Kaputt 1.2

Reference Manual

http://kaputt.x9c.fr

August 29, 2012

Contents

Chapter 1

Overview

1.1 Purpose

Kaputt is a unit testing tool for the OCaml language[1]. 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* tools[2];
- specification-based tests, inspired by the *QuickCheck* tool[3].

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.

1.2 License

Kaputt is distributed under the terms of 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.

[1] The official OCaml 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.

[2] Unit 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/`

1.3 Contributions

In order to improve the project, I am primarily looking for testers and bug reporters. Pointing errors in documentation and indicating where it should be enhanced is also very helpful.
Bugs and feature requests can be made at `http://bugs.x9c.fr`.
Other requests can be sent to kaputt@x9c.fr.

Chapter 2

Building Kaputt

2.1 Step 0: dependencies

Before starting to build Kaputt, one first has to check that dependencies are already installed. The following elements are needed in order to build Kaputt:

- OCaml, version 4.00.0;
- `make`, in its GNU Make 3.81 flavor;
- a classical Unix shell, such as `bash`;
- **optionally:** Findlib[1], version 1.3.3.

2.2 Step 1: configuration

The configuration of Argot is done by executing `./configure`. 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 OCaml installation (usually `/usr/local`);
- `-ocamlfind` to specify the path to the `ocamlfind` executable;
- `-no-native-dynlink` to disable the build of the native version, even if native dynamic linking is available.

The Java[2] version will be built only if the `ocamljava`[3] compiler is present and located by the makefile. The syntax extension will be compiled only to bytecode.

2.3 Step 2: compilation

The actual build of Kaputt is launched by executing `make all`. When build is finished, it is possible to run some simple tests by running `make tests`. Documentation can be generated by running `make doc`.

[1] Findlib, a library manager for OCaml, is available at http://projects.camlcity.org/projects/findlib.html.

[2] The official website for the Java Technology can be reached at http://java.sun.com.

[3] OCaml compiler generating Java bytecode, by the same author – http://www.ocamljava.org

2.4 Step 3: installation

Kaputt is installed by executing `make install`. According to local settings, it may be necessary to acquire privileged accesses, running for example `sudo make install`. The actual installation directory depends on the use of `ocamlfind`: if present the files are placed inside the Findlib hierarchy, otherwise they are placed in the directory `‘ocamlc -where‘/kaputt` (*i. e.* `$PREFIX/lib/ocaml/kaputt`).

Chapter 3

Using Kaputt

3.1 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.

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
```

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`;
- `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.

3.2 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.

```
        OCaml version 4.00.0

# #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 = ()
#
```

3.3 Running tests from `.mlt` files

Since version 1.2, it is possible to use a preprocessor (named `kaputt_pp.byte`) in order to store tests in `.mlt` files. The underlying idea is to use the file `mod.mlt` to store the tests for the module `Mod` whose implementation is in file `mod.ml`. The preprocessor just appends the contents of `mod.mlt` to `mod.ml` when the compiler processes the file `mod.ml`. If no `.mlt` file is found alongside the `.ml` file, the preprocessor does nothing.

The `kaputt_pp.byte` should always be invoked with a first parameter either equal to `on` or `off`, indicating whether concatenation of `.ml` and `.mlt` files should occur. The other parameters should indicate which preprocessor to use for the `.ml` file, leading to a command-line with the following form:

```
ocamlc -c -pp './kaputt_pp.byte on camlp4o' source.ml
```

or

```
ocamlc -c -pp './kaputt_pp.byte off camlp4o' source.ml
```

This way of organizing tests is useful because is allows to simultaneously:

- have tests stored separately, in their own files;
- have test code that can access to every element of the tested module (including non-exported elements);
- easily switch between building with or without test code.

When using the preprocessor while compiling with the `ocamlbuild` tool, one has to be cautious and not to forget to copy `.mlt` files into the build directory of `ocamlbuild`. Assuming that all source files are in the `src` directory and its subdirectories, this can be done through the `ocamlbuild` plugin shown by code sample 3. Then, it is sufficient to tag files with the `kaputt` tag defined by the plugin with a line such as `<src/**/*.ml>: kaputt` in the `_tags` file.

Code sample 3 `myocamlbuild.ml` file with support for Kaputt preprocessor.

```
open Ocamlbuild_plugin
open Ocamlbuild_pack

let rec copy_mlt_files path =
  let elements = Pathname.readdir path in
  Array.iter
    (fun p ->
      if Pathname.is_directory (path / p) then
        copy_mlt_files (path / p)
      else if Pathname.check_extension p "mlt" then
        let src = path / p in
        let dst = !Options.build_dir / path / p in
        Shell.mkdir_p (!Options.build_dir / path);
        Pathname.copy src dst
      else
        ())
    elements

let () =
  dispatch begin function
    | After_rules ->
        copy_mlt_files "src";
        flag ["kaputt"; "pp"]
          (S [A"kaputt_pp.byte"; A"on"; A"camlp4o"])
    | _ -> ()
  end
```

Chapter 4

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 tested 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 4 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.

Mock functions may be useful when writing assertion-based tests. Mock functions are functions that can be created from usual functions, from ⟨ input, output ⟩ couples, or from ⟨ input, output ⟩ sequences. They also record all the calls made to the function, allowing to check if the function has been used as expected. Code sample 5 shows how to write an assertion-based test for the `List.map` function, ensuring that the higher-order function is called on each element of the passed list from left to right.

Code sample 4 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]
```

Code sample 5 Mock function and assertion-based tests.

```
open Kaputt.Abbreviations

let () =
  Test.add_simple_test
    (fun () ->
      let eq_int_list = Assert.make_equal_list (=) string_of_int in
      let f = Mock.from_function succ in
      let i = [0; 1; 2; 0] in
      let o = List.map (Mock.func f) i in
      let o' = [1; 2; 3; 1] in
      eq_int_list o' o;
      eq_int_list i (Mock.calls f);
      Assert.equal_int 4 (Mock.total f))
```

Chapter 5

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 specification[1];
- 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 be 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 tool[2] by the same author.

[1] Useful to avoid non-terminating issues if a non-satisfiable precondition is passed in the specification.
[2] Code coverage tool for the OCaml language – http://bisect.x9c.fr

The specification is a list of ⟨precondition, postcondition⟩ 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 ⟨generated values, image by tested function⟩ couples.
An easy way to build ⟨precondition, postcondition⟩ 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 6 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 $\texttt{pre}\_i$ functions are of type `string -> bool`, while the $\texttt{post}\_i$ functions are of type `(string * t) -> bool` where `t` is the codomain (also sometimes referred to as the "range") of the tested function `f`.

Code sample 6 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
```

It is also possible to write specification for partial function, and to check then though `xyz_partial` functions. Partial functions have a codomain type that is `'b outcome` rather than simply `'b`. The `Specification.outcome` type is a sum type with two constructors: `Result of 'b`, and `Exception of exn`. The `Specification` module provides two combinators `is_exception` and `is_result` that allow to respectively test an exceptional result and a normal result. Code sample 7 tests that the tested `f` function (of type `int -> int`) raises an exception when passed an odd value, and return an even value when passed an even value.

Code sample 7 Specification-based tests of a partial function.

```
open Kaputt.Abbreviations

let () =
  check_partial
    Gen.int
    f
    [Spec.is_even_int ==> Spec.is_result Spec.is_even_int ;
     Spec.is_odd_int ==> Spec.is_exception Spec.always ]
```

Chapter 6

Output modes

The previous chapter 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 8
- `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 11, destination being the given channel – *cf.* code sample 9
- `Xml_junit_output of out_channel`
  JUnit-compatible XML layout (enabling for instance Jenkins[1] 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 10

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

[1] Continuous integration server http://jenkins-ci.org/

Code sample 8 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; ]
```

Code sample 9 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="&amp;quot;OAsdUXKf&amp;quot;"/>
      <counterexample value="&amp;quot;dhVMK&amp;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 10 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 11 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>
```