LAAS-CNRS Internal Report

TAF 1.0 User Manual Testing Automation Framework V1.0

Clément Robert
Luca V. Sartori
Jérémie Guiochet
Hélène Waeselynck
Mateo Mangado
LAAS-CNRS
Toulouse, France

This publication is available from: HAL LINK TBD

Table of Contents

1. Introduction

TAF is a tool for automatic generation of test cases, starting from an XML file (actually an extension of XML) describing the structure of the generated data, and the constraints over the structure (e.g., number of instances). This file is called the *template* file in this document. The generated test cases are produced in XML, but an export facility is provided by TAF to specify in the Python language how to translate the generated XML test cases into a specific format (e.g., a JSON, a bitmap, a CSV, etc.) required to test the user's 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 dot require a proper installation, but only a copy/paste 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 v1.0 (minimum requirement, might work with more recent versions):

- Python 3.8 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 is python based, only getting the sources 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

3. Running the first example

To run the first example (default one is a *template* for generation of a gradient bitmap), you have to type the following command lines (later explained in section 5).

```
> 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 ../experiment/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

The corresponding bmp (export is here to translate the xml into a bmp image) is here :
yourpath/taf/experiment/test_case_x/test_artifact_0/bmp.bmp

You can open it with any bmp viewer.

4. The template and the export files

In *yourpath/taf/experiment/templates* you will find several template folders with examples. Every template folder is composed of 2 files: a **template** and an **export** file. In order to use them you have to proceed as described below.

4.1 Step 1. Select the template file

The template file written in XML-TAF will give specifics instructions to TAF. This file can be modified easily, we did a lexic of his syntax further in this manual, in Section 6.
The template filename cannot contain a full stop punctuation mark "." before ".template". Furthermore, the ".template" extension is not mandatory (it could be ".txt") but is advised to identify template files.

To select a template file for TAF:

- Copy the original file and then paste it in the 'templates' folder.
- Launch TAF (python3 TAF.py) and use the command *set template_file_name <xxx.template>* to replace the older template.

4.2 Step 2. Specify the export file

The "export.py" allows the user to specify in python how the generated XML test cases will be transformed (exported). If no export file is specified, then only the xml test cases will be produced.

Every export.py is specific to the selected template. Using an export that is incorrect will cause the generation process to fail, so the export.py file has to be deleted every time that you use a new or different template. To specify an export file (from the examples provided):

- Copy the export.py file from *taf/templates/xxxExamples* (the same folder in which you took your template file) and paste it into *taf/src*
- Restart TAF

The way the export file is programmed is pure python and not presented in this manual.

4.3 Step 3. Launch test case generation

As only these two files are required, the job is (almost) done :

- Delete the experiment folder if there was one
- in TAF, parse again if the template has been changed using: *parse_template*
- In TAF, launch generation (test cases are generated in both xml and export format) using: *generate*

After generation, the generated files are stored in the "experiment" folder. For each test case, a folder will contain an XML file and a file called an "artifact" (in another folder) created by the export.py file.

Comments on generation :

- The generation is aleatory with an optimisation for diversity of generated test cases. However, it is possible to specify a partially instantiated test case (in XML TAF), with fixed values for some parameters, and also using a fixed seed that will always generate the same test cases. For that create a partially instantiated test case in the experiment folder with just the seed line (and removing the other lines).
  You can also give a string as a seed, e.g. <seed value ="fUKryH5bUJ"> or <seed value ="pizza">.
- TAFv1 is not optimized to use all the CPU resources (not using multiprocessors).

5. TAF commands

Commands	Explanations
display all	: lists all the parameters.
display <param>	: gives the value of the selected parameter .
overwrite	: sets overwrite to True. Existing file will be overwritten during generation.
print_test_case	: prints the current test case.
set	: sets the value of the setting parameter given as argument (detailed just later).
silent	: sets verbose to True. Details will be printed during generation.
help	: prints some help. The same command can be executed using "?".
parse_template	: parses the template pointed by the setting parameters.
shell	: runs a shell command. The same command can be executed using "!".
generate	: generates the test cases.
exit	: same as quit.
quit	: quit Taf.

When launched, type "help" in the prompt to get a list of the available commands.

```
> help <command_name>
```

The *set* command is used to modify parameters like this :

```
> set <parameter> <new_value>
```

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. The template file : XML-TAF language

This section describes how to write a template file in the XML-TAF language. We will use the example of template from 'oz.template'. 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].

```
1  <?xml version="1.0"?>
2
3  <root name="test_case">
4      <node name="field" nb_instances="1">
5          <parameter name="vegetable" type="string" values="cabbage;leek" weights="5;7"/>
6          <node name="row" min="1" max="40">
7              <parameter name="length" type="real" min="10.0" max="100.0"/>
8              <constraint name="interval" types="forall"
9                          expressions="row[i]\length INFEQ 1.1*row[i-1]\length;
10                                      row[i]\length SUPEQ 0.9*row[i-1]\length"
11                         quantifiers="i"
12                         ranges="[1, row.nb_instances-1]"/>
13             <constraint name="interval_2"
14                         expressions="row[0]\length INFEQ 1.1*row[row.nb_instances - 1]\length;
15                                      row[0]\length SUPEQ 0.9*row[row.nb_instances - 1]\length"/>
16         </node>
17     </node>
18
19     <node name="mission" nb_instances="1">
20         <parameter name="is_first_track_outer" type="boolean"/>
21         <constraint name="first_track"
22                     expressions="IMPLIES(..\field\row.nb_instances EQ 1, .\is_first_track_outer EQ True)"/>
23     </node>
24 </root>
```

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

6.1 How to write data structures

A test case template involves four different types of XML elements: *Root*, *Node*, *Parameter*, and *Constraint*. Every element must have a "name" attribute and are nested to form a tree starting from the root. The root and nodes are composite data structures with *child* elements, while a parameter is not composite. The types of parameters are boolean, string, real, and integer.

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

Examples of how to use the distributions are inside the folder */templates/otherExamples/* in the TAF repository.

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 (provide an example here). The parser of the template will check that the requested generator is compatible with the parameter type.

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. An example is given in the tax payer template on the git repository in the "templates" directory.

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.
- it's impossible to use upper case letters for parameters and nodes
- NOT(var EQ 1)"" is like "var DIF 1"
- the quantifiers identifier (e.g., i,j,k) can be just one letter, it is not possible to have quantifiers like "my_iterator"

7. Using TAF with command lines and as a library

7.1 TAF with command line in the Shell

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 silent parse_template generate
```

```
> python3 Taf.py set template_file_name <newfile.template>
                                        silent parse_template generate
```

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

7.2 Taf with command line from a Python source file

It is also possible to generate it with a python file, as this ”CommandLineTaf.py” example, which allows to have the deletion of the experiment folder as well as the command lines usually checked in the terminal one by one directly coded in it :

```
import shutil
import os
from os.path import basename
from pathlib import Path

#set the new template
for txt_path in Path("../templates").glob("*.template"):
    templ=basename(txt_path)
os.system("python3 Taf.py set template_file_name {}".format(templ))

#check of the export and the template files from the user
test=str(input('Does the export file is in the "src" folder and does the template \
file associated is in the "templates" folder ? \nYes/No (y/n):'))

if test.upper()=='Y':
    #delete the older experiment folder
    if os.path.exists('../experiment'):
        shutil.rmtree('../experiment')

#use TAF without the GUI, directly from this python file
    os.system("python3 Taf.py silent overwrite parse_template generate")
    os.system('xdg-open "../experiment"') #open the experiment folder
else :
    print('Please make sure all is good before start TAF.')
```

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”.

7.3 Using TAF as a python library

It is possible to use TAF as a python library following these steps :

1. Create a python script as in the figure bellow,
2. create a file settings.xml in the same repository of your script with your custom settings,
3. create a template file and add its path to settings.xml.

```
# add TAF path to system
import sys
sys.path.append('path_to_taf_repository/taf/src')
import Taf

# instanciate a TAF Client
myTaf = Taf.CLI()

# calls to TAF functions
myTaf.do_parse_template('')
myTaf.do_generate('')
```

8. 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.

9. 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].

Appendices

A. XML-TAF grammar - BNF style

```
<root> ::= "<root " <name> ">" <node> <parameter> <constraint> "</root>"
<node> ::= "<node " <name> ">" <node> <parameter> <constraint> "</node>"
         | <node> <node>
<parameter> ::= "<parameter " <name> <p_arg> <nb_instances> "/>"
     | <parameter> <parameter>
     | ε
<name> ::= "name="" <letter> <string> "" "
<nb_instances> ::= "nb_instances="" <integer> "" "
         | ε
<p_arg> ::= "type="boolean" "
     | "type="boolean" " <boolean_values>
     | "type="boolean" " <boolean_values> <boolean_weights>
     | "type="string" " <string_values>
     | "type="string" " <string_values_and_weights>
     | "type="integer" " <integer_min> <integer_max> <numerical_optional_attributes>
     | "type="real" " <real_min> <real_max>
<numerical_optional_attributes>
<boolean_values> ::= "values="[True;False]" "
     | "values="[False;True]" "
<boolean_weights> ::= "weights="[" <integer> ";" <integer> "]" "
<string_values> :== "values="[" <val> "]" "
<val> ::= <string>
         | <string> ";" <string>
<string_values_and_weights> ::= "values="[" <string> <vnw> <integer> "]" "
<vnw> ::= ";" <string> <vnw> <integer> ";"
     | "]" weights="["
<integer_min> ::= "min="" <integer> "" "
<integer_max> ::= "max="" <integer> "" "
<real_min> ::= "min="" <float> "" "
     | "min="" <integer> "" "
<real_max> ::= "max="" <float> "" "
     | "max="" <integer> "" "
<numerical_optional_attributes> ::= ε
     | "distribution="u" "
     | "distribution="n" "
     | "distribution="n" " <mean>
     | "distribution="n" " <variance>
     | "distribution="n" " <mean> <variance>
     | "distribution="i" " <interval>
     | "distribution="i" " <interval_and_weights>
<mean> ::= "mean="" <float> "" "
<variance> ::= "variance="" <float> "" "
<interval> ::= "ranges="" <inter_list> "" "
<inter_list> ::= <inter> ";" <inter>
```

```
    | <inter>
<inter> ::= "[" <float> "," <float> "] "
<interval_and_weights> :== "ranges="" <inter> <inw> <integer> "]" "
<inw> ::= ";" <inter> <vnw> <integer> ";"
    | "" weights="["
<constraint> ::= "<constraint " <name> <types_quantifiers_and_ranges> <expressions> "/>"
    | <constraint> <constraint>
    | ε
<types_quantifiers_and_ranges> ::= ε
    | "types="" (<type>";")n <type> "" quantifiers="" (<letter>";")n
    <letter> "" ranges=""(<range>";")n <range> """ <expressions>
<type> ::= "forall"
    | "exist"
<range> ::= "[" <expr> "," <expr>"]"
<expressions> ::= " expressions="" <expression> """
<expression> ::= <expr>
    | <expression> ";" <expression>
<expr> ::= "(" <expr> ")"
    | "AND(" <expr> "," <expr_list> ")"
    | "OR(" <expr> "," <expr_list> ")"
    | "NOT(" <expr> ")"
    | "IMPLIES(" <expr> ","<expr> ")"
    | <term> <comparison_operator> <term>
<expr_list> ::= <expr>
    | <expr> "," <expr_list>
<comparison_operator> ::= "SUP"
    | "SUPEQ"
    | "INF"
    | "INFEQ"
    | "EQ"
    | "DIF"
<term> ::= <term> "+" <term>
    | <term> "-" <term>
    | <term> "*" <term>
    | <term> "/" <term>
    | <term> "%" <term>
    | <tree_path>
    | <integer>
    | <float>
<tree_path> ::= <tree_path_beginning> <tree_path_element> <tree_path_ending>
<tree_path_beginning> ::= ε
    | "./"
    | "../"
    | <tree_path_beginning> <tree_path_beginning>
<tree_path_element> ::= <name> "[" <term> "]"
    | <name>mateo.mangado@gmail.com
    | <tree_path_element> "/" <tree_path_element>
<tree_path_ending> ::= ε
    | ".nb_instances"
<float> ::= <integer> "." <integer>
<integer> ::= <non_nul_digit><digit_string>
```

```
<digit_string> ::= <digit_string><digit_string>
    | <digit>
<string> ::= <string><string>
    | <letter>
    | <digit>
    | "_"
<non_nul_digit> ::= "1" | "2" | "3" "9"
<digit> ::= "0"
    | <non_nul_digit>
<letter> ::= "a" | "A" | "b" | "B" | "c" | "C" "y" | "Y" | "z" | "Z"
```

B. XML-TAF DTD

```
<!ELEMENT root (parameter*, node*, constraint*)>
<!ELEMENT node (parameter*, node*, constraint*)>
<!ELEMENT parameter EMPTY>
<!ELEMENT constraint EMPTY>

<!ATTLIST root name CDATA #REQUIRED>
<!ATTLIST node name CDATA #REQUIRED
             min CDATA #IMPLIED
             max CDATA #IMPLIED
             nb_instances CDATA #IMPLIED>
             distribution (u | n | i) #IMPLIED
             mean CDATA #IMPLIED
             variance CDATA #IMPLIED>
             ranges CDATA #IMPLIED
             weights CDATA #IMPLIED
<!ATTLIST constraint name CDATA #REQUIRED
               expressions CDATA #REQUIRED
               types (forall | exist | unique) #REQUIRED
               quantifiers CDATA #IMPLIED
               ranges CDATA #IMPLIED>
<!ATTLIST parameter name CDATA #REQUIRED
              types (boolean | string | integer | real) #IMPLIED
              values CDATA #IMPLIED
              ranges CDATA #IMPLIED
              weights CDATA #IMPLIED
              min CDATA #IMPLIED
              max CDATA #IMPLIED
              distribution (u | n | i) #IMPLIED
              mean CDATA #IMPLIED
              variance CDATA #IMPLIED>
```