Leveraging Formal Specifications to Generate Fuzzing Suites

Nicolas Osborne, Clément Pascutto

Leveraging Formal Specifications to Generate Fuzzing Suites

Nicolas Osborne[1] and Clément Pascutto[1,2]

[1]Tarides, 75005 Paris, France
nicolas.osborne@tarides.com, clement@tarides.com

[2]Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria, Laboratoire Méthodes Formelles, 91190, Gif-sur-Yvette, France

Abstract

When testing a library, developers typically first have to capture the semantics they want to check. They then write the code implementing these tests and find relevant test cases that expose possible misbehaviours.

In this work, we present a tool that automatically takes care of these last two steps by automatically generating fuzz testing suites from OCaml interfaces annotated with formal behavioural specifications. We also show some ongoing experiments on the capabilities and limitations of fuzzing applied to real-world libraries.

1 Introduction

Library testing is the most common and accessible approach for ensuring software engineering safety. It typically requires developers to agree on the semantics of their programs before writing the code implementing checks for this semantics, and finally finding relevant test cases that expose possible misbehaviours or trigger edge cases. We present a tool to automate these last two steps by generating fuzz testing suites out of formal specifications, so developers only have to write down the properties that the implementation should satisfy.

This work relies on Gospel [2], a contract-based behavioural specification language, Monolith [3], a model-based test framework for fuzzing OCaml libraries, and is part of `ortac`, a runtime assertion checking tool for OCaml. The `ortac` project is open-source and available at `https://github.com/ocaml-gospel/ortac`.

At its core, `ortac` identifies an executable subset of the Gospel specifications and translates formulae into OCaml Boolean expressions. It then wraps the user-written implementation with these runtime checks. If the implementation complies with the specification, the original implementation and the wrapper have the same behaviour. Otherwise, the wrapped code raises an exception carrying information about the unmet specifications.

In this presentation, we will show an example of our workflow and explain how the code generation works under the hood. We will finish with some experiments on the capabilities and limitations of fuzzing and applications to real-world libraries.

2 Workflow Overview

The first—and only—task of the developer is to annotate their module signature with Gospel comments that hold function contracts and type invariants. Figure 1 shows the contents of a file `power.mli` declaring a function `power` annotated with a simple Gospel contract.

```
val power : int -> int -> int
(*@ r = power x n
    requires n >= 0
    ensures r = pow x n *)
```

Figure 1: `Power` module interface.

Once the user has specified their module interface, they can call `ortac` with `monolith` enabled on the interface file. The tool prints the generated code on `stdout` so they — or the build system — can redirect it in a file:

```
$ ortac --frontend=monolith power.mli > main.ml
```

At this point, `main.ml` contains a complete and correct Monolith program that will exercise the contract of `power`. The user may run it either in random mode or in fuzzing mode with `afl-fuzz` [1]:

```
# random mode
$ ./main
# fuzzing mode
$ afl-fuzz -i inputs/ -o outputs/ -- ./main @@
```

In both cases, Monolith provides inputs to the annotated functions and reports errors in the directory `outputs/crashes` in the form of replayable scenarios. To get more information about a misbehaviour, the user can replay the scenario by passing the corresponding file name to the generated program as an argument. This way, the user has access to the failure scenario and all the errors reported by `ortac` on these specific inputs, highlighting the broken specifications.

```
$ ./main outputs/crashes/<filename>
File "power.mli", lines 1-4, characters 0-26:
 Runtime error in function 'power':
 - the post-condition
     'r = pow x n'
   was violated.
```

3 Under the Hood: a Frontend for `ortac`

The architecture of `ortac` is designed such that its core can be specialised by a *frontend* in order to extend its default behaviour. Our fuzzing frontend specialises it in three different ways to integrate it with Monolith. Figure 2 presents the structure of the generated code.

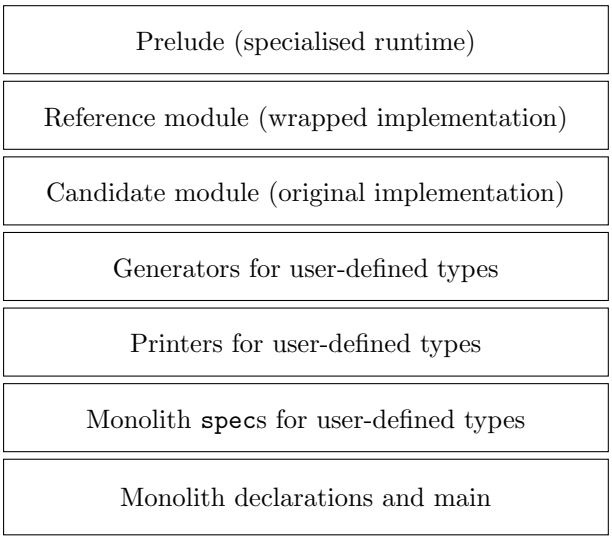

Figure 2: Organisation of the generated code.

A specialised runtime. We provide the final code with a specialised runtime, adapting the default `ortac-runtime` to Monolith error reporting. For example, if a precondition is unmet, we do not report an error but rather ignore this scenario. Our runtime also provides generators and printers that Monolith does not provide natively.

Reference and candidate implementations. Monolith is a *model* based test framework; it requires two modules with the same signature, a `Reference` and a `Candidate`. We use the original implementation as the `Candidate`, and the wrapper generated by `ortac` as the `Reference`, as it contains the information about the expected behaviour.

Monolith declarations. The last part of the program registers the functions in the interface and their types so that Monolith can exercise them. Our frontend generates these declarations automatically from the typing information by constructing printers and random generators for user-defined types.

4 Conclusion and Future Work

The `ortac` tool and its `monolith` frontend are under active development to extend the support for user-defined OCaml types and Gospel specifications. We also plan to continue the experiments on real-world libraries in order to explore further the capabilities and limitations of `ortac` combined with fuzzing.

References

[1] `afl-fuzz` — `American fuzzy lop`. `https://lcamtuf.coredump.cx/afl/`.

[2] Arthur Charguéraud, Jean-Christophe Filliâtre, Cláudio Lourenço, and Mário Pereira. GOSPEL - Providing OCaml with a Formal Specification Language. In *FM 2019 - 23rd International Symposium on Formal Methods*, October 2019.

[3] François Pottier. Strong automated testing of OCaml libraries. In *Journées Francophones des Langages Applicatifs (JFLA)*, February 2021.