Testing with Crowbar

Stephen Dolan
University of Cambridge

Mindy Preston
Docker, Inc.

Introduction

Crowbar is a new testing library for OCaml, combining two software testing techniques that have proven remarkably effective at finding bugs:

Property-based testing With QuickCheck, Claessen and Hughes [1] introduced a new style of software testing: using a library for describing generators of structured data, they show how to write tests as properties that must hold universally, rather than just in specific instances. The advantage of QuickCheck over previous means of generating test data is that by composing generators with simple combinators, it becomes easy to write specific generators to test arbitrary interfaces.

Coverage-based fuzzing Testing software by repeatedly feeding it unexpected (often randomly generated) input is called *fuzzing*. Modern fuzz-testing tools have sophisticated means of generating input data: Zalewski's `afl-fuzz` [2] instruments the program under test to build a table of execution paths, and mutates the input data using a mix of random and deterministic techniques with the goal of getting the highest coverage of execution paths.

While sophisticated techniques are used to generate input data, the condition being checked during fuzzing is generally simple: often, no more than "did the program crash?". This is in contrast to QuickCheck, where the input data is generated in a simple way but complex conditions are checked.

Crowbar is a testing library combining property-based testing with coverage-based fuzzing, by using `afl-fuzz` to generate test data for QuickCheck-style tests. To support it, new versions of the OCaml compiler are able to generate `afl-fuzz`-compatible instrumentation, logging the execution paths taken while evaluating a particular testcase. Future OCaml versions from 4.05 onwards have this support, and an unofficial backport `4.04.0+afl` is also available.

In use, Crowbar is similar to QuickCheck, with the user specifying generators and properties. Figure 1 defines a generator `json` for structured JSON values, and tests that they are correctly serialised and deserialised by the `yojson` library. The `guard` function is used to restrict the test cases generated, avoiding illegal infinity and `NaN` values.

Note the lack of options controlling frequency and distribution of the test cases, which are a common feature of QuickCheck-style libraries. Instead, the distribution of test data is controlled automatically by `afl-fuzz`, so as to maximise the number of code paths tested.

```
let float_is_finite f =
  match classify_float f with
  | FP_infinite | FP_nan -> false
  | _ -> true

let rec json = Choose [
  Map ([bool], fun b -> `Bool b);
  Map ([int], fun i -> `Int i);
  Map ([List json], fun xs -> `List xs);
  Const `Null;
  Map ([float], fun f ->
    guard (float_is_finite f); `Float f);
  Map ([str], fun s -> `String s);
  Map ([List field], fun xs -> `Assoc xs)]
and str =
  Choose [Const "a"; Const "b"; Const "c"]
and field =
  Map ([str; json], fun s e -> s, e)

let pp_json ppf json =
  pp_string ppf
    (Yojson.Safe.pretty_to_string json)

let () =
  add_test ~name:"yojson" [json] (fun j ->
    let s = Yojson.Safe.to_string j in
    check_eq ~pp:pp_json
      j (Yojson.Safe.from_string s))
```

Figure 1: Testing the `yojson` library

Finding bugs

The above test reproduces an old bug in version 1.2.1 of `yojson`, although it finds no new bugs in the current version. Beyond `yojson`, 12 other libraries were tested, with previously-unreported bugs found in 9 of them.

Table 1 shows the results, comparing how long it took to find a bug using Crowbar versus the standard QuickCheck approach (for which we used the same tests, but with input data generated at at random rather than via `afl-fuzz`). Most of the tests were only a few dozen lines long (including pretty-printing of test cases), with the exception of `charrua-core`, which was extensively tested (only one of the tests is shown). All packages tested may be found in the public OPAM repository.

The bugs in `bencode`, `cbor` and `yojson` were all to do with integer overflow, and triggered only on certain bit-patterns. Choosing integers at random did not manage to trigger the bug, but by using a special generator which picks a number of the form 2^k+e ($0 \leq k < 64$, $-2 \leq e < 2$) half of the time, the bugs were quickly reproduced (in a median time <1s).

		Instrumented			Uninstrumented		
package	version	time	tests	range	time	tests	range
bencode	1.0.2	<1 s	6 k	0.3x–2.7x	-	-	-[1]
bes	0.9.4.2	2 s	24 k	0.6x–1.3x	<1 s	3 k	0.1x–5.0x
bson	0.89.3	<1 s	12 k	0.8x–1.1x	<1 s	<1 k	0.0x–5.0x
calendar	2.03.2	<1 s	18 k	0.5x–1.7x	<1 s	7 k	0.2x–7.7x
cbor	0.1	<1 s	10 k	0.3x–1.4x	-	-	-[1]
charrua-core	0.7	139 s	860 k	0.1x–2.6x	4579 s	31587 k	0.1x–2.3x
fpath	0.7.2	2 s	30 k	0.2x–2.7x	<1 s	<1 k	0.0x–4.4x
map	base	-	-	-	-	-	-
opamver	2.0.0 beta3	-	-	-	-	-	-
pprint	20140424	-	-	-	-	-	-
uunf	2.0.1	495 s	5660 k	0.2x–227.6x	-	-	-
xmldiff	0.5.0	2 s	22 k	0.6x–1.5x	2 s	29 k	0.0x–4.6x
yojson	1.2.1[2]	<1 s	12 k	0.2x–2.4x	-	-	-[1]

Instrumented testing used `afl-fuzz` to generate inputs, while uninstrumented testing generated inputs randomly. Execution counts are the median of 11 runs, with the ratio against the shortest and longest runs shown in the "range" column (where median = 1x). "-" indicates no failures after at least 1 billion executions.
[1] bug could be found using custom integer distribution (see text) [2] old version, bug not present in current version

Table 1: Comparing instrumented and uninstrumented testing.

Since integer overflow is always tricky, some QuickCheck implementations provide integer distributions like the one described. However, it is generally hard to construct a good distribution and verify that it is testing the program thoroughly.

The bug found in `charrua-core` (a DHCP library) was that it is possible to construct an unserialisable packet by specifying a conflicting combination of fixed-length and variable-length options. This bug was reproduced using uninstrumented QuickCheck-style testing, but took a median time of an hour and a quarter, compared to two minutes and 19 seconds for instrumented search. In fact, the slowest instrumented run was still faster than the fastest random search.

The `uunf` library implements Unicode normalisation, and contained a subtle bug which caused diacritics to be moved to the wrong letter in very rare circumstances. Random search did not find the bug, with `uunf` handling randomly-generated test cases correctly more than 20 billion times, consuming over a week of CPU time. Instrumented fuzzing found the bug after a median time of less than ten minutes. (The variance was high, but can be reduced by running several jobs in parallel).

The usefulness of instrumentation can be seen by examining the test cases generated by `afl-fuzz`: despite starting with no information about Unicode, after running for a minute the generated testcases were heavily biased towards Unicode blocks having complicated normalisation rules. This information can be used to improve randomised testing: randomised testing using only characters from the 10 blocks that were more than 100x over-represented in the generated corpus found the bug in about 40 minutes, although this was still slower than letting the fuzzer finish the job.

Discussion and future work

Effective use of QuickCheck and its descendants requires careful attention to the distribution of test cases: it is easy to generate too many easy cases, and leave the program insufficiently tested. Some manual effort is required to ensure that the test case generator hits the edge cases. However, the code itself contains enough information to determine which cases have complex behaviour and require more testing, and by exposing this information through instrumentation, modern fuzz-testing tools can be used to generate an effective distribution of test data.

Beyond Crowbar, there are many other applications of OCaml's new instrumentation, not least the use of `afl-fuzz` for standard fuzzing jobs (which has found bugs in OCaml's lexer). Another that we're keen to explore is the testing of concurrent programs by allowing `afl-fuzz` to drive scheduler decisions.

Crowbar is available from:
`https://github.com/stedolan/crowbar`

Acknowledgements We thank KC Sivaramakrishnan and Anil Madhavapeddy for comments on this abstract, and Jeremy Yallop for suggestions about the design of Crowbar.

References

[1] K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *ICFP '00*, pages 268–279. ACM, 2000.

[2] M. Zalewski. american fuzzy lop. `http://lcamtuf.coredump.cx/afl/`, 2014.