MoPyRegtest: A Python package for continuous integration-friendly regression testing of Modelica libraries

Philipp Emanuel Stelzig[1]

[1]simercator GmbH, Germany, philipp.stelzig@simercator.com

Abstract

Regression testing is a commonly used strategy in continuous integration workflows to ensure reproduceability of outputs. It is widely used in software engineering and model development, including Modelica. In this article we introduce the open source regression testing framework MoPyRegtest written in Python. Its primary focus is to provide Modelica library developers with a simple regression testing tool that features test automation and can be integrated with continuous integration toolchains, in particular for open source developments. In order to simulate the Modelica models for testing analysis, we provide an interface to Modelica simulation tools that have a scripting interface, like *e.g.* `.mos` files. Our current implementation works with OpenModelica. We outline the design and functionality of MoPyRegtest and show its potential usefulness for open source development of Modelica models and libraries.

Keywords: Modelica, regression testing, Python, open source, continuous integration

1 Introduction

The development of Modelica libraries and models requires test and validation. It gives indications to model developers as to the quality and robustness of their developments. Also users request a variety of quality indicators before they choose a certain library. In particular test and test automation, besides a number of other indicators like reputation, community size, update cycles, or response times for support and handling of issues, license conditions and many more. Testing benefits both Modelica library developers and users.

For a user, testing can give an immediate feedback whether a certain Modelica model or library will run on the intended target environment, produce the results its developers intended and which parts of the library have been systematically studied for quality.

For developers, testing carries additional benefits. Unit testing means that debugging does not have to be done with large monolithic models, but rather with smaller individual elements, which helps in locating and isolating issues. Test automation means that developers can quickly detect effects of changes by running test suites, instead of discovering effects manually or, worse, having users discover bugs after releases. Especially with Modelica models, developers implicitely always do tests by running simulations and eventually judging the results as satisfactory. Since Modelica libraries shall feature examples with certain expected results, library developers usually have already created natural candidates to be turned into regression tests. Testing can also open up entirely different development methodologies, like *e.g.* *test-driven development* (TDD) (Beck (2003)). Indeed, if libraries or library elements are intended to model a certain product's or device's physical behavior rather than containing only generic modeling blocks, it is the *test data*, *i.e.* measurement data, that is available first. A model has to be created such that it correctly predicts the reference data. Hence, TDD can be a valid approach for Modelica library development, too. A TDD approach to simulation has been studied in Onggo and Karatas (2016).

Many test criteria employed during testing of Modelica libraries are basically the same as in software development, ranging from static code analysis, reproduceability to integration testing. The one we will focus on here is *reproduceability* of results through the technique of *regression testing*. Wong et al. (1997) summarizes that "[t] he purpose of regression testing is to ensure that changes made to software [...] have not adversely affected features of the software that should not change". Regression testing is an established practice in many Modelica library developments and a number of tools have been proposed or developed. Basically, regression testing for Modelica library development means evaluating whether simulating a certain library element produces a result that is sufficiently close – in a suitable metric – to a reference data set provided by the developer. As we shall see in section 2, a dedicated, lightweight regression test solution for mostly open source library development can be useful. Especially, if it focuses on test automation and integration into *continuous integration* (CI) or *continuous delivery* (CD) toolchains. This is why have developed *MoPyRegtest*[1] (from Modelica Python Regression testing) from a mere helper into an open source solution that can be run with open source tools only.

The outline of this article is as follows. In section 2 we give a broad overview over the many tools for testing Modelica models and their use in open source Modelica library developments. Section 3 summarizes requirements for a regression testing solution derived from the devel-

[1] https://github.com/pstelzig/MoPyRegtest

opers' needs as identified in the previous section. Then, we outline the design of MoPyRegtest and its functionalities. Section 4 covers the currently built-in mathematical metrics that the user can choose from when comparing simulation and reference results. It also shows how we implemented the possibility to provide user-defined metrics. We present a showcase for how to use MoPyRegtest in section 5. Section 6 states conclusions as well as open points, and gives an outlook on future work.

2 State of the art

We outline the state of the art regarding testing and tools for Modelica library development starting with a rough analysis over actual usage of testing solutions. We then derive indicators as to how testing is primarily used today, and where we see that MoPyRegtest can be beneficial to library developers.

2.1 Testing in open source Modelica libraries

In order to have some quantitative indication on the usage of testing solutions for Modelica library development, we did some analysis on the repository collection "Modelica 3rd-party libraries" curated by Dietmar Winkler on GitHub.[2] This collection only covers open source libraries. Hence, from its analysis we cannot derive any insights on testing solutions as they are used in a commercial context. We only considered libraries that had at least one commit since the beginning of 2019. In these repositories we looked for

(C1) any automated tests in the repository's source folder (syntax checking or build/run automation or regression),
(C2) whether there are CI-pipelines set up defining automated tests, *e.g.* in a `.github` or `.gitlab` folder,
(C3) whether there is an automated regression test.

This approach is entirely manual and as such error-prone. To our knowledge, there is no universally adopted practice yet on where to put tests in Modelica libraries and how to formulate or execute them. We found that tests are sometimes hidden deeper inside folder structures and that documentation does not always reveal their existence. As we shall see, there is also some variety when it comes to testing tools used, and sometimes ad hoc testing solutions are implemented that are not obvious to discover. Also, not all developers use GitHub Actions[3] or GitLab CI/CD.[4] Moreover, the nonexistence of test automation, *i.e.* the automated execution of automated tests, does not imply the nonexistence of automated tests themselves. Hence, we might have overlooked or not correctly recognized tests. At this point, we emphasize once more that the numbers in the following Table 1 can at best be interpreted as an indication. We advise against using these numbers in follow-up work. Since we expect at least some corrections to this preliminary and manual analysis, rather than putting the detailed listing here, we put it in an openly accessible repository on GitHub.[5]

Table 1. Estimated relative occurrence of tests in repositories in the GitHub collection "Modelica 3rd-party libraries" as of 14 August 2023. These numbers might not be accurate.

$\exists$ commit since	#(repos)	(C1)	(C2)	(C3)
2019	75	$\approx 28\%$	$\approx 18\%$	$\approx 14\%$
2021	60	$\approx 32\%$	$\approx 20\%$	$\approx 15\%$
2022	45	$\approx 40\%$	$\approx 24\%$	$\approx 20\%$

2.2 Insights

Despite the likely uncertainty in the numbers, we think that one can at least observe tendencies from Table 1:

1. Repositories with more recent activity use more automated tests.
2. Developers prefer other testing strategies first before turning to regression tests.

2.3 Testing tools

In the repositories we looked into, two popular testing tools were BuildingsPy[6,7] for regression testing, and moparser for syntax checking.[8]

BuildingsPy has been developed as part of the Buildings library (Wetter et al. 2014). It supports unit testing and regression testing, but can also orchestrate simulation runs using Dymola[9], OPTIMICA[10] or OpenModelica[11]. The documentation also shows it can visualize simulation results, including regression test results.

moparser is a binary executable by MapleSoft. It is available at MapleSoft's homepage for various platforms. The Modelica Association includes it as the "MapleSim Standalone Modelica Parser" in its Modelica tools list. Several projects use moparser to validate correctness of Modelica syntax as an automated test.

There are also some sophisticated ad hoc regression testing solutions developed for specific projects using various languages. We have found a regression test implementation written in Python as part of the Modelica-Arduino

[2] https://github.com/modelica-3rdparty
[3] https://docs.github.com/en/actions
[4] https://docs.gitlab.com/ee/ci/

[5] https://github.com/pstelzig/modelica-oss-lib-testing-analysis
[6] https://github.com/lbl-srg/BuildingsPy
[7] https://simulationresearch.lbl.gov/modelica/buildingspy/development.html
[8] https://modelica.org/tools.html
[9] https://www.3ds.com/products-services/catia/products/dymola/model-design-tools/
[10] https://help.modelon.com/latest/reference/oct/
[11] https://openmodelica.org/

DOI
10.3384/ecp20443

project.[12] The ModPowerSystems library[13] uses a testing solution developed as part of a bigger utility suite at the RWTH Aachen,[14] also in Python. A shell script solution is used for regression test automation in the PNlib project.[15]

As far as we understood, BuildingsPy, the solution in Modelica-Arduino, and various commercial tools use a sort of maximum deviation metric that checks whether the actual simulation result stays within a “funnel” (BuildingsPy) or a “band” (Modelica-Arduino) around the original result. Mathematically speaking, the “band” amounts to the $\|\cdot\|_{L^\infty}$ norm (in the sense of the space of essentially bounded functions; rather than $\|\cdot\|_{C^0}$ because Modelica simulation result data is usually not time-continuous) applied to the difference of the time-varying functions defined through the reference data and the actual simulation result. Since the timestamps in the reference data and the actual result do generally not coincide, some sort of interpolation technique has to be employed. BuildingsPy uses the pyfunnel[16] module for this; the funnel computation is more than just an $\|\cdot\|_{L^\infty}$-norm, but takes into account also the difference of the timestamps. It regards the reference data as a point cloud $(t_0, y_0), (t_1, y_1), \ldots$ and builds a tolerance area around each (t_i, y_i) datapoint of the reference data. Then, it checks whether the simulated data points fall into these tolerance areas. This requires that timestamps in reference and actual data need to be close, too. As the pyfunnel documentation states, this can make sense to enforce control events to occur at similar times.

Apart from the solutions we found, there are of course other powerful testing solutions available for Modelica library development.

In the open source world, besides the BuildingsPy library, there is the OpenModelicaLibraryTesting.[17] OpenModelica (Fritzson et al. 2020) has extensive library coverage including regression testing for the libraries featured in its package manager.

The tool CSV Result Compare[18] is well known in the Modelica community. Its main purpose is comparing result timeseries files in the `.csv` format. It can also compare `.csv` files recursively by walking through directory trees. It does not allow for test formulation or execution. But it can of course be used to perform result comparison as part of regression testing. Not unlike pyfunnel, it constructs rectangular or ellipsoidal tolerance areas around each datapoint $(t_0, y_0), (t_1, y_1), \ldots$ of the reference result, and then constructs a tube around these tolerance areas defined through an upper and a lower hull curve. csv-compare is implemented in C#, which can cause additional efforts in Linux-based CI toolchains due to its dependencies on .NET or mono.[19]

PySimulator by Pfeiffer et al. (2012) is a simulation and analysis environment with a graphical user interface that can use various different simulators through a plugin infrastructure. Therefore, Asghar et al. (2015) study the use of PySimulator[20] for regression analysis, in particular across different simulation tools, and outline the implementation of a dedicted testing plugin for PySimulator. This plugin uses a simple comparison metric, but features automatic reporting and parallelization of test execution. To our knowledge, PySimulator is no longer actively maintained on GitHub and the last release dates back to 2016. It uses Python 2 which is no longer supported since 2020. Its much broader scope and its plugin dependencies make it difficult to revive for regression testing only.

Commercial tools for regression testing are available from a number of tool vendors. They often come with sophisticated visualization functionality, sometimes directly integrated into Modelica simulation tools, sometimes as standalone products. Generally, except for the case where regression tests are executed by so-called *runners* in a privately managed runtime environment, it is not possible to use commercial, license-bound tools in open source CI toolchains. The term runner refers to an application that executes tests or build tasks for CI applications on resources outside of the CI applications' own infrastructure, and propagates results back to the CI application.

3 Design and functionality

We first sum up the design criteria that guided us in the development of MoPyRegtest. MoPyRegtest shall

(DG1) be a pure testing library,
(DG2) be self-contained,
(DG3) allow for simple formulation of test automation,
(DG4) allow for simple automatic execution with popular CI toolchains,
(DG5) be platform independent,
(DG6) use a popular programming language,
(DG7) allow for user-defined comparison metrics,
(DG8) allow for use with different Modelica tools,
(DG9) integrate with other test automation tools,
(DG10) be usable within open source projects.

3.1 Rationale

Our first design choice was to implement MoPyRegtest entirely in Python. Python is easy to learn, already in use for regression tests in Modelica (BuildingsPy), it is platform independent, easy to automize, and integrates well

[12] https://github.com/modelica-3rdparty/Modelica-Arduino
[13] https://github.com/ModPowerSystems/ModPowerSystems
[14] https://git.rwth-aachen.de/acs/public/simulation/python-for-modelica
[15] https://github.com/AMIT-FHBielefeld/PNlib
[16] https://github.com/lbl-srg/funnel
[17] https://github.com/OpenModelica/OpenModelicaLibraryTesting
[18] https://github.com/modelica-tools/csv-compare
[19] https://github.com/modelica-tools/csv-compare/blob/master/README.md
[20] https://github.com/PySimulator/PySimulator

with popular CI toolchains like GitHub Actions or GitLab CI/CD. Furthermore, the official Python distribution includes the `unittest`[21] framework by default. Hence, it is available to every Python user, it supports test automation, reporting as well as test discovery. For this reason, we use `unittest` as the basis for MoPyRegtest.

Any regression test for Modelica libraries requires a software tool that translates Modelica code into executable simulations that can be run by the testing tool. Since MoPyRegtest shall be usable in open source projects, the natural choice is to use OpenModelica, more precisely the OpenModelica compiler `omc`. However, we want to be able to run MoPyRegtest with other solvers, too. Therefore, we do not call OpenModelica natively from within Python, *e.g.* through OMPython[22] (Ganeson et al. 2012). Instead, we use a file interface and create Modelica script files `.mos` from templates, which we pass to `omc` in order to run simulations. This approach allows for integration with any other simulation tool that supports a file-based scripting interface. Also, it does not introduce any source code dependencies on 3rd party APIs, which could easily break builds or be incompatible for different API versions.

Regression requires the comparison between a reference result and a simulation result produced by a library model. MoPyRegtest does not aim at creating superior or new comparison metrics. Instead of a hard-coded comparison metric, for MoPyRegtest we chose to implement a built-in selection of metrics and give users also the option to define their own metrics. In this fashion, one could even reproduce proven algorithms from other regression or comparison tools like pyfunnel or csv-compare. Calling an external tool for comparison is also possible. For details on the implementation see section 4.

3.2 Architecture and functionality

Conceptually the architecture is very simple and illustrated in Figure 1.

Defining a test in MoPyRegtest is very similar to one in Python's `unittest` module, see Listing 3.1.

In the simplest case, the regression test is a file starting with the prefix `test_<...>.py` to allow for test discovery. It must contain a child class inheriting from `unittest.Testcase`. Inside this class, every single regression test is defined as a method called `test_<...>`, which in its body instantiates a `mopyregtest.RegresstionTest` object like in Listing 3.1. This object is given information on

- where to find the Modelica package to test (`package_folder`),
- which model to test (`model_in_package`),
- where to put the results (`result_folder`),
- [optional] which simulation binary to use from `PATH` (`tool`, default=`"omc"`)
- [optional] which Modelica Standard Library version to use (`modelica_version`, default=`"default"`),
- [optional] a list of Modelica library dependencies loaded before test execution (`dependencies`, default=`None`).

The `mopyregtest.RegresstionTest` object then calls its `compare_result` method with information on

- where to find the reference result as a `.csv` file (`reference_result`),
- [optional] a tolerance threshold which the distance between each individual variable of reference and actual result may not exceed in order to pass (`tol`, default=10^{-7}),
- [optional] a list of variable names for which the comparison metric shall be evaluated (`validated_cols`, default="all variables common in both data sets"),
- [optional] which comparison metric to use (`metric`, default=$\|\cdot\|_\infty$ vector norm on the difference of variable values),
- [optional] which method to use to fill in missing values for timestamps which are not present in either reference or actual result (`fill_in_method`, default=`"ffill"` from `pandas.DataFrame.fillna`).

Remark.

1. In the current implementation, the Modelica STL is treated differently from other dependencies. It is always required by MoPyRegtest. This is just a design choice, because in practice most Modelica models use the STL in one way or the other.
2. The default tolerance has been chosen small for two reasons. First, MoPyRegtest's original scope was to ensure reproduceability during refactorings. Second, choosing a small default value makes it is unlikely that results are judged as being close by accident.
3. The user has to specify the variables to be compared through the `validated_cols` parameter. Theoretically, this could be extended to passing a text file containing the respective variable names, *e.g.* like the `comparisonSignals.txt`[23] proposed for Modelica STL regression testing.

Note that a single file can contain more that one test case like it is common with `unittest`. If a test case definition like in Listing 3.1 is put in a file like `test_mymodel.py`, then all of its test methods are executed by running

```
$ python3 test_mymodel.py
```

[21] https://docs.python.org/3/library/unittest.html

[22] https://github.com/OpenModelica/OMPython

[23] https://github.com/modelica/ModelicaStandardLibrary/files/4270977/SetupForMSLRegressionTesting_2014-01-13.pdf

DOI 10.3384/ecp20443

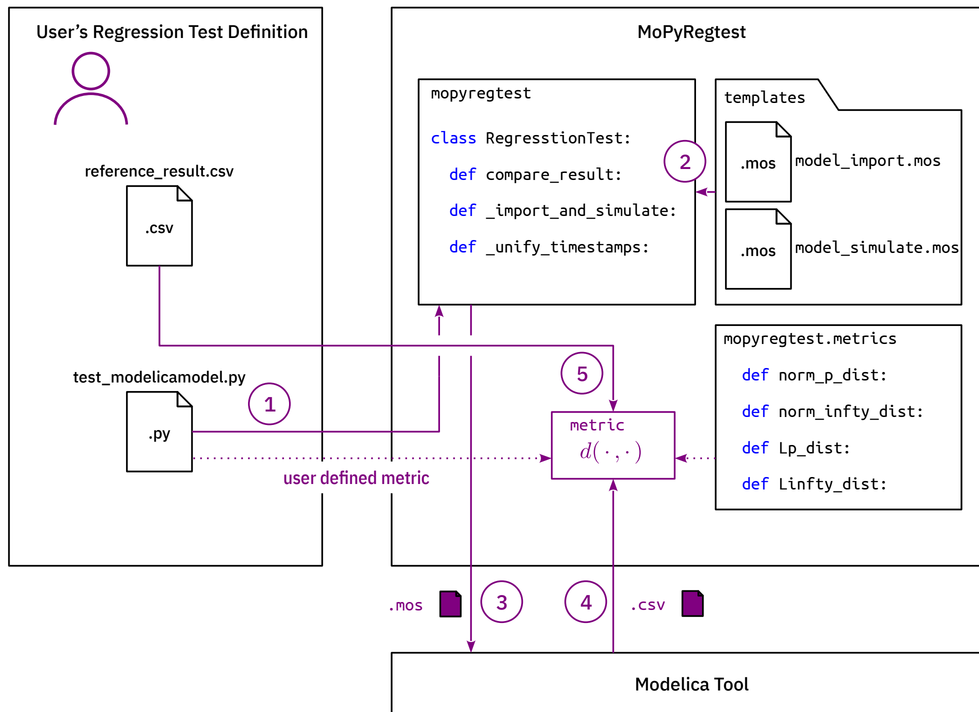

Figure 1. MoPyRegtest architecture with order of test execution steps

Or, if the file is put into a folder structure like required by `unittest` for test discovery,[24] from the structure's root folder one can run

```
$ python3 -m unittest
```

to discover and execute all `unittest` cases. See section 5 for a complete example, which is also contained in the MoPyRegtest implementation.

The call to `tester.cleanup()` in Listing 3.1 is optional. The examples in MoPyRegtest do not call the `cleanup` method, because automatic deletion must always be handled with extreme care.

4 Comparison metrics for regression analysis

Generally speaking, choosing a metric to measure the closeness, or rather the distance of a simulation result from a reference data set, is problem specific. Therefore, we designed MoPyRegtest to give the user full control over the metric which is used in a regression test. The current MoPyRegtest implementation also features a set of predefined metrics.

4.1 Motivation

In section 2 we have outlined strategies that in some way or the other ("funnel" or "band") require the *values* of reference and actual simulation result to be close. This might not always fit. Both data sets are in fact time-discrete representations of functions depending on time, which are known to us only through their values at the timestamps in the respective datasets. In our case, `.csv` files for reference result and the actual simulation run.

Mathematically speaking, there is a wide variety of norms and metrics to measure "closeness" for such time-dependent functions. Two functions may be close in one metric, but not in another one.

A classic example is Gibb's phenomenon in Fourier series approximations for discontinuous functions,[25] most prominently the approximation of a Heaviside step-function $h : [0,2\pi] \mapsto \mathbb{R}$ (step at π) with partial sums of its Fourier series. It will overshoot at the jump discontinuity. However, with the basis functions $\{t \mapsto e^{int} : n \in \mathbb{Z}\}$ forming an orthonormal basis of $L^2([0,2\pi])$ equipped with the canonical norm $\|f\|_{L^2([0,2\pi])} := \left(\int_0^{2\pi} |f(t)|^2 \mathrm{d}t\right)^{\frac{1}{2}}$, the partial sums of the Fourier expansion will converge in that norm (and for a suitable subsequence also pointwise *almost everywhere*). Hence, in this case it is more meaningful to use the $\|\cdot\|_{L^2([0,2\pi])}$ norm as a measure for closeness instead of the common "band" notion.

Another example are events for state-discrete variables, which might occur at slightly different times in reference and actual simulation result. See section 2 and how pyfunnel and csv-compare address this issue.

In order to give users full flexibility in choosing the right metric for the regression test formulation, we allow users to define their own metrics (subsection 4.2) or choosing from a set of predefined metrics (subsection 4.3).

[24] https://docs.python.org/3/library/unittest.html#unittest-test-discovery

[25] https://en.wikipedia.org/wiki/Gibbs_phenomenon

Listing 3.1. Test case definition with MoPyRegtest and pre-defined comparison metric

```
import unittest
import mopyregtest

class TestUserDefinedMetrics(unittest.TestCase):
  def test_modelicamodel(self):
    tester = mopyregtest.RegressionTest(
      package_folder="/path/to/mylibrary",
      model_in_package="MyModel",
      result_folder="/path/to/result/folder/MyModel",
      tool="omc", modelica_version="4.0.0", dependencies=None)

    tester.compare_result(
      reference_result=str("/path/to/reference/result/MyModel_res.csv"),
      metric=mopyregtest.metrics.Lp_dist,
      validated_cols=["myvar1", "myvar2"], tol=1e-8, fill_in_method="interpolate")

    tester.cleanup()

    return


if __name__ == '__main__':
  unittest.main()
```

4.2 Implementation

In our situation, both the reference data and the actual simulation data that a user compares in a regression test are given as `.csv` files. They contain columns of data for individual variables at time-discrete timestamps. The timestamps are identical for all variables within one `.csv` file. In general, the timestamps in different `.csv` files do not coincide. A `.csv` file from a simulation run[26] might look like in Table 2.

As it can be seen, timestamps might have multiple occurrences, depending on whether events occurred at that time (in one or more variables).

Metric definition We require a user-defined metric to be a function d that

- takes two `numpy.ndarray`[27] arrays of *identical* shape $(N_{\text{tstamps}}, 2)$, say r_{ref} and r_{act} with
- both r_{ref} and r_{act} having the timestamps as the first column and the values *of one result variable* in the second column, and then
- returns a nonnegative real number $d(r_{\text{ref}}, r_{\text{act}})$.

Then, a user can simply define metrics by passing a function handle, or even *in situ* using lambda functions. In software development, a lambda function refers to an anonymous, *i.e.* an unnamed function that can be defined *ad hoc* and in place. For example,

```
metric=lambda r_ref, r_act: numpy.linalg.
    norm(r_ref[:, 1] - r_act[:, 1], ord=1)
```

[26] https://github.com/pstelzig/MoPyRegtest/blob/master/examples/test_user_defined_metrics/references/SineNoisy_res.csv

[27] https://numpy.org/

which amounts to taking the $\|\cdot\|_1$ vector norm in $\mathbb{R}^{N_{\text{tstamps}}}$ on the difference in values for all timestamps. Then, the distance according to this metric is computed for every variable (defined through the `validated_cols` parameter in `RegressionTest.compare_result`).

Note that in this fashion it is entirely up to the user if he wants to employ absolute or relative error measures in a comparison metric. One could also write

```
metric=lambda r_ref, r_act: mopyregtest.
    metrics.Lp_dist(r_ref, r_act)/
    mopyregtest.metrics.Lp_norm(r_ref)
```

to compute a relative error in the L^2-norm (Lebesgue space norm) weighted by the L^2-norm of the reference result. Here, $p = 2$ is the default value in `Lp_dist` and `Lp_norm`.

Despite being very convenient for the user, we require the data r_{ref} and r_{act} to have identical shape and, in order for computations to make sense, have identical timestamps. Which is generally not the case. For instance, when data sampling rates in reference result and actual result are different.

Timestamp unification One possibility would be to leave it to the user to provide meaningful interpolation for data at missing timestamps. To make it easier for the user, we have implemented the method `RegressionTest._unify_timestamps`. This function is always called in `RegressionTest.compare_result` before the metric is evaluated.

It takes both the reference result r_{ref} and the actual result r_{act} with their timestamps $\mathscr{T}_{\text{ref}} = [t_{\text{ref},0}, \ldots, t_{\text{ref},N_{\text{ref}}}]$ and $\mathscr{T}_{\text{act}} = [t_{\text{act},0}, \ldots, t_{\text{act},N_{\text{act}}}]$ and creates a union $\mathscr{T}_{\text{unified}}$ that

DOI
10.3384/ecp20443

Table 2. Example for a `.csv` result from a Modelica tool run.

time	sine.y	uniformNoise.y	y	uniformNoise.state[1]
0	0	0.289372473723095	2.89372473723095E-05	363258270
0	0	0.289372473723095	2.89372473723095E-05	363258270
0	0	0.289372473723095	2.89372473723095E-05	-2054081690
0.02	0.125333233564304	0.289372473723095	0.125362170811677	-2054081690
0.04	0.248689887164855	0.289372473723095	0.248718824412227	-2054081690
0.05	0.309016994374947	0.289372473723095	0.30904593162232	-2054081690
0.05	0.309016994374947	0.837498257278269	0.309100744200675	14228464
0.06	0.368124552684678	0.837498257278269	0.368208302510406	14228464

- contains every timestamp from the union of both $\mathscr{T}_{\text{ref}}$ and $\mathscr{T}_{\text{act}}$ interpreted as sets (*i.e. no* multiplicities) and
- repeats each timestamp as often as the maximum of its occurrences in $\mathscr{T}_{\text{ref}}$ and $\mathscr{T}_{\text{act}}$.

For both r_{ref} and r_{act}, data rows are repeatedly added for every timestamp from $\mathscr{T}_{\text{unified}}$ until in the such extended r_{ref} and r_{act} each timestamp's multiplicity matches the one in $\mathscr{T}_{\text{unified}}$. The newly added values are initalized with NaN. Then, both r_{ref} and r_{act} are sorted along the timestamp axis, using a stable sorting algorithm that preserves the original order of timestamps.

This leaves the question of interpolating the such added NaN values. To this end, we simply use the strategies offered by `pandas.DataFrame.fillna`[28] and `pandas.DataFrame.interpolate`.[29] The user can choose between these strategies, see the options in subsection 3.2.

Remark. A valid question is whether, instead of unifying the timestamps, it would be easier to define the metric $d(\cdot,\cdot)$ for results r_{ref} and r_{act} of different shapes $(N_{\text{ref}},2)$ and $(N_{\text{act}},2)$, respectively. And then, if needed, require interpolation as part of the metric implementation. *E.g.* for integral based metrics, timestamp unification is not needed. Only at the numerical integration points both r_{ref} and r_{act} need to be evaluated. Also, there is some risk as to the interpolation error introduced by the timestamp unification. We opted for the timestamp unification for three reasons:

1. It is more convenient for the user and allows for shorter metric definitions.
2. Users can still use any interpolation they want inside the metric implementation.
3. It has the benefit of being able to write out both r_{ref} and r_{act} into a single `.csv` result for visual comparison.

In the future, we might make the now always executed call to `RegressionTest._unify_timestamps` optional, putting such users who want back in control of handling different timestamps in their metric definitions themselves.

[28] `https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.fillna.html`

[29] `https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.interpolate.html`

4.3 Built-in metrics

MoPyRegtest comes with the following built-in metrics in the module `mopyregtest.metrics`. With some slight abuse of notation:

`mopyregtest.metrics.norm_p_dist`

$$d(r_{\text{ref}},r_{\text{act}}) := \left\| r_{\text{ref}}[:,1] - r_{\text{act}}[:,1] \right\|_p$$

for some $p \in \{1,2,\dots\}$ where $\|v\|_p = (\sum_{i=0}^{N-1} |v_i|^p)^{\frac{1}{p}}$ is the canonical p-norm in $\mathbb{R}^N$.

`mopyregtest.metrics.norm_infty_dist`

$$d(r_{\text{ref}},r_{\text{act}}) := \left\| r_{\text{ref}}[:,1] - r_{\text{act}}[:,1] \right\|_\infty$$

where $\|v\|_\infty = \max\{|v_0|,\dots,|v_{N-1}|\}$ is the canonical maximum norm in $\mathbb{R}^N$.

`mopyregtest.metrics.Lp_dist`

$$d(r_{\text{ref}},r_{\text{act}}) := \left(\sum_{i=0}^{N_{\text{tstamps}}-1} (t_{i+1}-t_i)\cdot \left| r_{\text{ref}}[i,1] - r_{\text{act}}[i,1] \right|^p \right)^{\frac{1}{p}}$$

for some $p \in \{1,2,\dots\}$ where

$$[t_0,\dots,t_{N_{\text{tstamps}}}] = r_{\text{ref}}[:,0] = r_{\text{act}}[:,0]$$

are the unified timestamps of r_{ref} and r_{act}. This is the common L^p-norm $\|f\|_{L^p} = \left(\int_{t_0}^{t_{N_{\text{tStamps}}}} |f|^p \mathrm{d}t\right)^{\frac{1}{p}}$ (Lebesgue space norm) when viewing r_{ref} and r_{act} as piecewise continuous functions.

`mopyregtest.metrics.Linfty_dist` When again viewing r_{ref} and r_{act} as piecewise continuous functions, the L^∞-norm (norm of essentially bounded functions) reduces to the canonical $\|\cdot\|_\infty$ maximum norm on the function's values. Hence, it returns the same value as `mopyregtest.metrics.norm_infty_dist`. This metric has been included for notational consistency.

Remark. The values of p are chosen as integers other than the usual real $p \in [1,\infty)$ because we use `numpy.linalg.norm` which requires the order p to be of type `int` rather than `float`.

5 Showcase

As a showcase we present the example `test_user_defined_metrics` that is included MoPyRegtest's sources.[30]

5.1 Test definition

In this example we want to formulate a regression test that validates during library development, whether a certain Modelica library model stays close to a reference data set. For the showcase, the reference data set is $[0,1] \ni t \mapsto \sin(2\pi t) + 10^{-4} \cdot \mathrm{noise}(t)$. A Modelica model[31] has been created and run with OpenModelica to create the `.csv` reference result.[32] This model uses `Modelica.Blocks.Sources.Sine` and `Modelica.Blocks.Noise.UniformNoise`. The Modelica library model to be tested is the original `Modelica.Blocks.Sources.Sine` itself (without the noise).

The test has the folder structure

```
examples
└─ test_user_defined_metrics
   ├─ __init__.py
   ├─ test_user_defined_metrics.py
   └─ references
      └─ SineNoisy_res.csv
```

The `__init__.py` turns the folder `test_user_defined_metrics` into a Python package for test discovery. That is, if `python3 -m unittest` would be called from the parent directory `examples`, all `unittest` test definitions in `test_user_defined_metrics` would be executed. The entire test definition in shown in Listing 5.1.

5.2 Test automation

The test is automated "for free" because `unittest` features automated test execution and test discovery. The output is shown in Listing 5.2.

5.3 Automated test execution

Modern continuous integration toolchains like GitHub Actions or GitLab CI/CD allow the automated execution of tests triggered by certain events. With GitHub Actions for instance, one can automate test execution of Python code on `push` events to a GitHub repository. In that case, the respective user-defined job, say `python-test.yml`, in the repository's `.github/workflows/` folder is executed. The GitHub Actions documentation[33] explains how.

In our case, the execution of a regression test definition using MoPyRegtest, *e.g.* the one above, requires a suitable Modelica simulation tool. There is the option to install the OpenModelica compiler `omc` and the Modelica Standard Library as a step in the job definition, as well as the other dependencies of MoPyRegtest. This however would require significant computational resources and consume valuable usage time for GitHub Actions. The same goes for other continuous integration pipelines.

Another option is to execute the tests based on a Docker[34] image. Here, one could use the OpenModelica docker image tagged `v1.21.0-minimal` from dockerhub[35] (or respective newer versions) with the pre-installed `omc`. For instance, a GitHub Action that executes the same test definition as in Listing 5.1, but runs it in a docker container based on the OpenModelica `v1.21.0-minimal` image is shown in Listing 5.3.

In this fashion, one can easily implement test automation in an open source Modelica library development on GitHub. All that is needed are test definitions in MoPyRegtest like in Listing 5.1 and a GitHub Action like in Listing 5.3 that executes the test definitions. Either automatically, *e.g.* following push events, or manually triggered. Both developers and users can then review the test results in the repository's Actions tab.

6 Conclusions

We have outlined why regression testing is important in Modelica library development. Then we gave a rough overview over how test and test automation is being used with open source Modelica library development and identified some potential trends. We have also identified concrete tools used for regression testing in the open source Modelica library community. We then formulated the rationale why a continuous integration-friendly testing solution like MoPyRegtest could be of value for the community and described its design and functionality. We highlighted in detail how we implemented the possibility for users to define their own comparison metrics for regression tests. Then we presented a showcase that is included in MoPyRegtest.

MoPyRegtest is work in progress and still under development. It has not been investigated yet how it could integrate with Modelica simulation software other than Open-

[30] `https://github.com/pstelzig/MoPyRegtest/tree/master/examples/test_user_defined_metrics`

[31] `https://github.com/pstelzig/MoPyRegtest/blob/master/examples/test_user_defined_metrics/SineNoisy.mo`

[32] `https://github.com/pstelzig/MoPyRegtest/blob/master/examples/test_user_defined_metrics/references/SineNoisy_res.csv`

[33] `https://docs.github.com/en/actions/automating-builds-and-tests/building-and-testing-python`

[34] `https://www.docker.com/`

[35] `https://hub.docker.com/r/openmodelica/openmodelica/tags`

DOI 10.3384/ecp20443

Listing 5.1. Showcase `test_user_defined_metrics.py`

```
# Preparing the dependencies ####################################################
import unittest
import pathlib
import mopyregtest
import functools

# Define the test ################################################################
# Example here for a Ubuntu environment with OpenModelica

class TestUserDefinedMetrics(unittest.TestCase):

    # Testing user defined metrics on a Modelica simulation result against a noisy
        reference result
    def test_Sine(self):
        tester = mopyregtest.RegressionTest(
            package_folder=pathlib.Path.home() / \
                               ".openmodelica/libraries/Modelica 4.0.0+maint.om/",
            model_in_package="Modelica.Blocks.Sources.Sine",
            result_folder=pathlib.Path(__file__).absolute().parent / \
                              "Modelica.Blocks.Sources.Sine",
            modelica_version="4.0.0",
            dependencies=None)

        # Comparing results
        tester.compare_result(
            reference_result=str(pathlib.Path(__file__).absolute().parent / \
                "references/SineNoisy_res.csv"),
            metric=functools.partial(mopyregtest.metrics.Lp_dist, p=2),
            validated_cols=["y"], tol=2e-3, fill_in_method="interpolate")

        return


if __name__ == '__main__':
    unittest.main()
```

Listing 5.2. Output of `test_user_defined_metrics.py`

```
$ python3 -m unittest

Testing model Modelica.Blocks.Sources.Sine
Simulating model Modelica.Blocks.Sources.Sine using the simulation tools: omc
Using simulation tool omc
Comparing simulation result /home/user/mopyregtest/examples/test_user_defined_metrics/
    Modelica.Blocks.Sources.Sine/Modelica.Blocks.Sources.Sine_res.csv and reference /home/
    user/mopyregtest/examples/test_user_defined_metrics/references/SineNoisy_res.csv
Comparing column "y"
.
----------------------------------------------------------------------
Ran 1 test in 2.679s

OK
```

Listing 5.3. GitHub Action to execute `test_user_defined_metrics.py` as part of MoPyRegtest's GitHub repo

```
name : Example job for Modelica library regression testing
on: [workflow_dispatch]
jobs:
  examples-test:
    runs-on: ubuntu-latest
    container: openmodelica/openmodelica:v1.21.0-minimal
    steps:
      - name: Install dependencies
        run: |
          apt-get -qq update
          apt-get -qq --no-install-recommends install python3 python3-pip git
          pip install numpy pandas
      - name: Install Modelica STL 4.0.0
        run: |
          echo "installPackage(Modelica, \"4.0.0+maint.om\", exactMatch=true);" >
              installModelicaStl.mos && omc installModelicaStl.mos
      - name: Install MoPyRegtest with tag v0.2.1
        run: |
          git clone https://github.com/pstelzig/MoPyRegtest.git mopyregtest
          cd mopyregtest
          git checkout v0.2.1
          pip3 install --user .
      - name: Run examples
        run: |
          cd mopyregtest/examples/test_user_defined_metrics
          python3 test_user_defined_metrics.py
```

Modelica. Also, it has no inherent reporting functionality except what is provided by Python's `unittest`. Furthermore, reference results need to be given as `.csv` files, whereas in the Modelica community result files are usually `.mat`, making reference result files bigger than they need to be. The timestamp unification has proven reliable in our use so far, but other interpolation techniques, as outlined in the respective remark in section 4, could perform better in certain scenarios. Multiple tests within a single test class are executed sequentially at the moment, despite being independent. Execution time could be saved by running tests in parallel. Finally, we have shown the feasibility of integrating MoPyRegtest with popular continuous integration toolchains like GitHub Actions.

Acknowledgements

MoPyRegtest is developed by Philipp Emanuel Stelzig as a private project. At simercator it is used for automating regression tests of inhouse Modelica libraries. The author would like to thank the reviewers for their helpful comments regarding the state of the art overview.

References

Asghar, Adeel et al. (2015). "Automatic regression testing of simulation models and concept for simulation of connected FMUs in PySimulator". In: *11th International Modelica Conference*. 118. Linköping University Electronic Press, pp. 671–679.

Beck, Kent (2003). *Test-driven development: by example*. Addison-Wesley Professional.

Fritzson, Peter et al. (2020). "The OpenModelica integrated environment for modeling, simulation, and model-based development". In: *Modeling, Identification and Control* 41.4, pp. 241–295.

Ganeson, Anand Kalaiarasi et al. (2012). "An OpenModelica Python Interface and its use in PySimulator". In: *9th International Modelica Conference*. Linköping University Electronic Press, pp. 537–548.

Onggo, Bhakti Stephan and Mumtaz Karatas (2016). "Test-driven simulation modelling: A case study using agent-based maritime search-operation simulation". In: *European Journal of Operational Research* 254.2, pp. 517–531.

Pfeiffer, Andreas et al. (2012). "PySimulator – A simulation and analysis environment in Python with plugin infrastructure". In: *9th International Modelica Conference*. Linköping University Electronic Press, pp. 523–536.

Wetter, Michael et al. (2014). "Modelica Buildings library". In: *Journal of Building Performance Simulation* 7.4, pp. 253–270. DOI: 10.1080/19401493.2013.765506.

Wong, W Eric et al. (1997). "A study of effective regression testing in practice". In: *PROCEEDINGS The Eighth International Symposium On Software Reliability Engineering*. IEEE, pp. 264–274.