

Kaputt – version 1.1

<http://kaputt.x9c.fr>

Copyright © 2008-2011 Xavier Clerc – kaputt@x9c.fr
Released under the GPL v3

February 20, 2011

Introduction

Kaputt is a unit testing tool for the Objective Caml language¹. Its name stems from the following acronym: *Kaputt is A Popperian Unit Testing Tool*. The adjective *popperian* is derived from the name of Karl Popper, a famous philosopher of science who is known for forging the concept of *falsifiability*. The tribute to Popper is due to the fact that Kaputt, like most test-based methodologies, will never tell you that your function is correct; it can only point out errors.

Kaputt features two main kinds of tests:

- assertion-based tests, inspired by the *xUnit* tools²;
- specification-based tests, inspired by the *QuickCheck* tool³.

When writing assertion-based tests, the developer explicitly encodes input values and checks that output values satisfy given assertions. When writing specification-based tests, the developer encodes the specification of the tested function and then requests the library to either generate random values, or enumerate values to be tested against the specification.

Kaputt also provides shell-based tests that barely execute commands such as **grep**, **diff**, *etc.* They can be regarded as a special kind of assertion-based tests, and can be useful to run the whole application and compare its output to reference runs whose output has been stored into files.

Kaputt, in its 1.1 version, is designed to work with version 3.12.0 of Objective Caml. Kaputt is released under the GPL version 3. This licensing scheme should not cause any problem, as *test versions* of applications are intended to be used during development but should not be released publicly. Bugs should be reported at <http://bugs.x9c.fr>.

Building Kaputt

Bolt can be built from sources using **make** (in its **GNU Make 3.81** flavor), and Objective Caml version 3.12.0. No other dependency is needed. Following the classical Unix convention, the build

¹The official Caml website can be reached at <http://caml.inria.fr> and contains the full development suite (compilers, tools, virtual machine, *etc.*) as well as links to third-party contributions.

²Unit testing tools for Java (JUnit – <http://junit.org>), OCaml (OUnit – <http://www.xs4all.nl/~mmzeeman/ocaml/>), *etc.*

³<http://www.cs.chalmers.se/~rjmh/QuickCheck/>

and installation process consists in these three steps:

1. `sh configure`
2. `make all`
3. `make install`

During the first step, one can specify elements if they are not correctly inferred by the `./configure` script; the following switches are available:

- `-ocaml-prefix` to specify the prefix path to the Objective Caml installation (usually `/usr/local`);
- `-ocamlfind` to specify the path to the `ocamlfind` executable (notice that the presence of `ocamlfind`⁴ is optional, and that the tool is used only at installation if present);
- `-no-native-dynlink` to disable dynamic linking.

During the third and last step, according to local settings, it may be necessary to acquire privileged accesses, running for example `sudo make install`.

The Java⁵ version will be built only if the `ocamljava`⁶ compiler is present and located by the makefile.

Using Kaputt

Running tests from compiled code

To use Kaputt, it is sufficient to compile and link with the library. This is usually done by adding of the following to the compiler invocation:

- `-I +kaputt kaputt.cma` (for `ocamlc` compiler);
- `-I +kaputt kaputt.cmxa` (for `ocamlopt` compiler);
- `-I +kaputt kaputt.cmja` (for `ocamljava` compiler).

Since version 1.0, to access `bigarray`- and `num`-specific elements, it is necessary to link with respectively `kaputtBigarray.cm[oxj]` and `kaputtNums.cm[oxj]`.

Typically, the developer wants to compile the code for tests only for internal (test) versions, and not for public (release) versions. Hence the need to be able to build two versions. The `IFDEF` directive of `camlp4` can be used to fulfill this need. Code sample 1 shows a trivial program that is designed to be compiled either to *debug* or to *release* mode.

To compile the *debug* version, one of the following commands (according to the compiler used) should be issued:

- `ocamlc -pp 'camlp4oof -DDEBUG' source.ml;`
- `ocamlopt -pp 'camlp4oof -DDEBUG' source.ml;`

⁴Findlib, a library manager for Objective Caml, is available at <http://projects.camlcity.org/projects/findlib.html>

⁵The official website for the Java Technology can be reached at <http://java.sun.com>.

⁶OCaml compiler generating Java bytecode, by the same author – <http://ocamljava.x9c.fr>

Code sample 1 Trivial program featuring two versions (`source.ml`).

```
let () =
  IFDEF DEBUG THEN
    print_endline "debug mode on"
  ELSE
    print_endline "debug mode off"
  ENDIF
```

- `ocamljava -pp 'camlp4oof -DDEBUG' source.ml.`

At the opposite, to compile the *release* version, one of following commands should be executed:

- `ocamlc -pp camlp4oof source.ml;`
- `ocamlopt -pp camlp4oof source.ml;`
- `ocamljava -pp camlp4oof source.ml.`

This means that the developer can choose the version to compile by only specifying a different preprocessor (precisely by enabling/disabling a preprocessor argument) to be used by the invoked OCaml compiler.

Running tests from the toplevel

Code sample 2 shows how to use Kaputt from a toplevel session. First, the Kaputt directory is added to the search path. Then, the library is loaded and the module containing shorthand definitions is opened. Finally, the `check` method is used in order to check that the successor of an odd integer is even.

Code sample 2 Toplevel session running a generator-based test.

```
Objective Caml version 3.11.1

# #directory "+kaputt";;
# #load "kaputt.cma";;
# open Kaputt.Abbreviations;;
# check Gen.int succ [Spec.is_odd_int ==> Spec.is_even_int];;
Test 'untitled no 1' ... 100/100 cases passed
- : unit = ()
#
```

Writing assertion-based tests

When writing assertion-based tests, one is mainly interested in the `Assertion` and `Test` modules. The `Assertion` module provides various functions performing tests over values. Then, the `Test` module allows to run the tests and get some report about their outcome. An assertion-based test built by the `Test.make_assert_test` function is made of four elements:

- a title;
- a *set up* function, whose signature is `unit -> 'a`;
- a function performing the actual tests, whose signature is `'a -> 'b`;
- a *tear down* function, whose signature is `'b -> unit`.

The idea of the *set up* and *tear down* functions is that they bracket the execution of the test function. If there is no data to pass to the test function (*i.e.* its signature is `unit -> unit`), the obvious choices for *set up* and *tear down* are respectively `Test.return ()` and `ignore`; another possibility is to use the `make_simple_test` function. Code sample 3 shows a short program declaring and running two tests, the first one uses no data while the second one does. The second test also exhibits the fact that the title is optional.

Code sample 3 Assertion-based tests.

```
open Kaputt.Abbreviations

let t1 =
  Test.make_simple_test
    ~title:"first test"
    (fun () -> Assert.equal_int 3 (f 2))

let t2 =
  Test.make_assert_test
    (fun () -> open_in "data")
    (fun ch -> Assert.equal_string "waited1" (f1 ch); ch)
    close_in_noerr

let () = Test.run_tests [t1; t2]
```

Writing specification-based tests

When writing specification-based tests, one is mainly interested in the `Generator`, `Specification`, and `Test` modules. The `Generator` module defines the concept of generator that is a function randomly producing values of a given type, and provides implementations for basic types and combinators. The `Specification` module defines the concept of specification that is predicates over values and their images through the tested function, as well as predicates over basic types and combinators. A specification-based test built by `Test.make_random_test` is made of nine elements (the six first ones being optional):

- a title;
- an integer, indicating how many cases should be generated;
- an integer, indicating how tries should be made to generate an input value matching the specification⁷;

⁷Useful to avoid non-terminating issues if a non-satisfiable precondition is passed in the specification.

- a classifier, used to categorize the generated cases;
- a reducer, used to try to produce smaller counterexamples;
- a randomness source;
- a generator;
- a function to be tested;
- a specification.

The generator, of type `'a Generator.t`, is used to randomly produce test cases. Tests cases are produced until the requested number has been reached. One should notice that a test case is counted if and only if the generated value satisfies one of the preconditions of the specification.

The classifier is used to characterize the generated test cases to give the developer an overview of the coverage of the test (in the sense that the classifier gives hints about the portions of code actually executed). For complete coverage information, one is advised to use the Bisect tool⁸ by the same author.

The specification is a list of $\langle \text{precondition}, \text{postcondition} \rangle$ couples. This list should be regarded as a case-based definition. When checking if the function matches its specification, Kaputt will determine the first precondition from the list that holds, and ensure that the corresponding postcondition holds: if not, a counterexample has been found.

Assuming that the tested function has a signature of `'a -> 'b`, a precondition has type `'a predicate` (that is `'a -> bool`) and a postcondition has type `('a * 'b) predicate` (that is `('a * 'b) -> bool`). The preconditions are evaluated over the generated values, while the postconditions are evaluated over $\langle \text{generated values}, \text{image by tested function} \rangle$ couples.

An easy way to build $\langle \text{precondition}, \text{postcondition} \rangle$ couples is to use the `=>` infix operator. Additionally, the `==>` infix operator can be used when the postcondition is interested only in the image through the function (ignoring the generated value), thus enabling lighter notation.

Code sample 4 shows how to build a test for function `f` whose domain is the `string` type. The classifier stores generated values into two categories, according to the length of the string. The `prei` functions are of type `string -> bool`, while the `posti` functions are of type `(string * t) -> bool` where `t` is the codomain (also sometimes referred to as the “range”) of the tested function `f`.

Output modes

The previous sections have exposed how to run tests using the `Test.run_tests` function. When only passed a list of tests, the outcome of these tests is written to the standard output in a (hopefully) user-friendly text setting. It is however possible to change both the destination and the layout by supplying an optional `output` parameter of type `Test.output_mode`, that is a sum type with the following constructors:

- `Text_output of out_channel`
classical layout, destination being the given channel – cf. code sample 5

⁸Code coverage tool for the OCaml language – <http://bisect.x9c.fr>

Code sample 4 Specification-based tests.

```
open Kaputt.Abbreviations

let t =
  Test.make_random_test
    ~title:"random test"
    ~nb_runs:128
    ~classifier:(fun s -> if (String.length s) < 4 then "short" else "long")
    (Gen.string (Gen.make_int 0 16) Gen.char)
  f
  [ pre_1 => post_1 ;
    ...
    pre_n => post_n ]

let () = Test.run_test t
```

- `Html_output of out_channel`
HTML table-based layout, destination being the given channel
- `Xml_output of out_channel`
XML layout using the dtd shown by code sample 8, destination being the given channel – *cf.* code sample 6
- `Xml_junit_output of out_channel`
JUnit-compatible XML layout (enabling for instance Hudson⁹ integration), destination being the given channel
- `Csv_output of out_channel * string`
CSV layout using the given string as the separator, destination being the given channel – *cf.* code sample 7

The passed channel is closed if it is neither `stdout`, nor `stderr`.

Code sample 5 Example of text output.

```
Test 'succ test' ... 100/100 cases passed
Test 'untitled no 1' ... 10/10 cases passed
Test 'sum of odds' ... 200/200 cases passed
Test 'strings' ... 0/2 case passed
  counterexamples: "eYbHu", "UEggsF"
  categories:
    short -> 1 occurrence
    long -> 1 occurrence
Test 'lists' ... 0/2 case passed
  counterexamples: [ 6; 5; 1; 3; 6; ], [ 3; 2; 6; ]
```

⁹Continuous integration server <http://hudson-ci.org/>

Code sample 6 Example of XML output.

```
<kaputt-report>
  <random-test name="succ test" valid="100" total="100" uncaught="0">
  </random-test>
  <random-test name="untitled no 1" valid="10" total="10" uncaught="0">
  </random-test>
  <random-test name="sum of odds" valid="200" total="200" uncaught="0">
  </random-test>
  <random-test name="strings" valid="0" total="2" uncaught="0">
    <counterexamples>
      <counterexample value="&quot;0AsdUXKf&quot;"/>
      <counterexample value="&quot;dhVMK&quot;"/>
    </counterexamples>
    <categories>
      <category name="long" total="1"/>
      <category name="short" total="1"/>
    </categories>
  </random-test>
  <random-test name="lists" valid="0" total="2" uncaught="0">
    <counterexamples>
      <counterexample value="[ 5; 1; 6; ]"/>
      <counterexample value="[ 6; 3; ]"/>
    </counterexamples>
  </random-test>
</kaputt-report>
```

Code sample 7 Example of CSV output.

```
random-test (stats)|succ test|100|100|0
random-test (stats)|untitled no 1|10|10|0
random-test (stats)|sum of odds|200|200|0
random-test (stats)|strings|0|2|0
random-test (counterexamples)|strings|"SHwJpJ"|"tbMlVNwqh"
random-test (stats)|lists|0|2|0
random-test (counterexamples)|lists|[[ 3; 6; 6; ]][[ 3; 4; 5; ]]
```

Code sample 8 DTD used for XML output.

```
<!ELEMENT kaputt-report
  (passed-test|failed-test|uncaught-exception|random-test|enum-test|shell-test)*>

<!ELEMENT passed-test EMPTY>
<!ATTLIST passed-test name CDATA #REQUIRED>

<!ELEMENT failed-test EMPTY>
<!ATTLIST failed-test name CDATA #REQUIRED>
<!ATTLIST failed-test expected CDATA>
<!ATTLIST failed-test not-expected CDATA>
<!ATTLIST failed-test actual CDATA #REQUIRED>
<!ATTLIST failed-test message CDATA>

<!ELEMENT uncaught-exception EMPTY>
<!ATTLIST uncaught-exception name CDATA #REQUIRED>
<!ATTLIST uncaught-exception exception CDATA #REQUIRED>

<!ELEMENT random-test (counterexamples?,categories?)>
<!ATTLIST random-test name CDATA #REQUIRED>
<!ATTLIST random-test valid CDATA #REQUIRED>
<!ATTLIST random-test total CDATA #REQUIRED>
<!ATTLIST random-test uncaught CDATA #REQUIRED>

<!ELEMENT enum-test (counterexamples?)>
<!ATTLIST enum-test name CDATA #REQUIRED>
<!ATTLIST enum-test valid CDATA #REQUIRED>
<!ATTLIST enum-test total CDATA #REQUIRED>
<!ATTLIST enum-test uncaught CDATA #REQUIRED>

<!ELEMENT counterexamples (counterexample*)>
<!ELEMENT counterexample EMPTY>
<!ATTLIST counterexample value CDATA #REQUIRED>

<!ELEMENT categories (category*)>
<!ELEMENT category EMPTY>
<!ATTLIST category name CDATA #REQUIRED>
<!ATTLIST category total CDATA #REQUIRED>

<!ELEMENT shell-test EMPTY>
<!ATTLIST shell-test name CDATA #REQUIRED>
<!ATTLIST shell-test exit-code CDATA #REQUIRED>
```
