The Versus framework: a file comparison standard for software testing

Omar Mohamed Ragi
Ahmed Sayed Ali
Reem Al-Adawi

omar_ragi@mentor.com
ahmed_ali@mentor.com
reem_eladawi@mentor.com

Abstract

In any software automated testing environment, file comparison is an essential step to determine if the outputs of the system under test match the expected results. Improving this testing step significantly enhances the testing flow and makes the whole process robust and reliable. For this reason, we created Versus, a file comparison framework and standard designed to facilitate the files comparison task by offering QA engineers a state-of-the-art internal tool to perform the comparison using the optimal environment and settings, eliminating the need to write and maintain complex scripts.

The main concepts of the Versus framework are maintainability, and ease of use. The framework is designed to only accept one input, a human-readable, easy to create and maintain, YAML-formatted configuration file that contains definitions of all the comparison checks. Versus parses this YAML file, executes the comparison checks, and generates an easy-to-analyze comparison summary report. This report, designed with the goal of reducing test failure analysis time, directs engineers to the fastest method for reviewing detected differences.

By standardizing file comparison, one of the most important steps in any automated testing flow, the Versus framework is a great tool for the automated testing environment. It makes fast, accurate file comparison accessible to all QA engineers, regardless of experience, saving time while ensuring best practices are used for file comparison in every automated test scenario.

This paper presents the concepts of the Versus framework, and how these concepts can and should be adopted in any QA testing automation process aiming for high efficiency and productivity.

Biography

Omar Mohammed Ragi is a SW QA manager for Siemens Digital Industries Software. He has a B.Sc. degree in Computer Science from Arab Academy for Science, Technology & Maritime Transport, Cairo, Egypt.

Ahmed Sayed Ali is a Senior QA Engineer for Siemens Digital Industries Software. He has a B.Sc. degree in Electronics and Communications from Ain Shams University, Cairo, Egypt.

Reem El-adawi is Director of Quality for Siemens Digital Industries Software. She holds a B.Sc., M.Sc., and Ph.D. from Ain Shams University, Electronics and Communication department, Egypt.

1 Introduction

Any testing automation system has two main steps:

- running the tool under test after providing the correct inputs that define the scenario to be tested
- comparing the outputs of the tool under test after the run is complete to determine if they match the expected results (the baseline or “golden” results).

The Versus framework simplifies and standardizes the comparison step, making the file comparison easy, and accessible for any QA, regardless of their experience level, while offering a robust process based on best practices. The Versus framework is a valuable component of ensuring the best tools, practices, and optimization are maintained in testing flows. In this paper, we present the main components of the Versus standard, and explain how they are synergized to achieve the described goals.

2 Principles of the Versus standard

Versus is both a framework and a file comparison standard that can be used to compare files efficiently and easily, making it the perfect tool for testing automation comparison step. There are five principles in in the Versus comparison standard—these principles are the pillars on which Versus is created and designed (Fig. 1). These principles serve and underlie the ultimate goal of making fast, accurate file comparison accessible to all QA engineers without the need for prior experience or scripting skills.

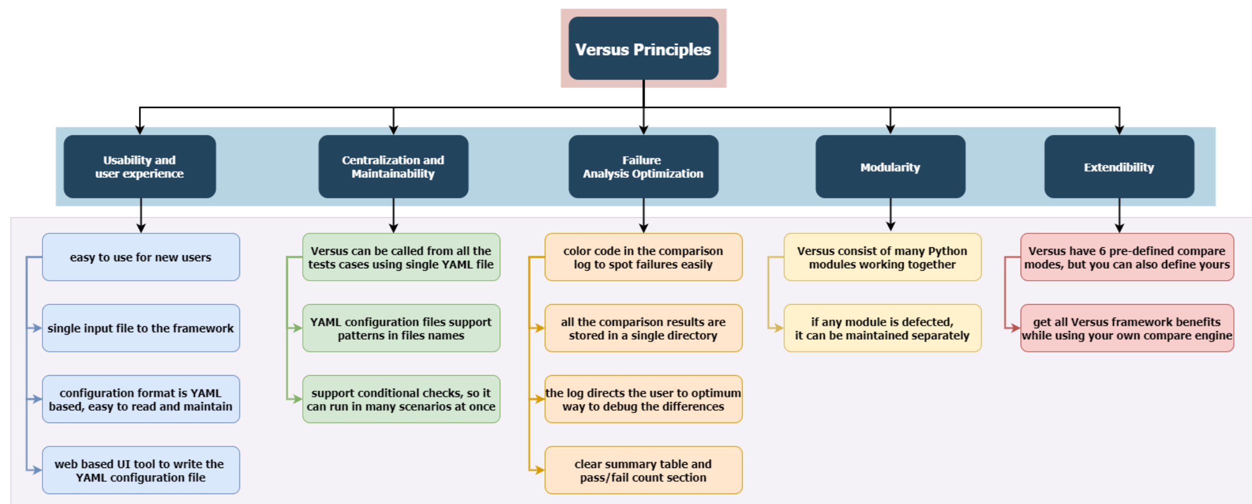

Fig. 1. The five principles of the Versus framework.

In the following sections, we define these principles and show how they are incorporated into the design of Versus.

2.1 Usability and user experience

The first principle of the Versus framework is usability. To enhance ease of use for new or inexperienced QA engineers, we focused on making the getting-started experience with the Versus framework easy and fast. By default, the Versus framework only accepts one input file, which reduces potential sources of complexity in the file comparison process.

The Versus input is a YAML format file, which is not only human-readable, but can also be parsed by many modules and libraries in almost any scripting language. Entry names and reserved words used in this YAML format input file are chosen carefully to be intuitive and easy to remember, to limit the need to consult the documentation whenever users need to write a new Versus configuration file.

To make the Versus framework easier to configure, we built a web-based UI application that enables users to choose the type of comparison, inputs, and settings for each check. The application then automatically generates the YAML configuration file, ready to be used with the Versus framework (additional details are provided in section 8).

This web-based UI application is also deliberately designed to minimize any need to consult the documentation of the Versus framework. It starts by asking users the important questions first, then guiding them through multiple questions until it has all the information needed to create the YAML configuration file. This approach eliminates the need for users to have prior knowledge about the syntax of the input configuration file, or the available entries it contains (additional details provided in section 7).

2.2 Centralization and maintainability

Centralization is a very powerful concept that is essential for a successful and maintainable testing automation process. Centralizing key components of the testing automation scripts and tools reduces the time and cost needed to develop them, allowing many team members to use a single source of code or tool that would otherwise have been re-invented many times in potentially different levels of quality. Centralization also supports and simplifies system maintenance—if any part of the testing tools needs modification or fixing, it can be done on a single location, instantly providing the resulting corrections or enhancements to all the test cases (scenarios) in the testing automation regression suite using the centralized testing tool or script.

The Versus framework is designed with centralization in mind to maximize its maintainability. If many test cases use the same comparison check, they can all use the same YAML configuration file, so changing anything in this single YAML configuration file instantly updates all of the testing suites using that check.

The Versus configuration file also supports defining file names with patterns, and even adds arrays of file names in both the entries of the output file and the baseline data. These features mean that even if the test cases in an automated testing suite are comparing different types of files, or files with different names, all the test cases can share the same compare checks, and whatever output in each test matches the file name pattern you entered will be compared just as if the full name of the file was written.

The Versus configuration file also supports conditional compare checks (i.e., those that depend on certain conditions to trigger the comparison). If the conditions are not present, the comparison will be ignored, or be performed with relaxed tolerances. This feature makes it easier to support a centralized configuration file.

2.3 Failure analysis optimization

Another goal of the Versus framework is to minimize the time required to analyze the failure of an automated test case, which includes spotting the source of the failure in the shortest time possible. The Versus framework achieves this goal with an optimized output log containing multiple shortcuts and aids to save analysis time.

If Versus is run from a terminal, important data in the log is color-coded so users can quickly distinguish passing compare checks from failing checks without the need to read every line in the log (refer to section 6).

Versus generates a "transcripts" directory while executing the comparison checks that contains all the comparison results, as well as the intermediate files used in the comparison execution. This directory is well organized with a specific hierarchy based on the compare checks parameters (refer to section 7).

Log contents are minimized. At the end of a log, Versus prints two summary reports: a table that contains all the compare checks and their final status, and a final count of the checks, including how many passed, and how many failed. This gives users a quick glimpse of the general status of the automated test case before they begin analyzing the failures and going deep into the comparison failure log (refer to section 6). However, the log doesn't contain the results of the comparison in the summary report, but provides a

reference to its location in the transcripts directory. (refer to section 6) The log also provides the optimal way to debug a comparison check failure—e.g., if the check type is an image comparison, the log provides a link to an image that contains a merged version of the two compared images, with the differences highlighted (refer to section 6).

Following these guidelines cuts down the automated test failure analysis time by a considerable amount and saves a lot of time for the QA engineers in one of the most important roles of their jobs, which is maintaining their automated testing suite.

2.4 Modularity

Versus code is built in a modular style, meaning each comparison mode, each extractor, and even the part that generates the final log and the summary report, are all built into separate python code sources. Modularity ensures that if any defect is introduced into one module, the other modules and pieces of code are not affected, while also making it easier to find and fix errors.

Modularity also supports some “plug-ins” that can be used to extract and filter the output files (or metric values) to be compared to the baseline version, so users can create their own python code to manipulate the output data, making it even more modular.

2.5 Extendibility

Versus allows users to include a custom comparison mode that uses a different comparison engine, rather than the engines supported by Versus by default (those described in section 5). This feature lets users use their preferred engine within the Versus framework environment, providing them all the benefits of Versus while allowing them the flexibility to use their own comparator (refer to section 5.7).

3 Versus framework components

In this section, we describe the main framework components that define the experience of using Versus in testing automation (Fig 2).

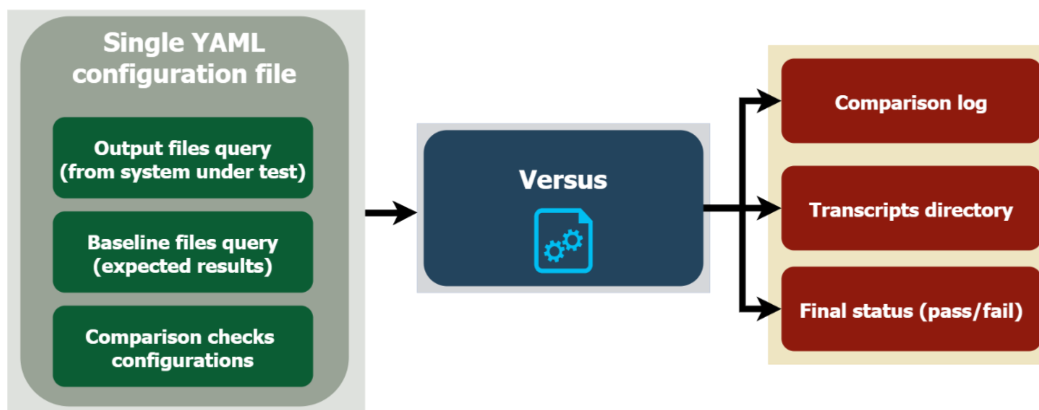

Fig. 2. Versus framework components.

3.1 Comparison script inclusion

Each test in the automated testing suite is expected to call Versus framework from its main location (source), using a central configuration file that contains all the needed comparison checks. This recommended approach achieves the Versus principle of centralization.

Fig. 3 shows a typical Versus call inside the main compare script of each automated test case.

```
1    #!/bin/sh
2
3    #calling Versus.
4    /user/pevtools/bin/versus --yaml /central/location/versus.yaml
5    status=$?
6
7    #exiting with proper status.
8    exit $status
```

Fig. 3. The recommended method for calling the Versus framework.

3.2 Configuration file

The configuration file is the only input Versus needs. It contains all the data needed to extract the files and metrics to be compared, and all the comparison checks to be executed on the outputs against the expected results (baseline version).

The configuration file must be written in YAML format (while Versus accepts JSON, it is not recommended as it is not as human readable). The typical Versus configuration file contains three sections in total (Fig. 3):

- **scripts**: this section is dedicated to any script or shell command users want to run inside the automated test before running the extraction and comparison checks It is typically used to run some code to filter unwanted data from the output files of the tool under test, or to generate an altered version of these output files that are more suitable for comparison.
- **extract_metrics**: this section is dedicated to defining the extractors, which are a set of python functions to be run with the parameters passed to them by the users. Its goal is to extract the needed values (metrics) from the output files to be compared to the baseline version.
- **compare_metrics**: this section is dedicated to defining the comparison checks that Versus will execute on the output files, the extracted metrics, or the filtered/altered versions of the output files.

```
1    scripts:
2        - /path/to/script1.sh
3        - /path/to/script2.sh
4        - ...
5    extract_metrics:
6        - metric1:
7            <parameters>
8        - metric2:
9            <parameters>
10       - ...
11   compare_metrics:
12       - check1:
13           <parameters>
14       - check2:
15           <parameters>
16       - check3:
17           <parameters>
18       - ...
```

Fig. 4. Versus configuration file sections.

More details on the configuration file sections are provided in section 4.

3.3 Log and summary report

Versus is always verbose about what it is doing at any moment. The log it generates shows the current step and the results of this step, as shown in Fig. 5. After the comparison is done, Versus generates a summary report table that summarizes the results of all of the compare checks and gives users final counts of passing and failed checks. More details about the log and summary report are provided in section 6.

```
Reading versus.yaml

Check: CSV_COMP_normal
  |  /user/peteoss/aoi/bin/numdiff -z @ -S --separators=' \t\n\r, ;:|"'  ./my_out1.csv  ./baseline/my_out1.csv
  |  FAILED, check this file: ./transcripts/numdiff_transcripts/CSV_COMP_normal.transcript

Check: CSV_COMP_fast
  |  /user/peteoss/aoi/bin/numdiff -z @ -S --separators=' \t\n\r, ;:|"'  ./my_out2.csv  ./baseline/my_out2.csv
  |  FAILED, check this file: ./transcripts/numdiff_transcripts/CSV_COMP_fast.transcript

Compare Report:
+-----------------+---------------------------+--------------+------------+--------+-----------------------------------------------------------+
| check           | baseline value [baseline] | metric value | check type | Status | comments                                                  |
+-----------------+---------------------------+--------------+------------+--------+-----------------------------------------------------------+
| CSV_COMP_normal | ./baseline/my_out1.csv    | my_out1.csv  | NUMDIFF    | FAIL   | ./transcripts/numdiff_transcripts/CSV_COMP_normal.transcript |
+-----------------+---------------------------+--------------+------------+--------+-----------------------------------------------------------+
| CSV_COMP_fast   | ./baseline/my_out2.csv    | my_out2.csv  | NUMDIFF    | FAIL   | ./transcripts/numdiff_transcripts/CSV_COMP_fast.transcript   |
+-----------------+---------------------------+--------------+------------+--------+-----------------------------------------------------------+


Summary Report:
==================
Total:2
PASS :0
FAIL :2
==================
```

Fig. 5. Versus log and summary.

3.4 Transcripts directory

The transcripts directory is where Versus keeps all intermediate files it generated, any filtered files, and the results of the comparisons it made. It is organized in a specific hierarchy to be intuitive and self-explanatory. If users know the check names they chose in the configuration file, they can easily navigate the transcripts directory to get the files they want for failure analysis (Fig. 6). More details about the structure of the "transcripts" directory are provided in section 7.

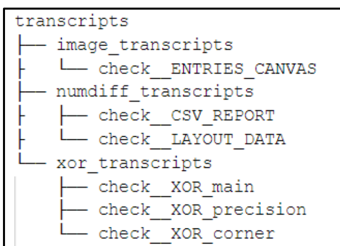

Fig. 6. Transcripts directory hierarchy.

4 Versus configuration file

4.1 Introduction

The configuration file is the Versus framework's entry point, where users describe the comparison checks for all automated tests. It is designed to meet the following criteria:

- Easy to read, understand and maintain
- Supports generic comparison checks, so that one configuration file can be used by the whole regression suite
- Modular and re-usable (check descriptions can be re-used in other configuration files)

In the following sections, the main sections of Versus configuration file will explained in details.

4.2 Versus YAML configuration file sections

The Versus configuration file contains three main sections:

- Scripts
- Metrics extraction
- Metrics comparison

4.2.1 Scripts

The *scripts* section is the first to be executed by Versus. Users can use this section to define any script/command that must be run before the comparison checks (Fig. 7). These commands/scripts are typically used to prepare the output files for comparison (filtration, aggregation, … etc.).

```
1  scripts:
2      - /path/to/filter.py output.txt > output.filt
3      - grep 'error' transcript*log* > errors.log
4      - ls core.* > core_files.list
5      - ...
6      - ...
```

Fig. 7. *scripts* section of configuration file.

4.2.2 Metrics extraction

The *extract_metrics* section is used to call pre-defined metric extractor functions for extracting and storing the metrics to be compared in a standard format that can be easily compared (Fig. 8).

```
1  extract_metrics:
2      - <extraction_step_1>:
3          functions: [extractor_func_1]
4          params: { name1: value1 }
5      - <extraction_step_2>:
6          functions: [extractor_func_1,extractor_func_2,...]
7          params: { name1: value1, name2: value2 }
8      - ...
```

Fig. 8. *extract_metrics* section syntax.

4.2.3 Comparison metrics

The *compare_metrics* section is used to define the comparison checks that use one of the pre-defined comparison modes (Fig. 9).

```
1   compare_metrics:
2       - <check_name_1>:
3           mode: <check_type>
4           metric: <output file(s)/folder(s)>
5           baseline: <baseline file(s)>
6           <check_parameters>
7       - <check_name_2>:
8           mode: <check_type>
9           metric: <output file(s)/folder(s)>
10          baseline: <baseline file(s)>
11          <check_parameters>
12      - <check_name_3>:
13          mode: <check_type>
14          metric: <output file(s)/folder(s)>
15          baseline: <baseline file(s)>
16          <check_parameters>
17      ...
```

Fig. 9. *compare_metrics* section syntax.

5 Versus comparison modes

Versus has six pre-defined comparison modes that cover most of the common output file types, as well as a seventh custom mode that can be used to compare any special file type by incorporating another comparison engine within the Versus framework, gaining all of the benefits of the framework without re-inventing the wheel.

5.1 diff mode

The *diff* mode is the simplest comparison mode in Versus—it compares two files line by line using the standard Linux diff command [1]. The comparison check passes only if the output file and baseline are identical (no tolerance support in this mode). Fig. 10 shows some examples of the *diff* mode usage.

```
1   compare_metrics:
2       #simplest scenario
3       - check_one:
4           mode: diff
5           metric: ./file.txt
6           baseline: ./baseline/file.txt
7       #comparing array of files.
8       - check_two:
9           mode: diff
10          metric: [./outputs/file1.txt,./outputs/file2.txt,]
11          baseline: [./baseline/file1.txt,./baseline/file2.txt,]
12      #having wildcards in the baseline (multiple files comparison per check).
13      - check_three:
14          mode: diff
15          metric: ./outputs
16          baseline: ./baseline/*.{txt,csv}
```

Fig. 10. *diff* mode syntax and examples.

5.2 scalar mode

The *scalar* mode is used to compare two singular numerical values, usually stored in files that are generated by the *extract_metrics* functions. This mode supports numerical tolerance, and ignores positive/negative differences. The comparison check passes only if the two metrics match, or if the error is within the defined tolerance. Fig. 11 shows the syntax of the *scalar* mode and its available parameters.

```
1   compare_metrics:
2       - check_name:
3           mode: scalar
4           metric: <output file(s)/folder(s)>
5           baseline: <baseline file(s)> | <number>
6           a: <absolute_tolerance>
7           r: <relative_tolerance>
8           ignore_negative: <'0'|'1'>
9           ignore_positive: <'0'|'1'>
```

Fig. 11, *scalar* mode syntax.

Fig. 12 shows examples of the *scalar* mode.

```
1   compare_metrics:
2       #simplest scenario
3       - check_one:
4           mode: scalar
5           metric: ./runtime.txt
6           baseline: ./baseline/runtime.txt
7       #having absolute tolerance.
8       - check_two:
9           mode: scalar
10          metric: ./memory.txt
11          baseline: ./baseline/memory.txt
12          a: '1300'
13      #having relative tolerance.
14      - check_three:
15          mode: scalar
16          metric: ./result_count.txt
17          baseline: '15334'    #hardcoded baseline
18          r: '0.04'            #this is 4% tolerance
19      #having both (higher tolerance wins)
20      - check_four:
21          mode: scalar
22          metric: ./LVHEAP.txt
23          baseline: ./baseline/LVHEAP.txt
24          a: '2235'
25          r: '0.10'
26      #ignore_negative
27      - check_five:
28          mode: scalar
29          metric: [ ./output , . ]      #metric is array of directories, including the current directory (.)
30          baseline: ./baseline/*.txt    #baseline is a pattern
31          a: '32'
32          r: '0.10'
33          ignore_negative: '1'
34      #ignore_positive
35      - check_five:
36          mode: scalar
37          metric: ./output      #metric is array of directories, including the current directory (.)
38          baseline: ./baseline/*.txt    #baseline is a pattern
39          r: '0.03'
40          ignore_positive: '1'
```

Fig. 12. *scalar* mode examples.

5.3 operator mode

The *operator* mode is also used to compare two singular numerical values, usually stored in files that are generated by the extract_metrics functions. This mode supports all python comparison operators (==, !=, <=, >=, <,>). The comparison check passes only if the python operator check is true. Fig. 13 shows the syntax of the *operator* mode and its available parameters.

```
1   compare_metrics:
2       - check_name:
3           mode: operator
4           metric: <output file(s)/folder(s)>
5           baseline: <baseline file(s)> | <number>
6           check: <one of the standard python numerical checks>
```

Figure 13: *operator* mode syntax.

```
1   compare_metrics:
2       #first scenario
3       - check_one:
4           mode: operator
5           metric: ./memory.txt
6           baseline: ./baseline/memory.txt
7           check: '<='                   #always making sure current memory is less than or equal to the baseline value.
8       #second scenario
9       - check_two:
10          mode: operator
11          metric: ./results_count.txt
12          baseline: '0'                 #baseline is hardcoded to '0' here.
13          check: '=='                   #making sure that the results count is always equal to Zero.
```

Fig. 14. *operator* mode examples.

5.4 numdiff mode

The *numdiff* mode is used to compare text files that contain numbers, using the numdiff tool [3]. It supports the comparison of these files with tolerance for the numbers within these files. It is usually used for the CSV file comparison, as it has many options for facilitating the CSV file comparison. Fig. 15 shows the parameters and the syntax of the *numdiff* mode.

```
1  compare_metrics:
2      - check_name:
3          mode: numdiff
4          metric: <output file(s)/folder(s)>
5          baseline: <baseline file(s)>
6          a: <absolute_tolerance>
7          r: <relative_tolerance>
8          separators: <separators_string>
9          columns: [<array_of_column_header_names>]
10         csv_separator: <csv_separators_string>
11         tolerance_if:
12             VARs:
13                 - [<array_of_"key=value"_variables>]
14                 - a: <absolute_tolerance>
15                 - r: <relative_tolerance>
16             VCOs:
17                 - [<array_of_VCOs>]
18                 - a: <absolute_tolerance>
19                 - r: <relative_tolerance>
```

Fig. 15. *numdiff* mode syntax.

Fig. 16 shows examples of the *numdiff* mode.

```
1  compare_metrics:
2      #having absolute tolerance.
3      - check_two:
4          mode: numdiff
5          metric: ./f2.csv
6          baseline: ./baseline/f2.csv
7          a: '13'
8      #having relative tolerance.
9      - check_three:
10         mode: numdiff
11         metric: ./f3.csv
12         baseline: ./baseline/f3.csv
13         r: '0.15' # this is 15% tolerance
14     #having both (higher tolerance wins)
15     - check_four:
16         mode: numdiff
17         metric: ./f4.csv
18         baseline: ./baseline/f4.csv
19         a: '22'
20         r: '0.10'
21     #second complex example (useful if you want higher tolerance if certain environment variable is set.)
22     - check_five:
23         mode: numdiff
24         metric: [ ./output , . ]
25         baseline: ./baseline/*.csv
26         a: '32'
27         r: '0.10'
28         separators: ' ;,\n'
29         tolerance_if:
30             VARs:
31                 - [USE_HIGH_TOL=yes,RELAX_TOL=1,MT_FLEX_RUN=1]
32                 - a: '40'
33                 - r: '0.15'
```

Fig. 16. *numdiff* mode examples.

5.5 image mode

The *image* mode is used to compare images. It supports comparison with tolerance, so if the images are close to each other within the defined tolerance, the comparison passes. The Frog-Logic engine is used in this mode [2]. Fig. 17 shows the parameters syntax of the *image* mode.

```
1  compare_metrics:
2      - check_name:
3          mode: image
4          metric: <output file(s)/folder(s)>
5          baseline: <baseline file(s)>
6          correlation-or-tolerance: <number>  #you use either "correlation" or "tolerance", not both.
```

Fig. 17. *image* mode syntax.

Fig. 18 shows some examples of the *image* mode.

```
1  compare_metrics:
2      #image with no tolerance
3      - check_one:
4          metric: ./test.png
5          baseline: baseline/test.png
6          mode: image
7      #image with correlation (allowed similarity range)
8      - check_two:
9          metric: ./out1.png
10         baseline: baseline/out1_diff_small.png
11         mode: image
12         correlation: '99'
13     #image with tolerance (allowed difference range)
14     - check_three:
15         metric: ./out1.png
16         baseline: baseline/out1_diff_big.png
17         mode: image
18         tolerance: '90'
```

Fig. 18. *image* mode examples.

5.6 xor mode

The *xor* mode is used to compare two physical layout files of an integrated circuit (IC) design mask that are stored in different file formats [4]. It is common practice in Siemens to compare these types of files, as one of our primary software solutions is focused on IC design intellectual property (IP). This mode supports geometrical tolerance and conditional tolerance based on variables or current CPU architecture and operating system that might give slightly different output.

Fig. 19 shows the main syntax of the *xor* mode.

```
1  compare_metrics:
2      - check_name:
3          mode: xor
4          metric: <output file(s)/folder(s)>
5          baseline: <baseline file(s)>
6          layers: [<array_of_layers>]
7          bins: [<array_of_bins>]
8          exclude_bins: [<array_of_bins>]
9          exclude_bins_if:
10             VARs:
11                 - [<array_of_"key=value"_varaibles>]
12                 - bins: [<array_of_bins>]
13             VCOs:
14                 - [<array_of_VCOs>]
15                 - bins: [<array_of_bins>]
```

Fig. 19. *xor* mode syntax.

Copies may not be made or distributed for commercial use

Fig. 20 shows some examples of the *xor* mode.

```
1   compare_metrics:
2       #xoring certain layers and bins.
3       - check_two:
4           mode: xor
5           metric: ./output/layout2.oas
6           baseline: ./baseline/layout2.oas
7           layers: [1,2,9]
8           bins: [1dbu,2dbu]
9       #excluding bins.
10      - check_three:
11          mode: xor
12          metric: ./output/layout3.oas
13          baseline: ./baseline/layout3.oas
14          layers: [1,2,9]
15          bins: [1,2,5,10]
16          exclude_bins: [1,2]
17      #excluding bins if  (more bins will be excluded if certain environment variable is set)
18      - check_five:
19          mode: xor
20          metric: ./output/layout4.oas
21          baseline: ./baseline/layout4.oas
22          layers: [1,2,9]
23          bins: [1,2,5,10]
24          exclude_bins: [1]
25          exclude_bins_if:
26              VARs:
27                  - [IGNORE_MORE_BINS=yes,RELAX_XOR=1]
28                  - bins: [1,2,5]
```

Fig. 20. *xor* mode examples.

5.7 Custom mode

The Versus framework can be extended to accommodate any custom comparison engine and integrate it within the framework, enabling users to realize the framework's benefits while using the comparison engine that meets their functional needs. To implement this process, users employ the Versus environment variable *QA_VERSUS_CUSTOM_MODE*, as shown in Fig. 21. This variable must point to the custom comparison engine (or engines) so the framework can recognize and use them in the comparison checks.

```
1   #!/bin/sh
2
3   #define custom mode engine
4   export QA_VERSUS_CUSTOM_MODE="/path/to/custom_mode_engine/compare_database.sh"
5
6   #run Versus
7   /path/to/versus --yaml versus.yaml
8   status=$?
9
10  #exit with proper status
11  exit $status
```

Fig. 21. Custom mode usage.

Once the *QA_VERSUS_CUSTOM_MODE* variable is defined, users can use the custom mode in the YAML configuration file just as they do any of the pre-defined modes, getting the benefits of the framework right out of the box. In the example shown in Fig. 22, a DRC Results Database comparison engine (which compares Siemens EDA format databases used in the Calibre nmDRC tool) is used with the Versus custom mode.

```
1   compare_metrics:
2       - MY_MODE:
3           mode: compare_database
4           metric: ./waived.rdb
5           baseline: baseline/waived2.rdb
```

Fig. 22. Custom mode configuration file example.

6 Versus output log and summary report

The output log of the Versus framework is designed to give users all the information they need about the comparisons without being distracted by lengthy or unnecessary data that would add time to the test failure analysis without adding any significant value.

Fig. 23 shows all the relevant sections in a typical Versus output log.

```
ahmedal::/home/ahmedal/versus_demo$ ./compare

Reading Configuration file:
  |  /home/ahmedal/versus_demo/versus.yaml

Sourcing env. varaibles:
  |  MGC_HOME=/path/to/calibre
  |  RUN_MULTI_CORE=yes
  |  CPU_COUNT=8
  |  REMOTES_COUNT=16

Check: NUMDIFF_CHECK
  |  /user/peteoss/aoi/bin/numdiff -z @ -S --separators=' \t\n\r, ;:|"'  baseline/out.csv ./out.csv
  |  FAILED, check this file: ./transcripts/numdiff_transcripts/NUMDIFF_CHECK.transcript

Check: NUMDIFF_CHECK_WITH_TOLERANCE
  |  /user/peteoss/aoi/bin/numdiff -z @ -S --separators=' \t\n\r, ;:|"'  -a 5 -r 0.1 baseline/out.csv ./out.csv
  |  PASS, no differences found, or differences within tolerance.

Check: IMAGE_CHECK
  |  correlationcompare  baseline/out.png ./out.png 100 --o ./transcripts/image_diff/image_IMAGE_CHECK_diff.png
  |  Images differ (correlation: 50.6388%, threshold: 100.0000%)
  |  FAILED, check this http://caluser.wv.mentorg.com//home/ahmedal/versus_demo/versus_demo_tests_paper/numdiff/transcripts/image_diff/image_IMAGE_CHECK_diff.png

Check: IMAGE_CHECK_WITH_TOLERANCE
  |  correlationcompare  baseline/out.png ./out.png 40.0 --o ./transcripts/image_diff/image_IMAGE_CHECK_WITH_TOLERANCE_diff.png
  |  Images match (correlation: 50.6388%, threshold: 40.0000%)
  |  PASS, no differences found, or differences within tolerance.

Check: XOR_CHECK
  |  xor_2_layouts.sh  -layout1 ./out.gds -layout2 baseline/out.gds
  |  FAILED, check this log file ./transcripts/xor_transcripts/check__XOR_CHECK/xor_2_layouts_XOR_CHECK.log
  |  to see the XOR differences, open $MGC_HOME/bin/calibrewb -m ./transcripts/xor_transcripts/check__XOR_CHECK/xor_result_XOR_CHECK.oas

Check: XOR_CHECK_WITH_TOLERANCE
  |  xor_2_layouts.sh  -layout1 ./out.gds -layout2 baseline/out.gds -bins 1,2,3,4 -ignore_bins 1,2
  |  PASS, no differences found, or differences within tolerance.

Compare Report:
+------------------------------+-----------------------------+--------------+------------+--------+------------------------------------------------------+
| check                        | baseline value [baseline]   | metric value | check type | Status | comments                                             |
+------------------------------+-----------------------------+--------------+------------+--------+------------------------------------------------------+
| NUMDIFF_CHECK                | baseline/out.csv            | ./out.csv    | NUMDIFF    | FAIL   | ./transcripts/numdiff_transcripts/NUMDIFF_CHECK.transcript |
+------------------------------+-----------------------------+--------------+------------+--------+------------------------------------------------------+
| NUMDIFF_CHECK_WITH_TOLERANCE | baseline/out.csv            | ./out.csv    | NUMDIFF    | PASS   |                                                      |
+------------------------------+-----------------------------+--------------+------------+--------+------------------------------------------------------+
| IMAGE_CHECK                  | baseline/out.png            | ./out.png    | IMAGE      | FAIL   |                                                      |
+------------------------------+-----------------------------+--------------+------------+--------+------------------------------------------------------+
| IMAGE_CHECK_WITH_TOLERANCE   | baseline/out.png            | ./out.png    | IMAGE      | PASS   |                                                      |
+------------------------------+-----------------------------+--------------+------------+--------+------------------------------------------------------+
| XOR_CHECK                    | baseline/out.gds            | ./out.gds    | XOR        | FAIL   | GEOMETRICAL                                          |
+------------------------------+-----------------------------+--------------+------------+--------+------------------------------------------------------+
| XOR_CHECK_WITH_TOLERANCE     | baseline/out.gds            | ./out.gds    | XOR        | PASS   |                                                      |
+------------------------------+-----------------------------+--------------+------------+--------+------------------------------------------------------+

Summary Report:
=================
Total:6
PASS :3
FAIL :3
=================
Final Status: FAIL, check the report above.
```

Configuration file source

Sourcing environment variables

Comparison checks report

Summary Report table

Final status Report

Fig. 23. Typical Versus output log, with each relevant section highlighted.

The following sections provide a closer look at each of the output log sections, providing more detail about the information presented, and the options designed to shorten failure analysis time.

6.1 Configuration file source and environment variables

Fig. 24 shows file sources and environmental variables used in the configuration file:

```
Reading Configuration file:
   |  /home/ahmedal/versus_demo/versus.yaml

Sourcing env. varaibles:
   |  MGC_HOME=/path/to/calibre
   |  RUN_MULTI_CORE=yes
   |  CPU_COUNT=8
   |  REMOTES_COUNT=16
```

Fig. 24. Configuration source and environment variables.

6.2 Comparison checks report

Fig. 25 shows the Versus comparison checks report. Each comparison check has its own mini check log:

- This log states the internal command used to make the comparison, and all the options and arguments used with this command, including the output and baseline files. It also states whether this check failed or passed, using color-coded lines to make reading the log easier and faster.
- In case of a check failure, the log directs users to the best option for checking the difference in the shortest time.

```
Check: NUMDIFF_CHECK
  |  /user/peteoss/aoi/bin/numdiff -z @ -S --separators=' \t\n\r, ;:|"'  baseline/out.csv ./out.csv
  |  FAILED, check this file: ./transcripts/numdiff_transcripts/NUMDIFF_CHECK.transcript

Check: NUMDIFF_CHECK_WITH_TOLERANCE
  |  /user/peteoss/aoi/bin/numdiff -z @ -S --separators=' \t\n\r, ;:|"'  -a 5 -r 0.1 baseline/out.csv ./out.csv
  |  PASS, no differences found, or differences within tolerance.

Check: IMAGE_CHECK
  |  correlationcompare  baseline/out.png ./out.png 100 --o ./transcripts/image_diff/image_IMAGE_CHECK_diff.png
  |  Images differ (correlation: 50.6388%, threshold: 100.0000%)
  |  FAILED, check this http://caluser.wv.mentorg.com//home/ahmedal/versus_demo/versus_demo_tests_paper/numdiff/transcripts/image_diff/image_IMAGE_CHECK_diff.png

Check: IMAGE_CHECK_WITH_TOLERANCE
  |  correlationcompare  baseline/out.png ./out.png 40.0 --o ./transcripts/image_diff/image_IMAGE_CHECK_WITH_TOLERANCE_diff.png
  |  Images match (correlation: 50.6388%, threshold: 40.0000%)
  |  PASS, no differences found, or differences within tolerance.

Check: XOR_CHECK
  |  xor_2_layouts.sh  -layout1 ./out.gds -layout2 baseline/out.gds
  |  FAILED, check this log file ./transcripts/xor_transcripts/check__XOR_CHECK/xor_2_layouts_XOR_CHECK.log
  |  to see the XOR differences, open $MGC_HOME/bin/calibrewb -m ./transcripts/xor_transcripts/check__XOR_CHECK/xor_result_XOR_CHECK.oas

Check: XOR_CHECK_WITH_TOLERANCE
  |  xor_2_layouts.sh  -layout1 ./out.gds -layout2 baseline/out.gds -bins 1,2,3,4 -ignore_bins 1,2
  |  PASS, no differences found, or differences within tolerance.
```

Fig. 25. Comparison checks report showing the check log for each comparison check.

6.3 Summary table report

The summary table report summarizes all the executed checks with their results in an organized table (Fig. 26). This report also includes the output and baseline file names, as well as comments (if possible) to direct users to the shortest path for checking the differences.

Compare Report:

check	baseline value [baseline]	metric value	check type	Status	comments
NUMDIFF_CHECK	baseline/out.csv	./out.csv	NUMDIFF	FAIL	./transcripts/numdiff_transcripts/NUMDIFF_CHECK.transcript
NUMDIFF_CHECK_WITH_TOLERANCE	baseline/out.csv	./out.csv	NUMDIFF	PASS	
IMAGE_CHECK	baseline/out.png	./out.png	IMAGE	FAIL	
IMAGE_CHECK_WITH_TOLERANCE	baseline/out.png	./out.png	IMAGE	PASS	
XOR_CHECK	baseline/out.gds	./out.gds	XOR	FAIL	GEOMETRICAL
XOR_CHECK_WITH_TOLERANCE	baseline/out.gds	./out.gds	XOR	PASS	

Fig. 26: Summary table report.

Copies may not be made or distributed for commercial use

6.4 Final pass/fail count report

The final pass/fail count is the last report that appears in the Versus output log (Fig. 27). It provides users with an overall view of how many checks failed and how many passed, so users can quickly determine how many checks passed without having to read the details of every single comparison check.

```
Summary Report:
===================
Total:6
PASS :3
FAIL :3
===================
Final Status: FAIL, check the report above.
```

Fig. 27. Final pass/fail counts report.

7 Versus configuration generator UI

Writing YAML code that contains multiple options can be challenging and time-consuming, so a web-based graphical user interface (GUI) tool was developed to create and edit Versus YAML configuration files (Fig. 28). This tool makes it easy to write Versus configuration files, eliminating the need to memorize any of the configuration syntax for all of the pre-defined modes while also reminding users of all of the available options the Versus framework offers, simplifying and speeding up configuration file creation for both new and experienced users.

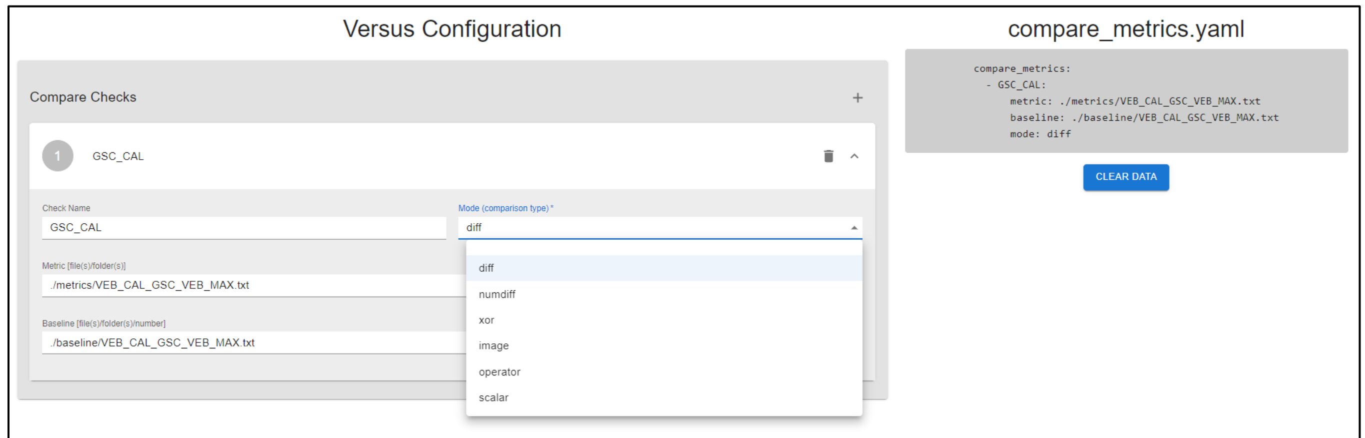

Fig. 28. Versus UI web-based tool for generating/editing configuration files.

8 Conclusion

The Versus framework is primarily designed for flexibility, maintainability, and control. It defines a standard for files comparison within automated regression suites, enabling QA engineers to use the best state-of-art tools for files comparison, while eliminating the need to write their own code every time a new comparison type is needed.

Acknowledgements

The authors would like to extend their gratitude to Siemens Calibre QA teams who assisted in testing the framework in-house, and provided insightful and helpful ideas and feedback to help the framework reach its maximum potential. We would also like to thank Shelly Stalnaker for editorial assistance in the preparation of this manuscript.

References

M. Kerrick, “The Linux Programming Interface,” Oct. 2010. https://man7.org/linux/man-pages/man1/diff.1.html

S. Vrkacevic, “Using command line tools for image comparisons,” froglogic, Oct 10, 2017. https://www.froglogic.com/blog/command-line-tools-for-image-comparisons/

I. Primi, “Numdiff,” Feb. 25, 2017. https://www.nongnu.org/numdiff/

“Open Artwork System Interchange Standard,” Layout Editor. https://www.layouteditor.org/layout/file-formats/oasis

“GDSII,” Layout Editor. https://www.layouteditor.org/layout/file-formats/gdsii