# Doping Tests for Cyber-Physical Systems – Tool

Sebastian Biewer[1] Pedro R. D'Argenio[2,3,1] Holger Hermanns[1,4]

[1]Saarland University, Saarland Informatics Campus, Saarbrücken, Germany
[2]Universidad Nacional de Córdoba, FAMAF, Córdoba, Argentina
[3]CONICET, Córdoba, Argentina
[4]Institute of Intelligent Software, Guangzhou, China

This article companions the work presented in *Doping Tests for Cyber-Physical Systems* by Sebastian Biewer, Pedro D'Argenio and Holger Hermanns, published in the *ACM Transactions on Modeling and Computer Simulation* 2021.

## 1 Introduction

To demonstrate that the theory, that is presented in the article mentioned above, is implementable, we provide a prototype implementation of a testing framework written in Python. The core implementation contains several abstract classes leaving the choice of value domains, distance functions, the system under test and test case selection unspecified. We show examples on how to instantiate the framework by means of subclasses for concrete values and distances and demonstrate different kinds of test case selections using an easy to understand program in Section 4. The Volkswagen example is presented in Section 5 and demonstrates how a concrete test case selection can be tailored to the needs of the diesel emissions scandal.

To overcome the problem of human imprecisions, we use a *monitoring* technique. A monitor can read the inputs and outputs of a system in order to detect incorrect behaviour of the system. In contrast to testing, the inputs are not provided by the test, but the system is monitored during normal operation. Monitors can be either online (evaluation is done while inputs are still received) or offline (observed behaviour is evaluated after the observation). A monitor can easily be extended to a test by controlling the environment providing the inputs to the system. Moreover, monitoring in a controlled environment allows for targeted testing under consideration of human imprecisions—typically arising with experiments with passenger cars. We extended our testing framework to support offline monitoring by specifying a system under test and a test case selection that are tailored to analyse recorded traces.

The *Nissan test* from the companioned article exploits the flexibility of monitoring in a controlled environment. Like for testing, the environment is controlled by some (ideal) test cycle generated by a deterministic $\Omega_{\text{case}}$ and $\Omega_{\mathsf{In}}$ for test case selection. However, we could observe deviation from the ideal environment due to human imprecisions when driving the car. The actual values of the (speed) inputs and the ($NO_x$) outputs are recorded and afterwards processed by means of offline monitoring. This results in a test outcome, that is either passed or failed—or trivially passed, if the input threshold has been violated. In the main article, we refer to this technique as *two-step approach*. We present the Nissan example to demonstrate our two-step approach in Section 6.

The repository is organised as follows.

- `doping_test` contains all classes for the abstract testing framework
- `doping_monitor` is the extension of `doping_test` to an abstract offline monitor
- `examples` contains several examples where values have type `float` and distances are defined as $d(t_1, t_2) := |\text{last}(t_1) - \text{last}(t_2)|$. Most examples are easy-to-understand programs. We also present the experiments from the companioned article in `examples/regions` and `examples/nissan`.

## 2 Abstract Testing Framework

The abstract testing framework is defined in `doping_test`. The main class is `DT`, which implements the algorithm $\mathsf{DT}_b$ from the companioned article. It works with instances of `SystemUnderTest` and `TestCaseSelection`, which are abstract interfaces for a system under test $\mathcal{I}$ and the selection of $\Omega_{\text{case}}$ and $\Omega_{\mathsf{In}}$. The outputs from $\mathcal{I}$ are checked by an instance of `AcceptanceChecker`. The default implementation of `AcceptanceChecker` determines if an output $o$ received from the system under test is clean w.r.t. to the test history $h$, i.e. it checks whether $o \in \mathsf{acc}_b(h)$.

Symbols are instances of `Input` or `Output`, which wrap a concrete value and allow for convenient distinction between inputs and outputs throughout the framework. The classes `Distance`, `ValueSet`, `Standard` and `Trace` define interfaces for distance functions, sets of values, a standard $\mathcal{S}$ and traces. A concrete implementation for `ValueSet` is `EmptySet` for the empty set and `SimpleTrace` wrapping an array of symbols for `Trace`. We provide an implementation for random testing in class `RandomTestCaseSelection`, which picks one of the three choices of $\mathsf{DT}_b$ randomly with different probabilities. For the selection of test inputs it tries to pick values that do not immediately let the test pass because it is too far away from a standard input. Beyond that, the choice of inputs is random with the distribution defined by the concrete subclass of `ValueSet` being used. The result of a test is by objects of class `TestResult`, which carry the information about whether the test passed or failed and the trace that has been tested. If the test fails, it also stores the standard trace which the system under test violated.

## 3 Abstract Doping Monitor

We extended the doping test framework by `doping_monitor` for offline monitoring of traces. The central class is `RecordedTrace`, which can load recorded traces from disk and can provide a system under test $\mathcal{I}$ of class `MonitorUnderTest`, which simulates exactly the recorded behaviour when provided with the recorded inputs. The correct choice of inputs is guaranteed by the provided instance of class `MonitorTestCaseSelection` (defining $\Omega_{\text{case}}$ and $\Omega_{\mathsf{In}}$), which makes $\mathsf{DT}_b$ pick exactly the inputs that were recorded or to wait for an output if an output has been recorded. For constructing a standard $\mathcal{S}$ from (possibly several) recorded traces, an array of recorded traces can be passed to the constructor of `MonitoredStandard`. For using the framework it is necessary to provide an implementation to `RecordedTrace` for parsing the encoded symbols in a file.

## 4 Example: Noisy Mirror

The class `NoisyMirror` in `examples/numbers` implements a function that randomly either returns the input or twice as much. However, the program is doped. A shady programmer observed that the official authority testing the software uses inputs that have at most two decimals. This observation seduced him to add a check if more than two decimals are passed to the program in order to output values up to four times as much as the input. The official standard test commits positive integers in ascending order to the program (i.e. 1, 2, 3, ...). The resulting traces are provided by `NoisyStandard`. The sets of values used in our examples are closed ranges of floats, hence we extended `ValueSet` to `NumberRange`. We use the past-forgetful distance function $d(t_1, t_2) := |\text{last}(t_1) - \text{last}(t_2)|$ for both inputs and outputs and it is implemented in `LastComponentDistance`. We test a contract with $\kappa_{\mathsf{i}} = 0.2$ and $\kappa_{\mathsf{o}} = 0.5$.

The program in `randomTest.py` uses the random test case selection from `doping_test` to test the NoisyMirror. Running the test several times shows that it sometimes passes and sometimes fails. In `predefinedTest.py` we demonstrate how to feed $\mathsf{DT}_b$ with inputs that have been hand-picked before the test. It uses `ManualTestCaseSelection` to generate one of four tests. We show two well defined tests: *good-test1* passes, because inputs always have one decimal and *good-test2* eventually has more than two decimals and mostly fails. The other two tests are badly designed as they both pass, because they are trivially satisfied. *bad-test1* provides an input that is too far away from a standard and the test will pass although outputs are suspiciously large. In *bad-test2*, the test waits for an output when it is supposed to provide an input, which leads to an input distance of infinity, so the test passes trivially, too. The third example is in `monitoring.py`, which

shows how offline monitoring works. We construct the same standard as for the previous examples by using a `MonitoredStandard` with the traces recorded in `run-std1.txt` and `run-std2.txt`. The trace recorded in `run-fail.txt` will fail (because the output in line 16 has no standard output in $\kappa_{\mathsf{o}}$-distance) and for `run-pass.txt` $\mathsf{DT}_b$ passes. The files can be parsed by instances of `RecordedNumberTrace`, which are than handled like normal traces.

# 5 Example: Volkswagen

In this example, the implementation under test is a toy version of a VW engine control unit (ECU) with a tampered emission cleaning system. Its implementation is inside `ECU`, which is a subclass of `SystemUnderTest`. It is explained in the main article that the VW case is based on a pair of piecewise linear functions. These functions, called nedc_lower and nedc_upper, are sampled in intervals of one second and are stored in two text files. On initialisation of an `ECU` object, these files are loaded into memory. Method `pass_input` is called by the testing framework if case 2 of Algorithm $\mathsf{DT}_b$ is chosen. It is the responsibility of the IUT to return an output if there is some output that the IUT wants to deliver to the testing framework. If an output is returned, the input must be dropped. Otherwise, the ECU object adds the input to a trace, which is stored in variable `history`. For case 3 of the algorithm, method `receive_output` is called. Similar to the trace $\sigma_S \coloneqq i_1 \cdots i_{1180}\, o_S\, \delta\, \delta\, \delta \cdots$ from the main article, there shall be only one output after 1180 seconds. Hence, the method checks if the length of the current history is of length 1180 and only then computes if the evolution of the travelled distance when driving according to the trip stored in `history` is in the white or grey region. The amount of $\mathrm{NO}_x$ is simulated according to the difference of the totally travelled distance of the current trip and the distance of a full NEDC. If the white region was left, however, this value is multiplied by a (arbitrarily picked) factor of 7.3.

In file `examples/regions/randomTest.py` test case selection and the contract, with $\kappa_{\mathsf{i}} = 15$ and $\kappa_{\mathsf{o}} = 180$, is set up. The thresholds are stored in variables `input_threshold` and `output_threshold`, respectively. $\mathsf{In} = \mathsf{Out} = \mathbb{R}_{\geq 0}$ is implemented by classes `Input` and `Output`. Similarly to the example above, we use a class `NumberRange` to encode ranges of these values. $d_{\mathsf{In}}$ ($d_{\mathsf{Out}}$) is defined as `input_distance` (`output_distance`) by instantiating a class called `LastComponentDistance` for past-forgetful distance functions. A virtual NEDC trip is stored in `nedc_standard.txt` with an invented output of 50 *mg/km*. The trip is encoded in a framework-specific format, which can be parsed by class `DynamometerTrace`. This NEDC trip is used as the only standard trace for this experiment. Standard behaviour is represented by the abstract class `Standard`. The monitor part of the framework provides support for constructing `Standard` objects by initialising class `MonitoredStandard` with a list of traces. The standard for this example is constructed by initialising `MonitoredStandard` with the single NEDC trace. The IUT is an instance of the class implementing a toy version of the the VW cleaning system as described above.

The bound on the length of the tests is given by 1180 inputs plus one output. Test case selection is provided by `RandomInputSelection` and provides the following test case selection (where $\mathit{rand}_{\mathbf{unif}}\ R$ samples uniformly from range $R$):

$$\Omega_{\text{case}}(h) = \begin{cases} 2 & \text{, if } |h| \leq 1180 \\ 3 & \text{, if } |h| = 1181 \end{cases} \qquad \Omega_{\mathsf{In}}(h) = \mathit{rand}_{\mathbf{unif}}\ [\max(0, \mathsf{last}(h) - \kappa_{\mathsf{i}})\ ,\ \mathsf{last}(h) + \kappa_{\mathsf{i}})]$$

In this implementation, the test case selection relies on the implementation of $d_{\mathsf{In}}$ and $\mathsf{In}$ in order to uniformly sample from the subset $[\max(0, \mathsf{last}(h) - \kappa_{\mathsf{i}})\ ,\ \mathsf{last}(h) + \kappa_{\mathsf{i}})]$ of $\mathsf{In}$. The evaluation of the predicate $o \in \mathsf{acc}_b(h)$ is done by objects of class `AcceptanceChecker`. Objects of this class rely on the implementation of $d_{\mathsf{In}}$ to get all standard traces with a maximum input distance of $\kappa_{\mathsf{i}}$. Similarly, it uses the implementation of $d_{\mathsf{Out}}$ to check whether an output shall be accepted (i.e., it uses $d_{\mathsf{Out}}$ to find a $\sigma$ as required by the existential quantifier in the last line of $\mathsf{acc}_b$). Algorithm $\mathsf{DT}_b$ is implemented in class `DT`, which is instantiated with the test case selection (i.e., $(\Omega_{\text{case}}, \Omega_{\mathsf{In}})$), the acceptance checker (i.e., $\mathsf{acc}_b$) and the IUT (i.e., the VW inspired toy ECU). Running $\mathsf{DT}_b$ with these parameters will mostly lead to a **fail**, because it is very likely to enter one of the grey areas early during these test. In our experiments, we had to change $\kappa_{\mathsf{i}}$ to 4 in order to see tests passing regularly.

Class `DT` offers a method `test`, which is to be called with the boundary $b$. Internally, this methods iterates $b$ times to call a method `advance`. There, the test case selection is asked to suggest one out of three choices. Picking the first choice lets the test **pass**. Choice 2 asks the test case selection for an input $i$. This input is passed to the IUT by calling `pass_input` on the interface to the IUT. The interface checks, if the IUT is offering an output. If this is the case, it discards the input and returns the output to `DT`. Otherwise, the input is processed by the IUT and `None` is returned. `DT` checks if the IUT returns an output $o$. If it does, $o$ is appended to $h$ (called `history` in the implementation). Then, `DT` asks $\mathsf{acc}_b$ (`output_verifier`), for a counter example that proves that $o$ violates robust cleanness (by finding $\sigma \in \mathsf{traces}_*(\mathcal{S}_\delta)$ that violates the second condition). If such a counter example is found, the test **fail**s and otherwise `advance` returns `None` indicating the test result is still inconclusive. If the IUT does not return an output, the input passed to the IUT is added to $h$ and `None` is returned. The third choice of the `DT` class asks the interface to the IUT for an output. It either gets `None`, which represents quiescence or some output $o$. $\delta$ or $o$ is added to $h$ and $\mathsf{acc}_b$ is asked for a counter example as described above for choice 2. If such a counter example is found, the test **fail**s, otherwise the test is inconclusive and `None` is returned. These three choices represent the choices in Algorithm $\mathsf{DT}_b$.

# 6 Example: Doped Nissan

The result presented in the paper can be verified with our algorithm by means of the two-step approach. Traces are stored in a format specified by the framework (see `examples/nissan/NEDC.txt` for an example). These files can be parsed by class `DynamometerTrace`. Moreover, this class offers two methods `get_system_under_test` and `get_test_case_selection` that together generate a virtual IUT and a test case selection, which, when used with `DT`, simulate the recorded experiment. File `examples/nissan/checkPowerNEDC.py` contains the a posteriori check of PowerNEDC and `examples/nissan/checkSineNEDC.py` of SineNEDC. Both files are structured equally and similar to file `examples/regions/randomTest.py` explained in Section 5. The recording of the experiments and the NEDC are loaded via `DynamometerTrace` from files `PowerNEDC.txt`, `SineNEDC.txt` and `NEDC.txt`. Then the contract is constructed the same way as for the Volkswagen example. However, the IUT, $\Omega_{\text{case}}$ and $\Omega_{\mathsf{In}}$ are generated by the `DynamometerTrace` instance holding the PowerNEDC (SineNEDC) record. As to be expected, this makes Algorithm $\mathsf{DT}_b$ return **pass** for PowerNEDC and **fail** for of SineNEDC.

# 7 How to run

The tool can be used by executing the Python files from folder `tools`. We executed these files using Python 2.7.16.

- `python examples/numbers/randomTest.py`
- `python examples/numbers/predefinedTest.py`
- `python examples/numbers/monitoring.py`
- `python examples/regions/randomTest.py`
- `python examples/nissan/checkPowerNEDC.py`
- `python examples/nissan/checkSineNEDC.py`

The tool will either say *Test passed!* and show the test trace or it says *Test FAILED for Standard Trace* and additionally gives a standard trace with an input distance of at most $\kappa_{\mathsf{i}}$ up to the length of the test trace. However, there is no standard trace with the same inputs as the standard trace shown, which has an output of at most $\kappa_{\mathsf{o}}$. Notice that the tool shows a warning when the test is trivially passed due to a test input deviating by more than $\kappa_{\mathsf{i}}$ from all standard traces.

## Acknowledgement

We gratefully acknowledge Thomas Heinze, Michael Fries, and Peter Birtel (Automotive Powertrain Institute of HTW Saar) for sharing their automotive engineering expertise with us, and for providing the automotive test infrastructure. This work is partly supported by the ERC Grant 695614 (POWVER), by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) grant 389792660 as part of TRR 248, see `https://perspicuous-computing.science`, by the Saarbrücken Graduate School of Computer Science, by the Sino-German CDZ project 1023 (CAP), by the Key-Area Research and Development Program Grant 2018B010107004 of Guangdong Province, by ANPCyT PICT-2017-3894 (RAFTSys), and by SeCyT-UNC 33620180100354CB (ARES).