LAAS-CNRS Internal Report

TAF : Testing Automation Framework v2.0 User Manual

Contributors (Alphabetical order):
Valentin Bouziat
Jérémie Guiochet
Mateo Mangado
Clément Robert
Luca V. Sartori
Hélène Waeselynck
LAAS-CNRS
Toulouse, France

This publication is available from: HAL LINK TBD / www.laas.fr/projects/taf
December 10, 2024

Table of Contents

1. Introduction

TAF is a tool for automatic generation of test cases, starting from an XML file describing the structure of the generated data, and the constraints over the structure. This file is called the *test_model* file in this document. The generated test cases are produced in XML, but an export facility is provided by TAF, specified by the user in Python language on how to translate the generated XML test cases into a specific format (e.g., a JSON, a bitmap, a CSV, etc.) for a specific system under test.

A complete description of algorithms is provided in [1], but other publications, example and source code are available at [2]

2. Installation

TAF is open source in python, it does not require a proper installation, but only a copy of the sources is required.

2.1 Install requirements

TAF has been tested on mac osx (>= 10.11), linux Ubuntu (>= 16.04), and Windows 10. As it is python-based project, we refer below the version of the packages required for running TAF v2.0 :

- Python 3.6 Packages : (https://www.python.org/downloads/)
- Numpy 1.18.4 : (https://numpy.org/install/)
- z3-solver 4.8.17 : (https://pypi.org/project/z3-solver/)

Please refer to official webpages of these packages to see specific os requirements.

2.2 Install TAF

As TAF v2.0 is Python based, only getting the sources from the git repository is required, using :

```
> git clone https://redmine.laas.fr/laas/taf.git
```

You can also navigate in the sources through the redmine repository :
https://redmine.laas.fr/projects/taf

2.3 Running the first example

To run the first example (default one is a generation of a gradient bitmap), you have to type the following command lines (later explained in section 4).

```
> cd yourpath/TAF/src      % go into the TAF folder (after git clone)
> python3 TAF.py           % launch TAF
> display all              % visualize the settings
> parse_template           % parse the template
> generate                 % generate test cases
> cat ../output/test_case_0/test_case_0.test_case % visualize the result
```

The generated test cases are provided in XML in the folder :
yourpath/taf/experiment/test_case_x

If you want to try TAF without the GUI, you can use the terminal typing python3 Taf.py followed by commands, e.g.:

```
> python3 Taf.py set template_file_name <newfile.template>

> python3 Taf.py silent parse_template generate
```

The generation will be executed based on the setting.xml configuration file.

3. Using TAF as a Python library

3.1 Add TAF to your system path

Solution 1 : Add the src folder of TAF to your system path in your python script. For example if you cloned the TAF git repository in home/Documents :

```
#Add TAF path to system in your python script
import sys
sys.path.append('/home/Documents/taf/src')
import TAF
```

Solution 2 : (recommended for Unix systems) Edit your .bashrc file and source it :

```
> cat ~/.bashrc
```

Add the path to the TAF src folder in the bashrc file, for example :

```
export PYTHONPATH="${PYTHONPATH}:/home/Documents/taf/src"
```

```
> source ~/.bashrc
```

3.2 Call TAF functions in a python script

After adding TAF to your system path, you will be able to import the python module TAF in your python scripts and calls TAF functions.

```
import TAF

#instanciate a Taf Client
myTaf = Taf.Cli()

# calls to TAF functions
myTaf.do_parse_template() #parse the test model
myTaf.do_generate() #generate the test cases
```

You must pay attention to your location in the tree structure because all the paths of the TAF files will depend on it.

You can change the name of the setting that you want to change. The names of the settings are inside the file “settings.xml”.

3.3 Running examples

In each folders from the examples folder you will find one or several templates for those examples. Every example folder is composed of 4 files:

- a python script *Generate.py* to call TAF as a library, parse the template and generate test cases
- a *settings* XML file to parameter TAF test cases generation,
- a *test model* XML file to express data structure and constraints on the test cases to be generated.
- a python script *Export.py* wich state the export rules to generate *test_artifacts*: particular files to be used as input in softwares, simulators etc..

To run the example just run the script *Generator.py* using python.

3.4 List of available examples

- **BitMap :** The aim of this example is to create a gradient image in the bitmapformat. The darkest pixel is at the bottom left corner and the lightest one at the upper right (see Figure 8). The content of the image is structured as a set of pixel rows, where each pixel contains a grayscale value. This is a good example for understanding TAF functionnalities like test definitions and export function.
- **ConstrainedType :** This example shows how to use the xml tag <type> in the test definition model and how to define constraints on instantiated objects and/or directly on its types.
- **OZ :** This example is from an industrial use case of TAF described in [1]. This use case treat the generation of a field of vegetables with a lot of constraint to ensure the generated fields are close to the ones in reality.
- **Planes :** This example generated test case for a Traffic alert and Collision Avoidance System (TCAS). Two planes trajectories are generated and constraints ensure that the trajectory and speed of both planes will make them meet at some point.
- **Square Root :** This example is the test generation of a number and its squared value. This is a good strating example to explore how constraint are defined on TAF object in the test model file.
- **TaxPayer :** This example describes the data for an income tax management application. In this model, a tax payer" is a physical person that can support child (that are also tax payers) and earns 1 or more income. This example enlight inheritance concepts in TAF and how to apply constraints on it.
- **Tree :** This example aims on the generation of tree structures of diverse sizes and heights. This example shows how to define recursive structures (tress or binary trees) in the TAF test model using both type abstraction and reference.
- **Triangles :** In this example, you will find test cases generation for rectangle and isoceles triangles. This example introduces types utilisation in TAF and constraint definition
- **UAV :** This example is a use case that aims to generate simulation parameters for a drone mission GAZEBO. A test cases is an instances of obstacles of different sizes to be disposed in a field that a flying drone must then avoid using autopilot.
  https://github.com/skhatiri/Aerialist
- **XMLDesignPatterns :** This example shows how XSD/XML design patterns can be used in the test model file. You can find more information and examples about XSD/XML design patterns in :
  https://www.oracle.com/technical-resources/articles/java/design-patterns.html

4. TAF Shell commands and Python functions

Shell Command	Python Function	Explanations
display all	do_display	: lists all the parameters.
display <param>	do_display(param)	: gives the value of the selected parameter .
overwrite	do_overwrite()	: sets overwrite to True.
print_test_case	do_print_test_case()	: prints the current test case.
set <parameter> <value>	do_set(parameter;value)	: sets the value of a parameter.
silent	do_silent()	: sets verbose to True.
help	help_help()	: display the list of TAF commands
help <command_name>	-	: display help for a specific command
parse_template	do_parse_template()	: parses test model.
shell	do_shell(command)	: runs a shell command.
generate	do_generate()	: generates the test cases.
exit	do_exit()	: same as quit.
quit	do_quit()	: quit Taf.

5. TAF Settings (in the settings.xml file)

Parameters	Explanations
template_path	: path of the template folder.
template_file_name	: actual template file (to change with ”set”).
experiment_path	: path to create the experiment folder.
experiment_folder_name	: name of the experiment folder.
nb_test_cases	: changes the number of cases.
test_case_folder_name	: changes the name of this folder situated in the experiment folder.
nb_test_artifacts	: there can be several artifacts, this parameter must be uploaded.
test_artifact_folder_name	: changes the name of the artifact folder.
parameter_max_nb_instances	: adapt this to the number of instances in your templates.
string_parameter_max_size	: size that the parameter name cannot exceed.
node_max_nb_instances	: reduce the multiplicity, by limiting the number of nodes.
max_backtracking	: limit the number of backtracking steps regarding taken to find a new solution while decreasing the depth, when it is not possible to find a solution at a certain depth.
max_diversity	: limit the number of times that the diversity is injected in the solution.
z3_timeout	: time of generation accepted before an error message.

6. Test model file definition

This section describes how to write a test model file (template) in the XML-TAF language. We will use the example of template from OZ example. This example comes from a case study for the generation of worlds for testing an agricultural robot in simulation as it is described in [1].

The test model written in XML-TAF will give specifics instructions to TAF. The template filename cannot contain a full stop punctuation mark "." before ".template". Furthermore, the ".template" extension is not mandatory (it could be ".xml") but is advised to identify template files.

```
<?xml version="1.0"?>

<root name="test_case">
    <node name="field" nb_instances="1">
        <parameter name="vegetable" type="string" values="cabbage;leek" weights="5;7"/>
        <node name="row" min="1" max="40">
            <parameter name="length" type="real" min="10.0" max="100.0"/>
            <constraint name="interval" types="forall"
                        expressions="row[i]\length INFEQ 1.1*row[i-1]\length;
                                     row[i]\length SUPEQ 0.9*row[i-1]\length"
                        quantifiers="i"
                        ranges="[1, row.nb_instances-1]"/>
            <constraint name="interval_2"
                        expressions="row[0]\length INFEQ 1.1*row[row.nb_instances - 1]\length;
                                     row[0]\length SUPEQ 0.9*row[row.nb_instances - 1]\length"/>
        </node>
    </node>

    <node name="mission" nb_instances="1">
        <parameter name="is_first_track_outer" type="boolean"/>
        <constraint name="first_track"
                    expressions="IMPLIES(..\field\row.nb_instances EQ 1, .\is_first_track_outer EQ True)"/>
    </node>
</root>
```

Fig. 1. Template file example for the autonomous weeder simulation

6.1 How to write data structures

A test case template involves five different types of XML tags:

- $< root >$,
- $< node >$,
- $< parameter >$,
- $< constraint >$.
- $< type >$

Every element must have a “name” xml attribute.

The root is unique and mandatory, but both the parameters and nodes have a “nb_instances” (number of instances) meta-attribute that allows for multiplicity. If multiplicity is not explicitly declared in the template, the number of instances is supposed to be 1.

The template in Figure 1 illustrates these structural concepts. The test case root (L3) is composed of a node “field” (L4) and a node “mission” (L19). The node “field” has one instance (L4), and contains a parameter “vegetable” (L5) that can take the values “leek” or “cabbage”. A field is composed of multiple rows. In the declaration of the “row” node (L6), the allowed number of instances is specified by its min and max values (1 and 40). Each row element contains a parameter “length” (L7), with min and max values as well (10 and 100). As a general rule, all numerical parameters (real or integer) must have an explicit definition range, and all string parameters must have a set of candidate values.

TAF attaches a generator to each parameter in the structure, it aims to produce diverse values from the parameter type. In TAF 3, types of sampling are proposed :

- Uniform sampling
- Weighted sampling
- Normal sampling

If nothing is specified in the template, uniform sampling over all possible values is used as the default. For instance, in Figure 1, L9, the parameter “length” will be determined in the range $[10, 100]$ with a uniform sampling. The user has also the possibility to select other default generators. This is done when the parameter is declared, by using dedicated attributes. The set of available generators depends on the data type. Boolean and string parameters can have weighted choices. For instance, in Figure 1, L5, the declaration of the vegetable parameter introduces a biased sampling of values, where the choice “leek” (of weight 7) is more likely than “cabbage” (of weight 5). For integer or real parameters, there are two alternatives to uniform sampling over their definition range. The user can assign weights to subranges of values, or request sampling according to a normal distribution with some mean and variance. The parser of the template will check that the requested generator is compatible with the parameter type.

<type> xml tag can be use to define reusable data-types in the test model. First define a type the same way as a node in the test model, then you can instantiate it whith the xml attribute ”type” when defining a node. For example :

```
<type name="my_type">
..
</type>

<node type="my_type">
..
</node>
```

6.2 How to write constraints

The XML-TAF language syntax lets the user specify a list of one or more expressions separated by a semi-colon (Figure 1, L8, L13, L21).

The expressions may involve operators :

- Logical (NOT, AND, OR, IMPLIES).
- Arithmetic (+, -, *, /).
- Relational (==, !=, <, <=, >, >= written as EQ, DIF, INF, INFEQ, SUP, SUPEQ).

Operators are use in the classical way, expect the relational which adopt the z3 style (prefix notation, see L22 of Figure 1, the operator is first, and the operands come after).

Operators	**Use**
Logical	: <expression> DIF <expression or value>
Arithmetic	: <expression> - <expression or value>
Relational	: AND(<expression> INFEQ <expression>, <expression> EQ <expression>)

The variables can be any parameter of the test case structure. They are referenced by an access path relative to the location where the constraint is declared:

- we use the windows file system notation with separators "\", to avoid ambiguity with the division symbol
- "." and ".." refer to the current node and the parent node.
- Paths can include indices to refer to the instances of nodes, for instance Figure 1, L9, $row[i]\backslash length$ refers to the length parameter of the ith row.

Our language also provides quantification over finite structures. For instance, the constraint named $interval$ (L8-12) has a universal quantification ($forall$) over all row instances. It has a single quantified variable (i in L11) taking the value of row indices (L12). The language also provides existential ($exist$) quantification (not used in our example). Note that it is possible to use nested quantifiers of universal and existential types.

Additional remarks

- the template weights have to be integers
- ";" in the expressions in the constraints is used to separate the constraints, to not rewrite the ranges and quantifiers (when they are the same). The result is as if there were two different constraints. Then, Z3 will solve the constraints separately.
- the quantifiers identifier (e.g., i,j,k) can be just one letter, it is not possible to have quantifiers like "my_iterator"

7. Troubleshooting

You could and you will encounter errors, so here is a list of some of them, with their solutions :

- **Error :** UnicodeEncodeError: 'ascii' codec can't encode characters in position 13-14

  Solution : Use (before python3 Taf.py) export PYTHONIOENCODING=utf-8

- **Error :** Not correct execution of "generate" command if another template and Export files are replaced while executing Taf.py

  Solution : If a new template will be used, close the application of Taf.py and proceed to place the template and Export files in the respective folders, then run in the terminal the Taf.py.

8. References

[1] Clément Robert, Jérémie Guiochet, Hélène Waeselynck, and Luca Vittorio Sartori. TAF: a tool for diverse and constrained test case generation. In *21st IEEE International Conference on Software Quality, Reliability and Security (QRS)*, Hanan Island, China, December 2021.

[2] TAF. Testing Automation Framework. https://www.laas.fr/projects/taf, 2022. [Online; accessed 7-July-2022].