From Static Code to Dynamic Values: Toward Live Programming Through Object-Oriented Fuzzing

Marcel Garus
marcel.garus@hpi.uni-potsdam.de
Hasso Plattner Institute,
University of Potsdam
Potsdam, Germany

Philipp Kolbe
philipp.kolbe@student.hpi.uni-potsdam.de
Hasso Plattner Institute,
University of Potsdam
Potsdam, Germany

Robert Hirschfeld
robert.hirschfeld@hpi.uni-potsdam.de
Hasso Plattner Institute,
University of Potsdam
Potsdam, Germany

Abstract

Examples can help illustrate the behavior of code, but they are not always easily available. We use fuzzing, a technique that typically aims to find bugs in security-critical software, to automatically derive example inputs for object-oriented code snippets. Starting with random object graphs as inputs, we mutate them so that they achieve high coverage in the code and conform to the shape that the code expects. Our prototype FuzzLens uses the Truffle compiler ecosystem to run code in a language-agnostic way, instrument code for coverage tracking, and intercept member accesses on the input objects. A VS Code extension shows a selection of examples that are deemed relevant as well as an abstracted summary of the function's behavior. In our anecdotal experience, we found the selected examples concise and helpful to aid code comprehension. We believe that this can help programmers gain a more solid understanding of code and lower the barrier to using dynamic tools such as probes or debuggers.

CCS Concepts: • **Software and its engineering → Software notations and tools**.

Keywords: fuzzing, example-based programming, Babylonian programming, dynamic languages, code coverage, randomized testing, virtual machine

ACM Reference Format:
Marcel Garus, Philipp Kolbe, and Robert Hirschfeld. 2026. From Static Code to Dynamic Values: Toward Live Programming Through Object-Oriented Fuzzing . In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (DEBT '26).* ACM, New York, NY, USA, 7 pages. https://doi.org/XXXXXXX.XXXXXXX

DEBT '26, Brussels, Belgium

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Code comprehension is an important activity [16]. To understand code, programmers can either read through the code statically or try to understand the dynamic behavior by running tests or using runtime tools such as debuggers.

However, the static and dynamic investigation of code don't complement each other well. A debugging session jumps around and disregards the big-picture structure of the code in favor of understanding the subset needed for a particular debugging example. A static read-through doesn't answer questions that programmers might have about the runtime behavior of specific parts of the code [7, 9]: What exactly is the meaning of this variable? When is this branch reached? What are typical results of this expression?

When writing or debugging code, there is a similar dilemma: Reasoning about the code abstractly can provide a more comprehensive understanding of its behavior, while running the code shows one instance of concrete behavior.

One approach of bringing static code and dynamic behavior closer together is by showing examples next to the source code. This practice of illustrating abstract algorithms with a narrative example exists since ancient Babylon [6]. Since programs are created to work on concrete data, examples allow programmers to build an intuition on how code behaves and an understanding of why some code exists.

However, this example-driven workflow requires a suite of tests or examples with sufficient coverage to illustrate all parts of the code. Especially when using code to explore a new problem domain, the intended behavior might not be obvious at the start of the programming session.

We implemented a fuzzer for dynamically-typed object oriented languages that can generate example inputs for Python and JavaScript functions. As seen in fig. 1, our tool FuzzLens[1] shows runtime-based feedback about a function's behavior next to the code. We believe that this can help programmers gain a more solid understanding of code and lower the barrier to using dynamic tools such as probes or debuggers.

[1]available at https://github.com/MarcelGarus/fuzzlens

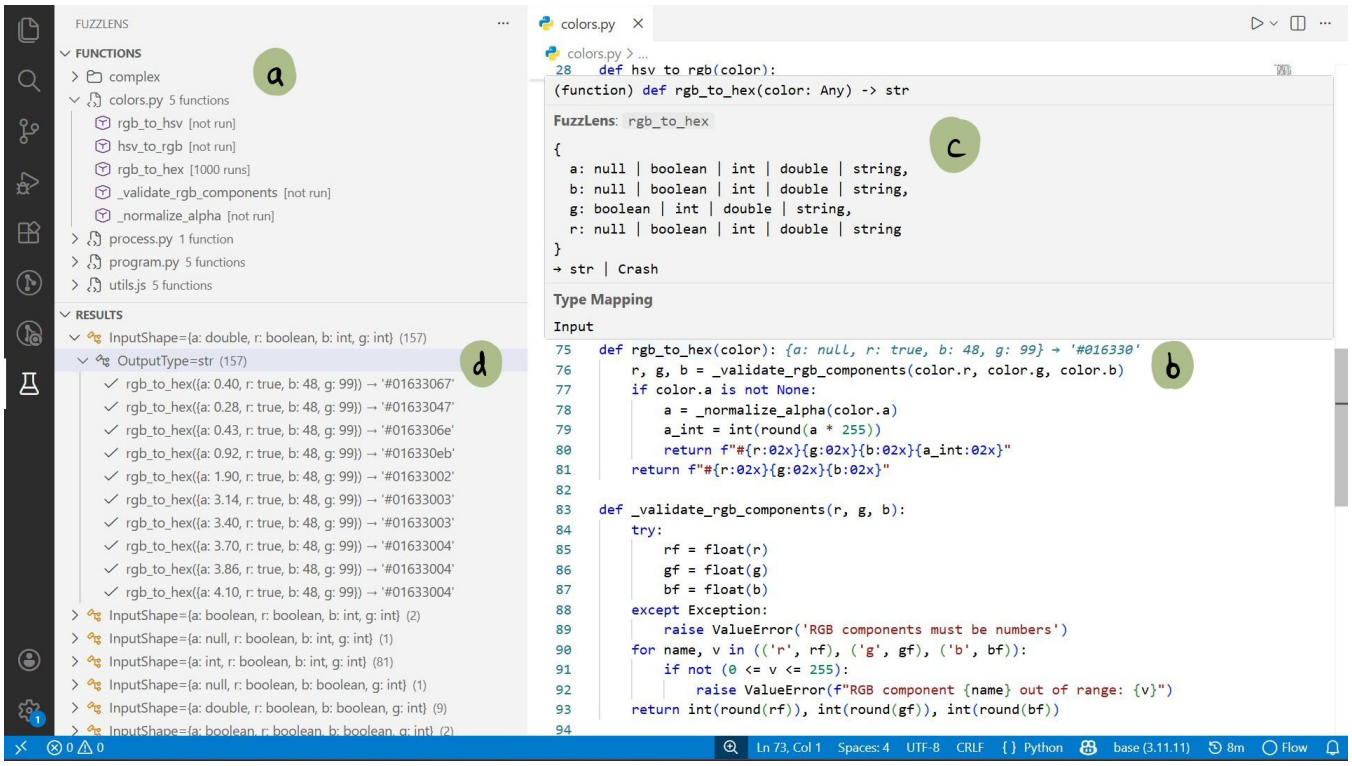

Figure 1. Our FuzzLens extension for VS Code can generate example inputs for functions. The left panel shows all functions in files of the current workspace (a). Clicking on one of those functions fuzzes it, causing examples to appear next to the function signature (b). Hovering over the name of the function displays a summary of the function's behavior (c). The left side shows the individual function invocations (d).

In this paper, we first describe related work of using examples to illustrate code (section 2). We describe our implementation (section 3), discuss it (section 4), and conclude (section 5).

2 Related Work

Examples map from an abstract program to a concrete behavior, so they can serve as the basis for debugging sessions, profiling runs, and other dynamic analyses that rely on actually running the code.

By showing examples directly in the source code, they can help illustrate how values flow through the program, a practice sometimes called Babylonian Programming. Figure 2 shows a Babylonian Programming editor [13], where programmers can specify inputs to the code and set probes to see the result of intermediary expressions.

To rate the usefulness of examples, *dimensions of examples* [10] proposes the following content-oriented criteria:

Complexity How much data does an example contain?
Exceptionality Is the example typical or an edge case?
Closeness to Domain Does the example use meaningless data or is it similar to real-word usages?

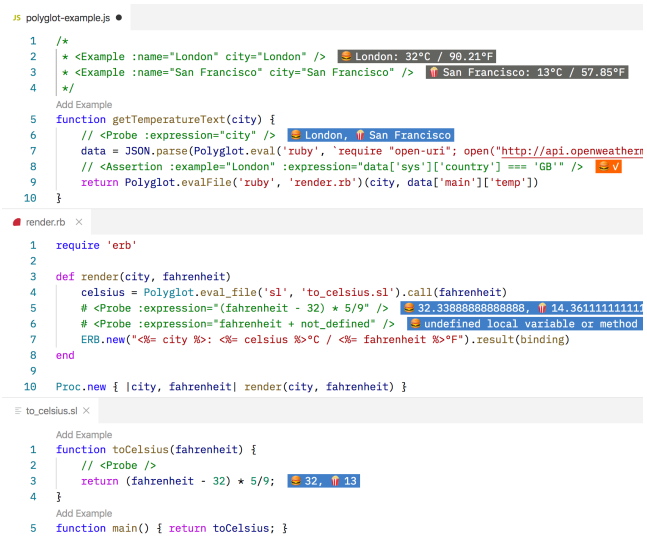

Figure 2. A Babylonian Programming [14] extension recognizes comments that contain example, probe, or assertion directives. It runs the annotated functions with the examples and displays results of probed expressions directly next to the source code in colorful boxes.

Scope Does the example show multiple aspects of the concept?

Abstractness Does the example contain placeholders?

To illustrate the behavior of code, we want a concise and comprehensive set of examples. Typically, not every individual method is extensively tested, but there are ways to find exemplary inputs for a method:

Human-provided examples in the form of tests can be amplified: Tracing their execution yields a wider variety of inputs for internal method calls [8]. This way, one example for a piece of code is amplified into many more examples for its dependencies.
Examples found through this example mining are concrete, close to the domain, and contain both common cases and edge cases. However, some examples may be too complex to be useful in isolation, or cover multiple different aspects of the behavior that could be better explained with multiple smaller examples. Also, example mining only works for existing, tested code. Newly written code may not be sufficiently tested to showcase the entire behavior.

LLMs can generate natural-looking examples for a given method body.
An exploration of this approach [11] in a dynamically-typed language showed that examples can be small, concrete, close to the domain, and generally useful when the LLM is prompted correctly. However, their (local) LLM sometimes struggled to generate examples that don't cause the code to crash, especially for code that lacks tests. Furthermore, LLMs are quite resource-intense, so examples are only generated on demand and one at a time.

Automatic testing techniques such as symbolic execution [2], property-based testing [4], or fuzzing [12, 17] can find inputs that reach corner cases in the program. Fuzzing in particular runs code with random inputs, observes metrics like coverage and crashes, and mutates the inputs to maximize those metrics. While this technique is primarily used to find bugs in security-critical software, it also produces many inputs that the code runs on successfully.
Fuzzers can often find a minimal set of concrete examples that together showcase the entire behavior, including edge cases. However, the generated inputs are often unnatural and don't respect domain expectations that are not explicitly checked by the code. For example, a fuzzer will happily use gibberish or empty strings for a parameter called `name`.

In a previous proof-of-concept implementation, we used fuzzing to generate example inputs for functions written in a small toy language. Based on static type signatures of functions, our tool would generate inputs and show them next to the source code [3].

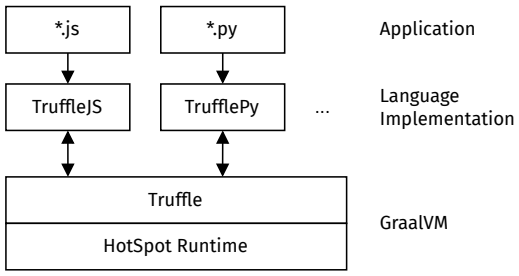

Figure 3. GraalVM makes it easy to implement programming languages. After writing an interpreter in Java using the Truffle compiler framework, GraalVM can automatically JIT-compile code.

A discussion at the PX/25 workshop hinted at potential use for dynamically-typed programming languages. Here, examples could document assumptions that the code makes about its inputs. Unlike generating examples based on tests or using LLMs, fuzzing works for newly-written, untested code, and cheaply produces a collection of examples with different behavior.

3 Implementation

We want to implement fuzzing for dynamically-typed, object-oriented languages. The Truffle/GraalVM ecosystem offers an instrumentable compiler infrastructure for object-oriented languages and has been used by previous work on example-based live programming [13].

Hence, we'll briefly describe the Truffle compiler infrastructure, how we use it to implement fuzzing, and how we extract insights from the fuzzing runs.

3.1 Truffle/GraalVM

GraalVM consists of a Java VM and Truffle, a Java-based compiler implementation framework [15]. Based on a program and an interpreter for the program's language written using Truffle, an optimizing JIT compiler constant-folds the program into the interpreter to generate machine code.

Truffle supports many languages, including JavaScript, Python, and Smalltalk. As seen in fig. 3, multiple interpreters can run side by side.

Because all interpreters live in the same object space, code written in different programming languages can call each other. By creating *proxy objects* that have a well-defined interface for discovering and accessing members, code can interact with objects created in a different programming language [5].

Using Truffle also enables cross-language tooling: *Instrumentation plugins* can inspect and dynamically rewrite programs, for example, to trace function calls or collect code coverage across languages.

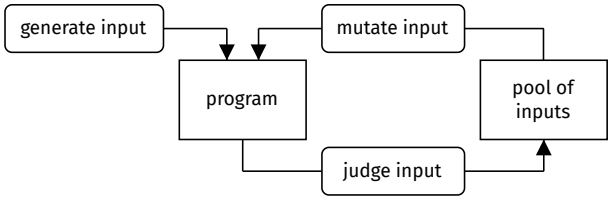

Figure 4. How fuzzing works on a high-level. Fuzzers feed random inputs into a tested program, judge the input quality based on the observed program behavior, and mutate promising inputs to explore similar behavior.

3.2 Fuzzing using Truffle

Typically, fuzzers maintain a pool of inputs along with aspects of the resulting program behavior, such as achieved coverage. Fuzzers start with random inputs and then gradually shift towards choosing promising inputs from the pool and mutating them slightly, as seen in fig. 4.

A single input in our fuzzer is a graph consisting of primitives and objects. Our fuzzer uses two Truffle mechanisms to get feedback about the input quality:

First, it creates an instrumentation plugin that wraps all expressions in nodes that track the coverage. When an expression of the program is reached for the first time, it records the new coverage and rewrites itself to remove coverage tracking.

Second, it uses Truffle's proxy objects intended for language interoperability to detect which input shape a function expects: The fuzzer turns an input into concrete GraalVM primitives and Truffle proxy objects to call the function. When a member on such a proxy object is accessed, Truffle transforms that into a language-independent object access—Truffle calls a `getMember` function on the proxy object with the name of the accessed member—which aborts the current run and allows the fuzzer to detect that attempted member access.

After a function execution, the fuzzer knows whether the function crashed, what part of the code was covered, and which members were accessed on input objects. Based on that, it then mutates the input (for example, changing numbers slightly or toggling booleans) and it expands the input (adding accessed members to objects). Over time, the fuzzer learns the structure of the expected inputs.

Take this Python function:

```
def distance(a, b):
    return sqrt((a.x - b.x) ** 2 + (a.y - b.y) ** 2)
```

When the fuzzer tries calling this function with primitive values such as `booleans`, the code will throw an `AttributeError` when trying to access `x` (fig. 5.a). At some point, the fuzzer will by chance decide to pass a proxy object to the function (fig. 5.b) and observe that the code tries to access `x`. This encourages the fuzzer to try out different values for the member

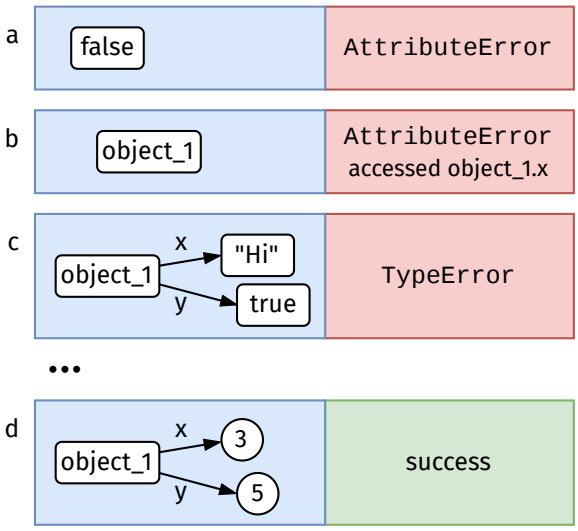

Figure 5. Our fuzzer builds up an object graph based on the observed behavior. Initially, graphs contain random primitives (a) and proxy objects (b). If a program tries to access an object's member and then crashes, the fuzzer considers adding that member (c). Over time, this can approximate the expected structure of objects (d). For illustration purposes, only one of the two inputs required for the prior code example is shown.

`x`. Apart from runs that result in `AttributeErrors` for missing members or `TypeErrors` for unsupported operand types for the minus operator (fig. 5.c), it also finds valid inputs (fig. 5.d).

3.3 Choosing examples

We want to show a set of examples that is small and simple enough not to overwhelm the developer, but diverse and big enough to be representative of the function's behavior. The examples should highlight both the default behavior as well as surprising edge cases.

In dynamically-typed languages, fuzzing finds many uninteresting crashes: A `logarithm` function may crash for the input `"Hello"` and the input `-1`, but work on the input `2`. We argue that `2` and `-1` are more relevant examples than `"Hello"` because the function behaves differently (crashing / non-crashing) for inputs of the same type. To capture this preference in our example selection, we abstract values into a "shape", which is a type-approximation based on the value's structure. Currently, shapes can represent primitive types (null, boolean, int, double, string) and objects. For example, the input `{x: 2, y: 3}` has the shape `{x: int, y: int}`.

We then define the following metrics that our tool uses to decide which inputs to show in the editor:

Output shape diversity Our tool prefers inputs where the same input shape results in a high variance in

output shapes or crashes, as described above for the `logarithm` function. The idea is that input shapes where it only fails sometimes are more interesting that input shapes that fail all the time because this variance indicates edge cases such as bugs or invariants on the arguments.

Input shape simplicity We prefer inputs with a simpler structure (objects with fewer fields) as those take up less visual and mental space. Minimizing inputs to illustrate behavior is common practice in fuzzers [17].

Input validity Unlike other fuzzers, our tool prefers showing inputs that don't cause a crash. Despite the over-representation of crashing inputs in our sampling (dynamically-typed code is generally easy to crash), we want to also show behavior for successful program executions.

Coverage rarity Our tool prefers inputs that cover rare behavior—if only very few of all fuzzing runs cover a piece of code, they are worth highlighting. Focusing on rarity has already been used successfully as a fuzzing strategy [1] to correct for the over-representation of inputs that are likely to be generated randomly.

Path simplicity Our tool prefers several inputs that each cover a specific behavior rather than one input that covers everything.

As these heuristics have different scales, our tool normalizes them to the range $[0, 1]$ and weighs them by hand-picked weights[2]. We then group examples by input shape and coverage path and select the top examples of every group. This allows us to get a variety of samples across different behavior groups that should together give a broad overview over the function's behavior.

4 Discussion

Our prototypical VS Code extension FuzzLens is shown in fig. 1. The left side contains an additional panel for our extension with the functions of all opened files (fig. 1.a). Clicking on a function starts a fuzzing run.

Afterwards, textual example inputs appear next to the function signature (fig. 1.b). Every couple seconds, this changes to a new example. Hovering over the function provides an abstracted summary of the input and output shapes (fig. 1.c). The panel on the left also shows individual function invocations, grouped by the input shape and the output type (fig. 1.d).

FuzzLens produces examples reasonably fast. Currently, it executes a given function 1000 times; for small functions this takes about 2 seconds and discovers every behavior. Depending on the complexity and runtime of the fuzzed function, the fuzzer may need more runs to explore the entire behavior space or a single run may take longer. Optimally, a fuzzing-based tool would continuously stream results, so that examples appear immediately and then improve over time.

Here's how our examples score among the Dimensions of Examples [10]:

Complexity Because small input size (while still achieving high coverage) is one of the scoring criteria, our shown examples are usually small.

Exceptionality Because we value diversity in the coverage paths when selecting examples, we show both examples that execute the happy path as well as edge cases.

Closeness to Domain Fuzzing can at best generate structurally correct input. Contrary to human- or LLMs-written examples, the actual data are meaningless in the context of the domain. For example, a `name` might be an empty string.

Scope Rather than finding examples that exhibit different behaviors, our tool tends to prefer multiple examples that each stress a single behavior. Depending on the use-case, different examples might be useful: Happy-path examples might help understand code, while edge cases can help when trying to make the code more robust.

Abstractness Our examples don't contain any placeholders. This should make it possible to integrate them with runtime tools such as debuggers.

While our prototype shows examples derived from fuzzing, we don't yet fully utilize all information that the fuzzer uncovers. In particular, coverage information is currently only used to guide the fuzzer, but could be used to show examples at relevant locations. For example, `if` statements could show examples satisfying the condition (fig. 6.a).

Also, our tool is currently quite isolated from other runtime tools. Reifying the examples into tests or integrating our extension more tightly with other tools could make them more useful. For example, crashing inputs might be good starting points for debugging sessions (fig. 6.b). Inputs that cause a long runtime might indicate a need for profiling (fig. 6.c). In our anecdotal teaching experience, computer science students automatically use static tools (syntax highlighting, squiggly lines for compiler errors, etc.) because they are enabled by default, but they miss good opportunities to use dynamic tools. Presenting them with readily-available examples could encourage them to use debuggers and profilers.

Apart from refining the presentation and integration of the examples in the editor, we also plan on improving the following technical aspects:

[2]Our weights: 10.0 output shape diversity; 10.0 input shape simplicity; 100.0 input validity; 2.0 coverage rarity; 1.0 path simplicity. We selected the weights based on what other inputs we wished to see. For example, when our tool showed many crashing inputs, but no valid example, we would increase the weight of input validity.

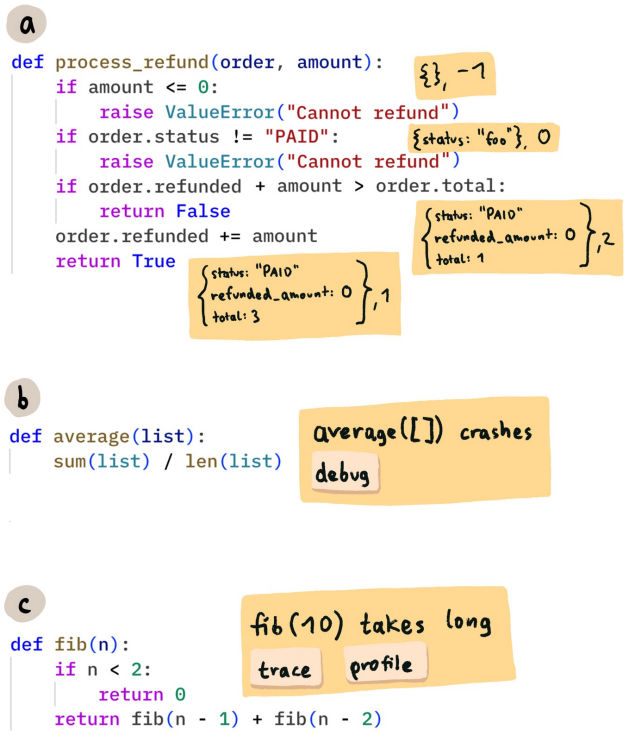

Figure 6. Sketches of how FuzzLens could be extended. (a) It could use coverage information more deliberately, showing examples at every branch and return statement. (b) It could integrate with other tools such as debuggers. (c) It could track additional metrics such as the time it takes to run the code.

Substitute concrete behavior for dynamic dispatch Many dynamically-typed languages use dynamic dispatch—when an object receives a message, the invoked method depends on the object's method dictionary. Currently, the fuzzer assumes nothing about how those methods relate, which does not match the assumptions of code: If code accepts a parameter called `list` and then calls push and pop on it, it generally expects the popped value to be the previously pushed one. The fuzzer could instead look for classes that correspond to the called methods and substitute concrete values of those classes.

Meta-Programming The fuzzer already struggles with code that uses basic meta-programming, e.g. JavaScript code that checks if `input instanceof Foo`. This can be solved by detecting object interactions not using proxy objects, but using the lower-level interoperability API, which allows customizing every observable aspect of an object, including the internal calls of `getPrototypeOf()` by Truffle's JavaScript VM.

Use metrics to guide fuzzer Currently, our fuzzer uses simple metrics like coverage to guide its mutations; the more advanced metrics are only used to select which inputs to display. They could instead also feed back into the mutation mechanism.

Finally, we need to conduct more experiments to validate the usefulness of FuzzLens in practice. Currently, we only have anecdotal experience on small, pure functions. There, the tool provides concise examples that cover all cases, sometimes highlighting surprising semantics for edge cases. While many production code bases contain such small, pure functions, they also bring more challenges in the form of larger functions, functions with side effects, higher-order functions, and framework-heavy code. The next step to evaluate the usefulness of FuzzLens in a more rigorous way on a wider audience is to conduct a user study.

5 Conclusion

Examples can illustrate how code behaves, but they are difficult to harvest. Fuzzing is an option to automatically generate examples that cover the behavior space. Our extension prototype FuzzLens shows examples inputs for functions for dynamically-typed object-oriented languages. We see potential in integrating the generated examples more deeply with other runtime tools.

Acknowledgments

To Christoph Thiede for proof-reading. To Tom Beckmann for feedback on our tool. To Tim Felgentreff and Fabio Niephaus for feedback on our tool and hints on how to use GraalVM effectively.

References

[1] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based Greybox Fuzzing as Markov Chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, Vienna Austria. doi:10.1145/2976749.2978428

[2] Cristian Cadar and Koushik Sen. 2013. Symbolic execution for software testing: three decades later. *Commun. ACM* 56, 2 (Feb. 2013), 82–90. doi:10.1145/2408776.2408795

[3] Marcel Garus, Jens Lincke, and Robert Hirschfeld. 2025. Fuzzing as Editor Feedback. In *Companion Proceedings of the 9th International Conference on the Art, Science, and Engineering of Programming (Programming 2025) (Open Access Series in Informatics (OASIcs), Vol. 134)*, Jonathan Edwards, Roly Perera, and Tomas Petricek (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 8:1–8:15. doi:10.4230/OASIcs.Programming.2025.8

[4] Harrison Goldstein, Joseph W Cutler, Daniel Dickstein, Benjamin C Pierce, and Andrew Head. 2024. Property-based testing in practice. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[5] Matthias Grimmer, Chris Seaton, Roland Schatz, Thomas Würthinger, and Hanspeter Mössenböck. 2015. High-performance cross-language interoperability in a multi-language runtime. In *Proceedings of the 11th Symposium on Dynamic Languages* (Pittsburgh, PA, USA) *(DLS 2015)*. Association for Computing Machinery, New York, NY, USA, 78–90. doi:10.1145/2816707.2816714

[6] Donald E. Knuth. 1972. Ancient Babylonian algorithms. *Commun. ACM* 15, 7 (July 1972), 671–677. doi:10.1145/361454.361514

[7] Amy J. Ko and Brad A. Myers. 2004. Designing the whyline: a debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Vienna, Austria) *(CHI '04)*. Association for Computing Machinery, New York, NY, USA, 151–158. doi:10.1145/985692.985712

[8] Eva Krebs, Patrick Rein, and Robert Hirschfeld. 2022. 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*. ACM, Porto Portugal. doi:10.1145/3532512.3535226

[9] Thomas D. LaToza and Brad A. Myers. 2010. Hard-to-answer questions about code. In *Evaluation and Usability of Programming Languages and Tools* (Reno, Nevada) *(PLATEAU '10)*. Association for Computing Machinery, New York, NY, USA, Article 8, 6 pages. doi:10.1145/1937117.1937125

[10] Toni Mattis, Lukas Böhme, Stefan Ramson, Tom Beckmann, Martin C Rinard, and Robert Hirschfeld. 2025. Dimensions of examples: Toward a framework for qualifying examples in programming. In *Companion Proceedings of the 9th International Conference on the Art, Science, and Engineering of Programming (Programming 2025)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 13–1. doi:10.4230/OASIcs.Programming.2025.13

[11] Toni Mattis, Eva Krebs, Martin C. Rinard, and Robert Hirschfeld. 2024. 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)*. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3660829.3660845

[12] Barton P. Miller, Lars Fredriksen, and Bryan So. 1990. An empirical study of the reliability of UNIX utilities. *Commun. ACM* 33, 12 (dec 1990). doi:10.1145/96267.96279

[13] Fabio Niephaus, Patrick Rein, Jakob Edding, Jonas Hering, Bastian König, Kolya Opahle, Nico Scordialo, and Robert Hirschfeld. 2020. 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) *(Onward! 2020)*. Association for Computing Machinery, New York, NY, USA, 1–17. doi:10.1145/3426428.3426919

[14] David Rauch, Patrick Rein, Stefan Ramson, Jens Lincke, and Robert Hirschfeld. 2019. 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 (feb 2019). doi:10.22152/programming-journal.org/2019/3/9

[15] Christian Wimmer and Thomas Würthinger. 2012. Truffle: a self-optimizing runtime system. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity* (Tucson, Arizona, USA) *(SPLASH '12)*. Association for Computing Machinery, New York, NY, USA, 13–14. doi:10.1145/2384716.2384723

[16] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E. Hassan, and Shanping Li. 2018. Measuring Program Comprehension: A Large-Scale Field Study with Professionals. *IEEE Transactions on Software Engineering* 44, 10 (2018), 951–976. doi:10.1109/TSE.2017.2734091

[17] Michal Zalewski. 2015. Technical "whitepaper" for afl-fuzz. https://lcamtuf.coredump.cx/afl/technical_details.txt

Received 11 March 2026; revised 11 March 2026; accepted 11 March 2026