testkraut Documentation

Release

The testkraut guys

August 21, 2015

Contents

> *Note: testkraut is still in its infancy – some of what is written here could still be an anticipation of the near future.*

This is a framework for software testing. That being said, testkraut tries to minimize the overlap with the scopes of unit testing, regression testing, and continuous integration testing. Instead, it aims to complement these kinds of testing, and is able to re-use them, or can be integrated with them.

In a nutshell testkraut helps to facilitate statistical analysis of test results. In particular, it focuses on two main scenarios:

1. Comparing results of a single (test) implementation across different or changing computational environments (think: different operating systems, different hardware, or the same machine before an after a software upgrade).
2. Comparing results of different (test) implementations generating similar output from identical input (think: performance of various signal detection algorithms).

While such things can be done using other available tools as well, testkraut aims to provide a lightweight (hence portable), yet comprehensive description of a test run. Such a description allows for decoupling test result generation and analysis – opening up the opportunity to "crowd-source" software testing efforts, and aggregate results beyond the scope of a single project, lab, company, or site.

At this point your probably want to *Get started (quickly)*.

Bug tracker | Build status | Documentation | Downloads | PyPi

CHAPTER 1

Wanna help?

If you think it would be worthwhile to contribute to this project your input would be highly appreciated. Please report issues, send feature-requests, and pull-request without hesitation!

CHAPTER 2

License

All code is licensed under the terms of the MIT license, or some equally liberal alternative license. Please see the COPYING file in the source distribution for more detailed information.

CHAPTER 3

Documentation

3.1 Get started (quickly)

This should not take much time. testkraut contains no compiled code. It should run with Python 2.6 (or later) – although Python 3x hasn't been tested (yet). If you are running Python 2.6 you should install the argparse package, otherwise you won't have much fun. Here is a list of things the make life more interesting:

NumPy not strictly required, but strongly recommended. There should be no need to have any particular version.

SciPy will improve the test result reporting – any reasonably recent version should do

libmagic helps to provide more meaningful information on file types

python-colorama for more beautiful console output – but monochrome beings don't need it

3.1.1 Download ...

testkraut is available from PyPi, hence it can be installed with `easy_install` or `pip` – the usual way. `pip` seems to be a little saner than the other one, so we'll use this:

```
% pip install testkraut
```

This should download and install the latest version. Depending on where you are installing you might want to call `sudo` for additional force.

`pip` will tell you where it installed the main `testkraut` script. Depending on your setup you may want to add this location to your `PATH` environment variable.

3.1.2 ... and run

Now we're ready to run our first test. The `demo` test requires FSL to be installed and configured to run (properly set `FSLDIR` variable and so on...). The main testkraut script supports a number of commands that are used to prepare and run tests. A comprehensive listing is available form the help output:

```
% testkraut --help
```

To run the `demo` test, we need to obtain the required test data first. This is done by telling testkraut to cache all required files locally:

```
% testkraut cachefiles demo
```

It will download an anatomical image from a webserver. However, since the image is the MNI152 template head that comes with FSL, you can also use an existing local file to populate the cache – please explore the options for this command.

Now we are ready to run:

```
% testkraut execute demo
```

If FSL is functional, this command will run a few seconds and create a subdirectory `testbeds/demo` with the test in/output and a comprehensive description of the test run in JSON format:

```
% ls testbeds/demo
brain_mask.nii.gz  brain.nii.gz  head.nii.gz  spec.json
```

That is it – for now...

3.2 Prototypes of a testkraut user

3.2.1 The concerned scientist

This scientist came up with a sophisticated data analysis pipeline, consisting of many pieces of software from different vendors. It appears to work correctly (for now). But this scientist is afraid to upgrade any software on the machine, because it might break the pipeline. Rigorous tests would have helped, but "there was no time". testkraut can help to (semi-automatically) assess the longitudinal stability of analysis results.

3.2.2 The thoughtful software developer

For any individual software developer or project it is almost impossible to confirm proper functioning of their software on all possible computing environments. testkraut can help generate informative performance reports that can be send back to a developer and offer a more comprehensive assessment of cross-platform performance.

3.2.3 The careful "downstream"

A packager for a software distribution needs to apply a patch to some software to improve its integration into the distribution environment. Of course, such a patch should not have a negative impact on the behavior of the software. testkraut can help to make a comparative assessment to alert the packager if something starts to behave in unexpected ways.

3.3 The SPEC

A test specification (or SPEC) is both primary input and output data for a test case. As input data a SPEC defines test components, file dependencies, expected test output, and whatever else that is necessary to describe a test case. As test output, a SPEC is an annotated version of the input SPEC, with a detailed descriptions of various properties of observed test components and results. A SPEC is text file in JSON format.

Path specifications for files can make use of environment variables which get expanded appropriately. The special variable `TESTKRAUT_TESTBED_PATH` can be used to reference the directory in which a test is executed.

The following sections provide a summary of all SPEC components.

3.3.1 authors

A *JSON object*, where keys email addresses of authors of a SPEC and corresponding values are the authors' names.

3.3.2 dependencies

A *JSON object* where keys are common names for dependencies of a test case. Values are *JSON object*s with fields described in the following subsections.

location

Where is the respective namespace?

For executables this may contain absolute paths and/or environment variables which will be expanded to their actual values during processing. Such variables should be listed in the `environment` section.

type

Could be an `executable` or a `python_mod`.

optional

A *JSON boolean* indicating whether an executable is optional (`true`), or required (`false`; default). Optional executables are useful for writing tests that need to accommodate changes in the implementation of the to-be-tested software.

version_cmd

A *JSON string* specifying a command that will be executed to determine a version of an executable that is added as value to the `version` field of the corresponding entry for this executable in the `entities` section. If an output to `stderr` is found, it will be used as version. If no `stderr` output is found, the output to `stdout` will be used.

Alternatively, this may be a *JSON array* with exactly two values, where the first value is, again, the command, and the second value is a regular expression used to extract matching content from the output of this command. Output channels are evaluated in the same order as above (first `stderr`, and if no match is found `stdout`).

version_file

A *JSON string* specifying a file name. The content of this file will be added as value to the `version` field of the corresponding entry for this executable in the `entities` section.

Alternatively, this may be a *JSON array* with exactly two values, where the first value is, again, a file name, and the second value is a regular expression used to extract matching content from this file as a version.

Example

```
"executables": {
   "$FSLDIR/bin/bet": {
     "version_cmd": [
           "$FSLDIR/bin/bet2",
           "BET \\(Brain Extraction Tool\\) v(\\S+) -"
     ]
   },
   "$FSLDIR/bin/bet2": {
     "version_file": "$FSLDIR/etc/fslversion"
   }
```

3.3.3 description

A *JSON string* with a verbal description of the test case. The description should contain information on the nature of the test, any input data files, and where to obtain them (if necessary).

This section is identical in input SPEC and corresponding output SPEC.

3.3.4 entities

A *JSON object*, where keys are unique identifiers (*JSON string*), and values are *JSON object*s. Identifiers are unique but identicial for identical entities, even across systems (e.g. the file sha1sum). All items in this section describe entities of relevance in the context of a test run – required executables, their shared library dependencies, script interpreters, operating system packages providing them, and so on. There are various categories of values in this section that can be distinguished by their `type` field value, and which are described in the following subsections.

This section only exists in output SPECs.

type: binary

This entity represents a compiled executable. The following fields are supported

path (*JSON string*) Executable path as specified in the input SPEC.

provider (*JSON string*) Identifier/key of an operating system package entry in the `entities` section.

realpath (*JSON string*) Absolute path to the binary, with all variables expanded and all symlinks resolved.

sha1sum (*JSON string*) SHA1 hash of the binary file. This is identical to the item key.

shlibdeps (*JSON array*) Identifiers/keys of shared library dependency entries in the `entities` section.

version (*JSON string*) Version output generated from the `version_cmd` or `version_file` settings in the input SPEC for the corresponding executable.

type: deb or rpm

This entity represents a DEB or RPM package. The following fields are supported

arch (*JSON string*) Identifier for the hardware architecture this package has been compiled for.

name (*JSON string*) Name of the package.

sha1sum (*JSON string*) SHA1 hash for the package.

vendor (*JSON string*) Name of the package vendor.

version (*JSON string*) Package version string.

type: library

This entity represent a shared library. The types and meaning of the supported fields are identical to `binary`-type entities, except that there is no `version` field.

type: script

This entity represents an interpreted script. The types and meaning of the supported fields are identical to `binary`-type entities, except that there is no `shlibdeps` field, but instead:

interpreter (*JSON string*) Identifier/key for the script interpreter entry in the `entities` section.

3.3.5 environment

A *JSON object*, where keys represent names of variables in the system environment. If the corresponding value is a string the respective variable will be set to this value prior test execution. If the value is `null` any existing variable of such name will be unset. If the value is `true` the presence of this variable is required and its value is recorded in the protocol. If the value is `false`, the variable is not required and its (optional) value is recorded.

3.3.6 comparisons

yet to be determined

3.3.7 id

A *JSON string* with an ID that uniquely identifies the test case. In a test library the test case needs to be stored in a directory whose name is equal to this ID, while the SPEC is stored in a file named `spec.json` inside this directory. While not strictly required, it is preferred that this ID is "human-readable" and carries an reasonable amount of semantic information. For example: `fsl-mcflirt` is a test the is concerned with the MCFlirt component of the FSL suite.

This section is identical in input SPEC and corresponding output SPEC.

3.3.8 inputs

A *JSON object*, where keys represent IDs of required inputs for a test case. Corresponding values are, again, *JSON object*s with a mandatory `type` field. The value of `type` is a *JSON string* identifying the type of input. Currently only type `file` is supported. For a `file`-type input the following additional fields should be present:

sha1sum (*JSON string*) SHA1 hash that uniquely identifies the input file.

tags (*JSON array*) Optional list of *JSON string*s with tags categorizing the input (see *tags*).

url (*JSON string*) URL where the respective file can be downloaded.

value (*JSON string*) name of the input file.

Example

```
"inputs": {
  "head.nii.gz": {
    "sha1sum": "41d817176ceb99ac051d8bd066b500f3fb89be89",
    "type": "file",
    "value": "head.nii.gz"
  }
}
```

3.3.9 outputs

This section is very similar to the `inputs` section, and may contain similar information in matching fields with identical semantics. In contrast to `inputs` this section can be substantially extended in the output SPEC. For example, output files may not have a SHA1 hash specified in the input SPEC, but a SHA1 hash for the actually observed output file will be stored in the output's `sha1sum` field. Most importantly, for any output file whose `tags` match one or more of the configured *fingerprint generators* a `fingerprints` field will be added to the *JSON object* for the corresponding output file.

fingerprints

The value of this field is a *JSON object* where keys are names of fingerprint generators, and values should be *JSON object*s with a custom structure that is specific to the particular type of fingerprint. All fingerprints should contain a `version` field (*JSON number*; integer) that associates any given fingerprint with the implementation of the generator that created it.

3.3.10 processes

A *JSON object* describing causal relationships among test components. Keys are arbitrary process IDs. Values are *JSON object*s with fields described in the following subsections.

This section is currently not modified or extended during a test run.

argv (*JSON array*) `argv`-style command specification for a process. For example:

```
    ["$FSLDIR/bin/bet", "head.nii.gz", "brain", "-m"]
```

executable (*JSON string*) ID/key of the associated executable from the `executables` section.

generates (*JSON array*) IDs/keys of output files (from the `outputs` section) created by this process.

started_by (*JSON string*) ID/key of the process (from the same section) that started this process.

uses (*JSON array*) IDs/keys of input files (from the `inputs` section) required by this process.

Example

```
"0": {
  "argv": [
    "$FSLDIR/bin/bet2",
    "head",
    "brain",
    "-m"
  ],
```

```
  "executable": "$FSLDIR/bin/bet2",
  "generates": [
    "brain.nii.gz",
    "brain_mask.nii.gz"
  ],
  "started_by": 1,
  "uses": [
    "head.nii.gz"
  ]
},
```

3.3.11 system

A *JSON object* listing various properties of the computational environment a test was ran in. This section is added by the test runner and only exists in output SPECs.

3.3.12 tests

A *JSON array* of *JSON object*s describing the actual test cases. All (sub-)test cases are executed in order of appearance in the array, in the same test bed, using the same environment. Multiple sub-tests can be used to split tests into sub parts to improve error reporting, while minimizing test SPEC overhead. However, output fingerprinting is only done once *after* all subtests have completed successfully.

For each *JSON object* describing a sub-test, the mandatory `type` field identifies the kind of test case and the possible content of this section changes accordingly. Supported scenarios are described in the following subsections.

For any test type, a test can be marked as an expected failure by adding a field `shouldfail` and setting its value to `true`.

An optional field `id` can be used to assign a meaningful identifier to a subtest that is used in the test protocol. If no `id` is given, as subtest's index in the tests array is used as identifier.

type: shell

The test case is a shell command. The command is specified in a text field `code`, such as:

```
"code": "$FSLDIR/bin/bet head.nii.gz brain -m"
```

In the output SPEC of a test run this section is amended with the following fields:

exitcode (*JSON number*; integer) Exit code for the executed command.

type: python

Explain me

type: nipype

Explain me

3.3.13 version

A *JSON number* (integer) value indicating the version of a SPEC. This version must be incremented whenever a change to a SPEC is done.

This section is identical in input SPEC and corresponding output SPEC.

3.4 Output fingerprinting

Modules with fingerprint generators:

fingerprints
fingerprints.base

3.4.1 Writing a custom fingerprinting function

Writing a custom fingerprint implementation for a particular kind of output is pretty straightforward. Start by creating a function with the following interface:

```
def fp_my_fingerprint(fname, fpinfo, tags):
  pass
```

The variable `fname` will contain the filename/path of the output for which a fingerprint shall be created. `fpinfo` is an empty dictionary to which the content of the fingerprint needs to be added. A test runner will add this dictionary to the `fingerprints` section of the respective output file in the SPEC. The name of the fingerprinting function itself will be used as key for this fingerprint element in that section. Any `fp_`-prefix, as in the example above, will be stripped from the name. Finally, `tags` is a sequence of *Output tags* that categorize a file and can be used to adjust the content of a fingerprint accordingly.

Any fingerprinting function must add a `__version__` tag to the fingerprint. The version must be incremented whenever the fingerprint implementation changes, to make longitudinal comparisons of test results more accurate.

There is no need to return any value – all content needs to be added to the `fpinfo` dictionary.

A complete implementation of a fingerprinting function that stores the size of an input file could look like this:

```
>>> import os
>>> def fp_file_size(fname, fpinfo, tags):
...   fpinfo['__version__'] = 0
...   fpinfo['size'] = os.path.getsize(fname)
>>> #
>>> # test it
>>> #
>>> from testkraut.fingerprints import proc_fingerprint
>>> fingerprints = {}
>>> proc_fingerprint(fp_file_size, fingerprints, 'COPYING')
>>> 'file_size' in fingerprints
True
>>> 'size' in fingerprints['file_size']
True
```

There is no need to catch exceptions inside fingerprinting functions. The test runner will catch any exception and everything that has been stored in the fingerprint content dictionary up to when the exception occurred will be preserved. The exception itself will be logged in a `__exception__` field.

To enable the new fingerprinting function, add it to any appropriate tag in the `fingerprints` section of the configuration file:

```
[fingerprints]
want size = myownpackage.somemodule.fp_file_size
```

With this configuration this fingerprint will be generated for any output that is tagged `want size`. It is required that the function is "importable" from the specified location.

3.5 Output tags

This glossary lists all known tags that can be used to label test outputs. According to the assigned tags appropriate fingerprinting or evaluation methods are automatically applied to the output data.

3D image, 4D image a sub-category of *volumetric image* with a particular number of axes

columns columns of a matrix or an array should be described individually

nifti1 format a file in any variant of the NIfTI1 format

numeric values a file containing an array/matrix of numeric values

rows rows of a matrix or an array should be described individually

store as much of the file content should be keep verbatim in a fingerprint

text file a file with text-only, i.e. non-binary content

table a file with data table layout (if a text format, column names are in first line; uniform but arbitrary delimiter)

tscores values from a Student's t-distribution

volumetric image a multi-dimensional (three or more) image

whitespace-separated fields data in a structured text format where individual fields are separated by any white-space character(s)

zscores standardized values indicating how many standard deviations an original value is above or below the mean

3.6 Terminology

JSON array An ordered sequence of values, comma-separated and enclosed in square brackets; the values do not need to be of the same type (for more information see the "JSON" Wikipedia entry section on data types)

JSON boolean Boolean value: `true` or `false` (for more information see the "JSON" Wikipedia entry section on data types)

JSON number Double precision floating-point format in JavaScript (for more information see the "JSON" Wikipedia entry section on data types)

JSON object an unordered collection of key:value pairs with the ':' character separating the key and the value, comma-separated and enclosed in curly braces; the keys must be strings and should be distinct from each other (for more information see the "JSON" Wikipedia entry section on data types)

JSON string Double-quoted Unicode, with backslash escaping (for more information see the "JSON" Wikipedia entry section on data types)

3.7 Frequently Asked Questions

Why this name? The original aim for this project was to achieve "crowd-sourcing" of software testing efforts. "kraut" is obviously almost a semi-homonym of "crowd", while at the same time indicating that this software spent its infancy at the Institute of Psychology Magdeburg, **Germany**.

3.8 Design

3.8.1 Goal

This aims to be a tool for testing real-world (integrated) software environments with heterogeneous components from different vendors. It does **not** try to be

- a unit test framework (you better pick one for you programming language of choice)
- a continuous integration testing framework (take a look at Jenkins or buildbot)
- a test framework for individual pieces of software (although that could work)

Instead, this tool is targeting the evaluation of fully deployed software on "production" systems. It aims at verifying proper functioning (or unchanged behavior) of software systems comprised of components that were not specifically designed or verified to work with each other.

Objectives

- Gather comprehensive information about the software environment
- Integrate test case written in arbitrary languages or toolkits with minimal overhead
- Make it possible to easily deploy the system on users' machines to verify their environments

3.8.2 Dump of discussion with Satra

- NiPyPE will do the provenance, incl. the gathering of system/env information
- anything missing in this domain needs to be added to nipype
- a test is a json file
- test code is stored directly in the json file
- tests will typically be nipype workflows
- individual tests will not depend on other tests (although a test runner could resolve data dependencies with outputs from other tests)
- a tests definition specifies: test inputs, test dependencies (e.g. software), and an (optional) evaluative statement

3.8.3 Dump of discussion with Alex

- A test fails or passes
- Evaluation assesses the quality of the test results (but doesn't necessarily let a test fail)
- Dashboard-level evaluation will provide highly aggregated analysis (e.g. distributions of evaluation metrics)

- Threshold levels for evaluation might need to be pulled from the dashboard
- Compare test output spec to actual content of the testbed after a test run
- Write little tool to check a test spec for comprehensive usage of all test output in evaluations

3.8.4 Generate test descriptions

cde -o /tmp/betcde/ bet head.nii.gz brain find /tmp/betcde/cde-root -executable -a -type f -a ! -name '*.cde' -a ! -name* '.so'

"depends": [

{ "type": "executable", "path": "$FSLDIR/bin/remove_ext", "dpkg": "fsl-5.0"

}

]

3.8.5 via Tomoyo (Just an idea)

Tomoyo is a lightweight and easy-to-use MAC (Mandatory Access Control) system for Linux, available in stock Linux kernel and tools shipped on Debian. In Learning mode it can easily collect provenance information on what executables/libraries were used for a particular parent process, what files were accessed, environment variables, etc.

Pros: should have virtually no run-time impact

Cons: might require admin privileges to get into learning mode and harvest result information

3.8.6 On SPECs

All nested dicts – except for leaves of the tree. That implies that no list can be used anywhere inside the tree!!

CHAPTER 4

Indices and tables

genindex | modindex | search

Symbols

C

E

J

N

R

S

T

V

W

Z