Angora: Efficient Fuzzing by Principled Search. In *2018 IEEE Symposium on Security and Privacy (SP)*, May 2018. doi:10.1109/SP.2018.00046.
- 4 Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. GREYONE: Data Flow Sensitive Fuzzing. In *Proceedings of the 29th USENIX Conference on Security Symposium*, SEC'20, 2020. doi:10.5555/3489212.3489357.
- 5 Marcel Garus. *Martinaise*. Software, swbId: swb:1:dir:8a3acf542aa172ae7ccc3a1384b6fd5303d42f61 (visited on 2025-09-08). URL: <https://github.com/MarcelGarus/martinaise>, doi:10.4230/artifacts.24697.
- 6 Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with Code Fragments. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, USA, 2012. USENIX Association. doi:10.5555/2362793.2362831.
- 7 Eva Krebs, Patrick Rein, and Robert Hirschfeld. Example Mining: Assisting Example Creation to Enhance Code Comprehension. In *Companion Proceedings of the 6th International Conference on the Art, Science, and Engineering of Programming*, Porto Portugal, March 2022. ACM. doi:10.1145/3532512.3535226.
- 8 Sorin Lerner. Projection boxes: On-the-fly reconfigurable visualization for live programming. In Regina Bernhaupt, Florian 'Floyd' Mueller, David Verweij, Josh Andres, Joanna McGrenere, Andy Cockburn, Ignacio Avellino, Alix Goguey, Pernille Bjørn, Shengdong Zhao, Briane Paul Samson, and Rafal Kocielnik, editors, *CHI '20: CHI Conference on Human Factors in Computing Systems*, Honolulu, HI, USA, April 25-30, 2020, CHI '20, pages 1–7, New York, NY, USA, 2020. ACM. doi:10.1145/3313831.3376494.
- 9 Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *CoRR*, abs/2305.01210, December 2023. doi:10.48550/arXiv.2305.01210.
- 10 Dominik Christian Maier, Lukas Seidel, and Shinjo Park. Basesafe: baseband sanitized fuzzing through emulation. In René Mayrhofer and Michael Roland, editors, *WiSec '20: 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, Linz, Austria, July 8-10, 2020, pages 122–132. ACM, July 2020. doi:10.1145/3395351.3399360.

- 11 Toni Mattis, Lukas Böhme, Stefan Ramson, Tom Beckmann, Martin C. Rinard, and Robert Hirschfeld. Dimensions of examples: Toward a framework for qualifying examples in programming. In *Proceedings of the Programming Experience 2025 (PX/25) Workshop*, Programming '25. Schloss Dagstuhl – Leibniz-Zentrum für Informatik (OASICS), 2025. doi:10.4230/OASICS.Programming.2025.22.
- 12 Toni Mattis, Eva Krebs, Martin C. Rinard, and Robert Hirschfeld. Examples out of Thin Air: AI-Generated Dynamic Context to Assist Program Comprehension by Example. In *Companion Proceedings of the 8th International Conference on the Art, Science, and Engineering of Programming*, Programming '24, New York, NY, USA, July 2024. Association for Computing Machinery. doi:10.1145/3660829.3660845.
- 13 Ruijie Meng, Martin Mirchev, Marcel Böhme, and Abhik Roychoudhury. Large Language Model guided Protocol Fuzzing. In *Proceedings 2024 Network and Distributed System Security Symposium*, San Diego, CA, USA, 2024. Internet Society. doi:10.14722/ndss.2024.24556.
- 14 Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12), December 1990. doi:10.1145/96267.96279.
- 15 Arghavan Moradi Dakhel, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel C. Desmarais, and Zhen Ming (Jack) Jiang. GitHub Copilot AI pair programmer: Asset or Liability? *Journal of Systems and Software*, 203, September 2023. doi:10.1016/j.jss.2023.111734.
- 16 Joe Nelson. The design and use of quickcheck, January 2017. URL: <https://begriffs.com/posts/2017-01-14-design-use-quickcheck.html>.
- 17 Fabio Niephaus, Patrick Rein, Jakob Edding, Jonas Hering, Bastian König, Kolya Opahle, Nico Scordialo, and Robert Hirschfeld. Example-based live programming for everyone: building language-agnostic tools for live programming with LSP and GraalVM. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Virtual USA, November 2020. ACM. doi:10.1145/3426428.3426919.
- 18 David Rauch, Patrick Rein, Stefan Ramson, Jens Lincke, and Robert Hirschfeld. Babylonian-style Programming: Design and Implementation of an Integration of Live Examples into General-purpose Source Code. *The Art, Science, and Engineering of Programming*, 3(3), February 2019. doi:10.22152/programming-journal.org/2019/3/9.
- 19 Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types. In Michael D. Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 2597–2614. USENIX Association, 2021. URL: <https://www.usenix.org/conference/usenixsecurity21/presentation/schumilo>.
- 20 Tijs Van der Storm and Felienne Hermans. Live Literals, 2016. Presented at the Workshop on Live Programming (LIVE). URL: <https://homepages.cwi.nl/~storm/livelit/livelit.html>.
- 21 Bret Victor. Inventing on Principle, February 2012. URL: <https://www.youtube.com/watch?v=PUv66718DII>.
- 22 Yanhao Wang, Xiangkun Jia, Yuwei Liu, Kyle Zeng, Tiffany Bao, Dinghao Wu, and Purui Su. Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*, San Diego, CA, 2020. The Internet Society. doi:10.14722/ndss.2020.24422.
- 23 Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. Fuzz4All: Universal Fuzzing with Large Language Models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, ICSE '24*, New York, NY, USA, April 2024. Association for Computing Machinery. doi:10.1145/3597503.3639121.

- 24 Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. Krace: Data race fuzzing for kernel file systems. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 1643–1660. IEEE, 2020. doi:10.1109/SP40000.2020.00078.
- 25 Michal Zalewski. Binary fuzzing strategies: what works, what doesn't, August 2014. URL: <https://lcamtuf.blogspot.com/2014/08/binary-fuzzing-strategies-what-works.html>.
- 26 Michal Zalewski. Technical “whitepaper” for afl-fuzz, January 2015. URL: https://lcamtuf.coredump.cx/afl/technical_details.txt.
- 27 Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. FIRM-AFL: High-Throughput greybox fuzzing of IoT firmware via augmented process emulation. In *Proceedings of the 28th USENIX Conference on Security Symposium, SEC'19, USA, 2019*. USENIX Association. doi:10.5555/3361338.3361415.