Autotester

Christopher Serr, Felix Zipperlen, Michael Selzer, Philipp Weißens

Nov 07, 2022

ABOUT AUTOTESTER

Autotester is a tool for automated black-box testing developed at the Institute for Applied Materials (IAM-CMS) of the Karlsruhe Institute of Technology (KIT).

The Source Code can be found on the OpenCarp Gitlab Instance.

CHAPTER

ONE

GOALS AND MOTIVATION

When doing quality assurance for scientific software, it is often necessary to look at the overall results of a scientific simulation or calculation, and compare the outcome carefully with those of previous runs. The validation or comparison of results is often highly specific to the software under test and the actual test case. Sometimes a discrepancy between the expected and the actual result can be a consequence of improvement rather than failure, for example when there was an increase in accuracy. In some cases it might be necessary to allow a discrepancy with certain conditions or boundaries, or there might be the need for more complex analysis of the results to decide if a certain validation fails or succeeds.

This is a perfect use-case for black-box testing as performed with Autotester: Program internals will not be taken into account, instead the focus lies on validating and comparing results with reference data after the program under test as a whole was executed as defined in the test case. Unit tests and other more fine-grained tests instead only enforce certain conditions for small parts of the code like functions or interfaces, and are often tied to specific data formats and programming languages. This is why frameworks for unit testing can not easily be adapted for black-box testing in all cases. Manually performing such tests is of course cumbersome, which is why an automated solution like Autotester with convenient features like summarizing test results, machine-readable results and statistics can help with the task of black-box testing of software in general.

Autotester allows describing and running automated test suits for black-box testing. Each test consists of a command to be executed, and any number of validations to be performed after the command was run successfully. This way, there are no restrictions in terms of programming language or specialized data file formats of the tested application, as long as suitable tooling is available to be integrated with Autotester. In most cases those tools can reuse code or library functions from the scientific framework they work with, and thus understand the specifics of involved data formats. This way, Autotester can provide testing functionality for any software or framework without growing in complexity.

CHAPTER

TWO

LICENSE

CHAPTER

THREE

AUTHORS

- Christopher Serr
- Felix Zipperlen
- Michael Selzer
- Philipp Weißenstein
- Philipp Zschumme

CHAPTER

FOUR

FUNDING

This project is funded by the DFG as part of the project SuLMaSS (Sustainable Lifecycle Management for Scientific Software).

CHAPTER

FIVE

DEPENDENCIES

5.1 Build dependencies

- make
- cmake (optional)
- gcc
- libxml2-dev

Build dependencies can be installed with:

```
apt install make cmake gcc libxml2-dev
```

On systems using the RPM package manager, they can be installed with:

```
yum install make cmake gcc libxml2-devel findutils
```

5.2 Runtime dependencies

- libxml2
- pandoc (when using markdown in test descriptions)
- libxml2-utils (necessary to run the example tests)

Runtime dependencies can be installed with:

```
apt install pandoc libxml2-utils
```

On systems using the RPM package manager, they can be installed with:

```
yum install pandoc libxml2
```

CHAPTER

SIX

BUILD FROM SOURCE

6.1 Build using make

```
make autotester
```

Build with debugging enabled:

```
make autotester DEBUG=true
```

6.2 Build using cmake

```
mkdir build/
cmake -B build/
make -C build/
```

CHAPTER

SEVEN

CREATING TEST CASES

Autotester reads test configurations from ATC files (**A**utotester **T**est **C**onfiguration). Each test has a name and consists of a description of the program under test, how to call the program for this test (including parameters and their values) as well as a number of validations which will be performed after the program was run successfully. Tests can also have a description and a list of variables which can be used to avoid duplicate values in the configuration. ATC files can define one or more tests. A schema description for the XML format can be found `schema/atc.xsd`.

By default, Autotester will recursively search files with the ending `.atc` and process them. Thus, it is possible to define a hierarchy of tests accompanied by input and reference data.

For example, a folder structure for a simple test setup could look like this:

```
.
├── group1
│   ├── compare
│   │   └── reference.dat
│   └── group1.atc
├── group2
│   ├── compare
│   │   └── reference.dat
│   ├── group2.atc
│   └── input
│       └── config.txt
└── main.atc
```

Important: Only one ATC file is allowed per directory!

If Autotester is called in the main directory of this folder structure, Autotester will find all ATC files and run the tests in the respective folder or subfolder. In case the option `-e|--regex [pattern]` is specified, tests will be skipped if their name does not match the pattern.

It is also possible to run a single ATC file using the option `-s|--single [FILE]`:

```
autotester -s examples/example.atc
```

In the test description it's possible to reference the folders `compare/`, `input/` and `output/` which reside in the same directory as the ATC file by using the pre-defined variables `$COMPARE_PATH$`, `$INPUTDATA_PATH$` and `$OUTPUT_PATH$`. The default values for those variables can be overwritten in the file `$HOME/.autotester/config.xml` (see `schema/config.xsd` for a description of the expected format of the configuration file).

7.1 Example ATC file

The following is an example ATC file that can be run with Autotester:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<atc>
  <test name="example system test">
    <description format="markdown">This is a simple test which copies a file</
↪description>
    <program name="cp" />
    <call expected-return-code="0">
      <command>$program$ $program_options$ $file1$ $file2$</command><!-- just copy a␣
↪file using cp -->
      <timeout unit="s" kill-signal="9">1</timeout>
    </call>

    <variables>
      <variable name="program_options">-f</variable>
      <variable name="file1">$COMPARE_PATH$/example_data.txt</variable><!-- the compare␣
↪path is relative to the current
          directory -->
      <variable name="file2">$OUTPUT_PATH$/output.txt</variable><!-- the output folder␣
↪will be created next to the
          .atc file in the same parent folder. The benefit of using is that it will be␣
↪cleaned up by autotester when
          the test succeeds by default (this behaviour can be changed using the options
↪"keeponsuccess" and
          "deleteonerror") -->
    </variables>

    <validations>
      <validation>
        <description>Output and compare file must match</description>
        <call expected-return-code="0">
          <command>cmp $options$ $file1$ $file2$</command>
          <timeout unit="s" kill-signal="9">12</timeout>
        </call>
        <variables>
          <variable name="options">-s -i 10</variable> <!-- for example: ignore the␣
↪first 10 bytes of data -->
        </variables>
      </validation>
    </validations>
  </test>
  <!-- add more tests here if necessary -->
</atc>
```

Find this and other examples in the folder examples/.

7.2 Caveats

- For some characters it is necessary to use escaped sequences in the ATC file, for example use `>` instead of > for piping to a file. Alternatively, use the CDATA directive:

```
<command><![CDATA[pandoc --version > /dev/null]]></command>
```

- Using bash variables or expressions like `$(...)` currently leads to problems because `$` will be interpreted as the start of a variable to be replaced by Autotester.

CHAPTER

EIGHT

RUNNING AUTOTESTER

As mentioned in the previous section, there are two general ways to run Autotester:

1. Recursively process all ATC files in a directory, for example:

```
autotester examples/structured_tests
```

If no folder is specified, Autotester will start in the current directory.

2. Using the option `-s|--single` to run a single ATC file, for example:

```
autotester -s examples/simple_test/main.atc
```

To get information about the command line options, run `autotester --help`. It's also possible to get an XML description of Autotester and it's options by running `autotester --xmlhelp`.

CHAPTER

NINE

RUNNING AUTOTESTER USING DOCKER

Warning: When using Autotester in a gitlab CI runner use the specialized image from docker.opencarp.org/iam-cms/autotester/gitlab instead. It creates output files with the user permissions configured for the gitlab-runner.

The docker image provided at docker.opencarp.org/iam-cms/autotester wraps the Autotester binary and accepts the same options and arguments:

```
docker run docker.opencarp.org/iam-cms/autotester --help
```

To run test suites or single test cases the local folder must be mapped into the container. The following example illustrates how to call Autotester from the current directory (cwd):

```
docker run --rm -v "$(pwd):/shared" docker.opencarp.org/iam-cms/autotester
```

The folder `/shared` is Autotesters' work-directory in the docker container. It should be mounted to the folder where Autotester should run (for example the ATC root folder). This way the output files can be created as usual, given that read and write permissions are available.

Apart from the need to map a folder to the container, the usage is the same as when using Autotester without docker. For example the `-X` switch can be used to let Autotester create the file `output.xml`:

```
docker run --rm -v "$(pwd):/shared" docker.opencarp.org/iam-cms/autotester -X
```

CHAPTER

TEN

OBTAINING TEST RESULTS

Results will be printed to the console (stdout) as well as a short summary at the end. Additionally, the logfiles `log_execution.txt`, `log_validation.txt` and `log_output.txt` will be created unless the option `-n|--nolocallog` is specified. Note that `log_output.txt` will be empty as long as there was nothing printed on stderr by the called program. This log file can be helpful to find out why a certain test failed.

There is also the possibility to let Autotester write an XML output file by specifying the option `-X|--xmloutput`. This output will be written to the file `output.xml` in the directory where Autotester is executed and contains detailed information about the tests and validations as well as their results. The XML format of this output is based on the ATC format. For a detailed description of the format, see `schema/output.xsd`. The file name and location of the output file can be overwritten using the option `-M|--xmloutputfile [FILE]`

CHAPTER

ELEVEN

GENERATING RESULT SUMMARIES

When running Autotester with the `-X` option, a file with a result summary in XML is created. This file can be used to generate a summary in other formats. For both available transformations, XSLT is used, so the command line utility `xsltproc` needs to be installed.

11.1 Generating an HTML result page

Note: When generating the HTML output, the folders 'css' and 'js' need to be available in the correct relative path.

```
mkdir -p build/transformation
xsltproc transformation/html.xsl output.xml > build/results.html
cp -r transformation/css build/transformation/
cp -r transformation/js build/transformation/
```

11.2 Generating a text summary

```
mkdir -p build/transformation
xsltproc transformation/text.xsl output.xml > build/results.txt
```

CHAPTER

TWELVE

HOW TESTS WILL BE PROCESSED BY AUTOTESTER

Tests will be processed by calling the specified command in the <command> section of the test description. In case the command contains variables, they will be replaced first.

- If the test configuration is invalid (invalid syntax, unknown variables etc.), the result will be **ATC_FAIL**.
- If the command fails (expected return code does not match the actual return code), the result will be **EXECUTION_FAIL**. The expected return code defaults to 0 but can be specified using an attribute of the <call> tag in the test description: <call expected_return_code="7">.
- If a timeout for the program call is specified (using the optional <timeout> tag) and the specified duration is exceeded by the running process, the process will be killed and the result will be **EXECUTION_TIMEOUT**.

In case none of the above error events occurred, the validations of the test will be processed. Similar to the processing of tests, the each validation will be started as described in its <call> section. The result of the test will be **VALIDATION_FAIL** in case any of the validation fails, **VALIDATION_TIMEOUT** in case one or more validations exceeded their limit, or **PASS** in case all validations were successful. When there are different validation errors for one test, the most severe failure determines the result status of the test case. **VALIDATION_FAIL** will be considered more severe than **VALIDATION_TIMEOUT**.

12.1 Autotesters return code

The return code of Autotester will be the number of testcases that have failed, independently for the reason they failed, or how many of their validations have failed.