



JUBE Documentation

Release 2.2.0

**2017, JUBE Developer Team,
Forschungszentrum Jülich GmbH**

December 21, 2017

1	Introduction	1
2	JUBE tutorial	3
2.1	Installation	3
2.2	Configuration	3
2.3	Hello World	4
2.4	Help	6
2.5	Parameter space creation	6
2.6	Step dependencies	7
2.7	Loading files and substitution	8
2.8	Creating a result table	10
3	Advanced tutorial	13
3.1	Schema validation	13
3.2	Scripting parameter	14
3.3	Scripting pattern	15
3.4	Statistic pattern values	16
3.5	Jobsystem	17
3.6	Include external data	19
3.7	Tagging	21
3.8	Platform independent benchmarking	22
3.9	Multiple benchmarks	22
3.10	Shared operations	23
3.11	Environment handling	24
3.12	Parameter dependencies	24
3.13	Parameter update	25
3.14	Step iteration	26
3.15	Step cycle	27
4	Frequently Asked Questions	29
4.1	Parameter groups	29
4.2	Workdir change	29
4.3	XML character handling	30
4.4	Analyse multiple output files	30
5	Command line documentation	33
5.1	general	33
5.2	run	33
5.3	continue	34
5.4	analyse	34
5.5	result	35
5.6	comment	35
5.7	remove	35
5.8	info	36
5.9	log	36

5.10	status	36
5.11	complete	36
5.12	help	37
5.13	update	37
6	Glossary	39
	Index	53

INTRODUCTION

Automating benchmarks is important for reproducibility and hence comparability which is the major intent when performing benchmarks. Furthermore managing different combinations of parameters is error-prone and often results in significant amounts work especially if the parameter space gets large.

In order to alleviate these problems *JUBE* helps performing and analyzing benchmarks in a systematic way. It allows custom work flows to be able to adapt to new architectures.

For each benchmark application the benchmark data is written out in a certain format that enables *JUBE* to deduct the desired information. This data can be parsed by automatic pre- and post-processing scripts that draw information, and store it more densely for manual interpretation.

The *JUBE* benchmarking environment provides a script based framework to easily create benchmark sets, run those sets on different computer systems and evaluate the results. It is actively developed by the Jülich Supercomputing Centre of Forschungszentrum Jülich, Germany.

JUBE TUTORIAL

This tutorial is meant to give you an overview about the basic usage of *JUBE*.

Installation

Requirements: *JUBE* needs **Python 2.7** or **Python 3.2** (or any higher version)

You also can use **Python 2.6** to run *JUBE*. In this case you had to add the `argparse-module` to your *Python* module library on your own.

To use the *JUBE* command line tool, the `PYTHONPATH` must contain the position of the *JUBE* package. This can be achieved in three different ways:

- You can use the **installation tool** to copy all files to the right position (preferred):

```
>>> python setup.py install --user
```

This will install the *JUBE* package and the binary to your `$HOME/.local` directory. Instead of `--user` also a user specific `--prefix` option is available. Here you might have to set the `PYTHONPATH` environment variable first (this will be mentioned during the install process).

- You can add the **parent folder path** of the *JUBE* package-folder (`jube2` directory) to the `PYTHONPATH` environment variable:

```
>>> export PYTHONPATH=<parent folder path>:$PYTHONPATH
```

- You can move the *JUBE* package by hand to an existing Python package folder like `site-packages`

To use the *JUBE* command line tool like a normal command line command you can add it to the `PATH` environment variable:

```
>>> export PATH=$HOME/.local/bin:$PATH
```

To check your final installation, you can use

```
>>> jube --version
```

which should highlight you the current version number.

Configuration

The main *JUBE* configuration bases on the given input configuration file. But in addition, some shell environment variables are available which can be used to set system specific options:

- `JUBE_INCLUDE_PATH`: Can contain a list of pathes (seperated by `:`) pointing to directories, which contain system relevant include configuration files. This technique can be used to store platform specific parameter in a platform specific directory.

- `JUBE_EXEC_SHELL`: *JUBE* normally uses `/bin/sh` to execute the given shell commands. This default shell can be changed by using this environment variable.
- `JUBE_GROUP_NAME`: *JUBE* will use the given *UNIX* groupname to share benchmarks between different users. The group must exist and the *JUBE* user must be part of this group. The given group will be the owner of new benchmark runs. By default (without setting the environment variable) all file and directory permissions are defined by the normal *UNIX* rules.

BASH autocompletion can be enabled by using the `eval "$(jube complete)"` command. You can store the command in your bash profile settings if needed.

Hello World

In this example we will show you the basic structure of a *JUBE* input file and the basic command line options.

The files used for this example can be found inside `examples/hello_world`.

The input file `hello_world.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<jube>
  <benchmark name="hello_world" outpath="bench_run">
    <comment>A simple hello world</comment>

    <!-- Configuration -->
    <parameterset name="hello_parameter">
      <parameter name="hello_str">Hello World</parameter>
    </parameterset>

    <!-- Operation -->
    <step name="say_hello">
      <use>hello_parameter</use> <!-- use existing parameterset -->
      <do>echo $hello_str</do> <!-- shell command -->
    </step>
  </benchmark>
</jube>
```

Every *JUBE* input file starts (after the general *XML* header line) with the root tag `<jube>`. This root tag must be unique. *XML* does not allow multiple root tags.

The first tag which contains benchmark specific information is `<benchmark>`. `hello_world` is the benchmarkname which can be used to identify the benchmark (e.g. when there are multiple benchmarks inside a single input file, or when different benchmarks use the same run directory).

The `outpath` describes the benchmark run directory (relative to the position of the input file). This directory will be managed by *JUBE* and will be automatically created if it does not exist. The directory name and position are very important, because they are the main interface to communicate with your benchmark, after it was submitted.

Using the `<comment>` you can store some benchmark related comments inside the benchmark directory. You can also use normal *XML*-comments to structure your input-file:

```
<!-- your comment -->
```

In this benchmark a `<parameterset>` is used to store the single `<parameter name="hello_str">`. The name of the parameter should contain only letters, numbers (should not be the first character) or the `_` (like a normal *Python* identifier). The name of the parameterset must be unique (relative to the current benchmark). In further examples we will see that there are more types of sets, which can be distinguished by their names. Also the name of the parameter must be unique (relative to the parameterset).

The `<step>` contains the operation tasks. The name must be unique. It can use different types of existing sets. All used sets must be given by name using the `<use>`. There can be multiple `<use>` inside the same `<step>` and also multiple names within the same `<use>` are allowed (separated by `,`). Only sets, which are explicitly used, are available inside the step! The `<do>` contains a single **shell command**. This command will run inside of

a sandbox directory environment (inside the `outpath` directory tree). The step and its corresponding *parameter space* is named *workpackage*.

Available parameters can be used inside the shell commands. To use a parameter you have to write

```
$parametername
```

or

```
${parametername}
```

The brackets must be used if you want variable concatenation. `$hello_strtest` will not be replaced, `${hello_str}test` will be replaced. If a parameter does not exist or isn't available the variable will not be replaced! If you want to use `$` inside your command, you have to write `$$` to mask the symbol. Parameter substitution will be done before the normal shell substitution!

To run the benchmark just type:

```
>>> jube run hello_world.xml
```

This benchmark will produce the following output:

```
#####
# benchmark: hello_world

A simple hello world
#####

Running workpackages (#=done, 0=wait, E=error):
##### ( 1/ 1)

  stepname | all | open | wait | error | done
  -+-----+-----+-----+-----+-----
  say_hello | 1 | 0 | 0 | 0 | 1

>>>> Benchmark information and further useful commands:
>>>>     id: 0
>>>>   handle: bench_run
>>>>     dir: bench_run/000000
>>>> analyse: jube analyse bench_run --id 0
>>>>  result: jube result bench_run --id 0
>>>>   info: jube info bench_run --id 0
#####
```

As you can see, there was a single step `say_hello`, which runs one shell command `echo $hello_str` that will be expanded to `echo Hello World`.

The **id** is (in addition to the benchmark directory `handle`) an important number. Every benchmark run will get a new unique **id** inside the benchmark directory.

Inside the benchmark directory you will see the following structure:

```
bench_run          # the given outpath
|
+- 000000          # the benchmark id
|
+- configuration.xml # the stored benchmark configuration
+- workpackages.xml  # workpackage information
+- run.log           # log information
+- 000000_say_hello  # the workpackage
|
+- done             # workpackage finished marker
+- work             # user sandbox folder
|
+- stderr           # standard error messages of used shell commands
+- stdout           # standard output of used shell commands
```

stdout will contain `Hello World` in this example case.

Help

JUBE contains a command line based help functionality:

```
>>> jube help <keyword>
```

By using this command you will have direct access to all keywords inside the [glossary](#).

Another useful command is the `info` command. It will show you information concerning your existing benchmarks:

```
1 # display a list of existing benchmarks
2 >>> jube info <benchmark-directory>
3 # display information about given benchmark
4 >>> jube info <benchmark-directory> -- id <id>
5 # display information about a step inside the given benchmark
6 >>> jube info <benchmark-directory> -- id <id> --step <stepname>
```

The third, also very important, functionality is the **logger**. Every run, continue, analyse and result execution will produce log information inside your benchmark directory. This file contains much useful debugging output.

You can easily access these log files by using the *JUBE* log viewer command:

```
>>> jube log [benchmark-directory] [--id id] [--command cmd]
```

e.g.:

```
>>> jube log bench_runs --command run
```

will display the `run.log` of the last benchmark found inside of `bench_runs`.

Log output can also be displayed during runtime by using the verbose output:

```
>>> jube -v run <input-file>
```

`-vv` can be used to display stdout output during runtime and `-vvv` will display the stdout output as well as the log output at the same time.

Since the parsing step is done before creating the benchmark directory, there will be a `jube-parse.log` inside your current working directory, which contains the parser log information.

Errors within a `<do>` command will create a log entry and stop further execution of the corresponding parameter combination. Other parameter combinations will still be executed by default. *JUBE* can also stop automatically any further execution by using the `-e` option:

```
>>> jube run -e <input-file>
```

There is also a debugging mode integrated in *JUBE*:

```
>>> jube --debug <command> [other-args]
```

This mode avoids any *shell* execution but will generate a single log file (`jube-debug.log`) in your current working directory.

Parameter space creation

In this example we will show you an important feature of *JUBE*: The automatic *parameter space* generation.

The files used for this example can be found inside `examples/parameterspace`.

The input file `parameterspace.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<jube>
  <benchmark name="parameterspace" outpath="bench_run">
    <comment>A parameterspace creation example</comment>

    <!-- Configuration -->
    <parameterset name="param_set">
      <!-- Create a parameterspace out of two template parameter -->
      <parameter name="number" type="int">1,2,4</parameter>
      <parameter name="text" separator=";">Hello;World</parameter>
    </parameterset>

    <!-- Operation -->
    <step name="say_hello">
      <use>param_set</use> <!-- use existing parameterset -->
      <do>echo "$text $number"</do> <!-- shell command -->
    </step>
  </benchmark>
</jube>
```

Whenever a parameter contains a `,` (this can be changed using the `separator` attribute) this parameter becomes a **template**. A step which **uses the parameterset** containing this parameter will run multiple times to iterate over all possible parameter combinations. In this example the step `say_hello` will run 6 times:

stepname	all	open	wait	error	done
say_hello	6	0	0	0	6

Every parameter combination will run in its own sandbox directory.

Another new keyword is the `type` attribute. The parameter type is not used inside the substitution process, but for sorting operations inside the result creation. The default type is `string`. Possible basic types are `string`, `int` and `float`.

Step dependencies

If you start writing a complex benchmark structure, you might want to have dependencies between different *steps*, for example between a compile and the execution step. *JUBE* can handle these dependencies and will also preserve the given *parameter space*.

The files used for this example can be found inside `examples/dependencies`.

The input file `dependencies.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<jube>
  <benchmark name="dependencies" outpath="bench_run">
    <comment>A Dependency example</comment>

    <!-- Configuration -->
    <parameterset name="param_set">
      <parameter name="number" type="int">1,2,4</parameter>
    </parameterset>

    <!-- Operations -->
    <step name="first_step">
      <use>param_set</use> <!-- use existing parameterset -->
      <do>echo $number</do> <!-- shell command -->
    </step>
```

```
<!-- Create a dependency between both steps -->
<step name="second_step" depend="first_step">
  <do>cat first_step/stdout</do> <!-- shell command -->
</step>
</benchmark>
</jube>
```

In this example we create a dependency between `first_step` and `second_step`. After `first_step` finished, the corresponding `second_step` will start. Steps can also have multiple dependencies (separated by `,` in the definition), but circular definitions will not be resolved. A dependency is a unidirectional link!

To communicate between a step and its dependency there is a link inside the work directory pointing to the corresponding dependency step work directory. In this example we use

```
cat first_step/stdout
```

to write the `stdout`-file content of the dependency step into the `stdout`-file of the current step.

Because the `first_step` uses a template parameter which creates three execution runs, there will also be three `second_step` runs each pointing to different `first_step`-directories:

stepname	all	open	wait	error	done
first_step	3	0	0	0	3
second_step	3	0	0	0	3

Loading files and substitution

Every step runs inside a unique sandbox directory. Usually, you will need to have external files inside this directory (e.g. the source files) and in some cases you want to change a parameter inside the file based on your current *parameter space*. There are two additional set-types which handle this behaviour inside of *JUBE*.

The files used for this example can be found inside `examples/files_and_sub`.

The input file `files_and_sub.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<jube>
  <benchmark name="files_and_sub" outpath="bench_run">
    <comment>A file copy and substitution example</comment>

    <!-- Configuration -->
    <parameterset name="param_set">
      <parameter name="number" type="int">1,2,4</parameter>
    </parameterset>

    <!-- Files -->
    <fileset name="files">
      <copy>file.in</copy>
    </fileset>

    <!-- Substitute -->
    <substituteset name="substitute">
      <!-- Substitute files -->
      <iofile in="file.in" out="file.out" />
      <!-- Substitute commands -->
      <sub source="#NUMBER#" dest="$number" />
    </substituteset>

    <!-- Operation -->
    <step name="sub_step">
      <use>param_set</use> <!-- use existing parameterset -->
```

```

<use>files</use>           <!-- use existing fileset -->
<use>substitute</use>      <!-- use existing substituteset -->
<do>cat file.out</do>      <!-- shell command -->
</step>
</benchmark>
</jube>

```

The content of file `file.in`:

```
Number: #NUMBER#
```

Inside the `<fileset>` the current location (relatively to the current input file; also absolute paths are allowed) of files is defined. `<copy>` specifies that the file should be copied to the sandbox directory when the fileset is used. Also a `<link>` option is available to create a symbolic link to the given file inside the sandbox directory.

If there are additional operations needed to *prepare* your files (e.g. expand a tar-file). You can use the `<prepare>`-tag inside your `<fileset>`.

The `<substituteset>` describe the substitution process. The `<iofile>` contains the input and output file-name. The path is relative to the sandbox directory. Because we do/should not know that location we use the fileset to copy `file.in` to this directory.

The `<sub>` specifies the substitution. All occurrences of `source` will be substituted by `dest`. As you can see, you can use parameters inside the substitution.

There is no `<use>` inside any set. The combination of all sets will be done inside the `<step>`. So if you use a parameter inside a `<sub>` you must also add the corresponding `<parameterset>` inside the `<step>` where you use the `<substituteset>`!

In the `sub_step` we use all available sets. The use order is not relevant. The normal execution process will be:

1. Parameter space expansion
2. Copy/link files
3. Prepare operations
4. File substitution
5. Run shell operations

The resulting directory-tree will be:

```

bench_run          # the given outpath
|
+- 000000          # the benchmark id
  |
  +- configuration.xml # the stored benchmark configuration
  +- workpackages.xml # workpackage information
  +- 000000_say_hello # the workpackage ($number = 1)
    |
    +- done          # workpackage finished marker
    +- work          # user sandbox folder
      |
      +- stderr       # standard error messages of used shell commands
      +- stdout       # standard output of used shell commands (Number: 1)
      +- file.in      # the file copy
      +- file.out     # the substituted file
  +- 000001_say_hello # the workpackage ($number = 2)
    |
    +- ...
  +- ...

```

Creating a result table

Finally, after running the benchmark, you will get several directories. *JUBE* allows you to parse your result files distributed over these directories to extract relevant data (e.g. walltime information) and create a result table.

The files used for this example can be found inside `examples/result_creation`.

The input file `result_creation.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<jube>
  <benchmark name="result_creation" outpath="bench_run">
    <comment>A result creation example</comment>

    <!-- Configuration -->
    <parameterset name="param_set">
      <!-- Create a parameterspace with one template parameter -->
      <parameter name="number" type="int">1,2,4</parameter>
    </parameterset>

    <!-- Regex pattern -->
    <patternset name="pattern">
      <pattern name="number_pat" type="int">Number: $jube_pat_int</pattern>
    </patternset>

    <!-- Operation -->
    <step name="write_number">
      <use>param_set</use> <!-- use existing parameterset -->
      <do>echo "Number: $number"</do> <!-- shell command -->
    </step>

    <!-- Analyse -->
    <analyser name="analyse">
      <use>pattern</use> <!-- use existing patternset -->
      <analyse step="write_number">
        <file>stdout</file> <!-- file which should be scanned -->
      </analyse>
    </analyser>

    <!-- Create result table -->
    <result>
      <use>analyse</use> <!-- use existing analyser -->
      <table name="result" style="pretty" sort="number">
        <column>number</column>
        <column>number_pat</column>
      </table>
    </result>
  </benchmark>
</jube>
```

Using `<parameterset>` and `<step>` we create three *workpackages*. Each writing `Number: $number` to `stdout`.

Now we want to parse these `stdout` files to extract information (in this example case the written number). First of all we have to declare a `<patternset>`. Here we can describe a set of `<pattern>`. A `<pattern>` is a regular expression which will be used to parse your result files and search for a given string. In this example we only have the `<pattern>` `number_pat`. The name of the pattern must be unique (based on the usage of the `<patternset>`). The type is optional. It is used when the extracted data will be sorted. The regular expression can contain other patterns or parameters. The example uses `$jube_pat_int` which is a *JUBE default pattern* matching integer values. The pattern can contain a group, given by brackets `(...)`, to declare the extraction part (`$jube_pat_int` already contains these brackets).

E.g. `$jube_pat_int` and `$jube_pat_fp` are defined in the following way:

```
<pattern name="jube_pat_int" type="int"> ([+-]?\d+) </pattern>
<pattern name="jube_pat_fp" type="float"> ([+-]?\d*\.\d+ (?:[eE] [-+]? \d+ )?) </pattern>
```

If there are multiple matches inside a single file you can add a *reduce option*. Normally only the first match will be extracted.

To use your `<patternset>` you have to specify the files which should be parsed. This can be done using the `<analyser>`. It uses relevant patterns. Inside the `<analyse>` a step-name and a file inside this step is given. Every workpackage file combination will create its own result entry.

The analyser automatically knows all parameters which were used in the given step and in depending steps. There is no `<use>` option to add additional completely new parameters.

To run the analyse you have to write:

```
>>> jube analyse bench_run
```

The analyse data will be stored inside the benchmark directory.

The last part is the result table creation. Here you have to use an existing analyser. The `<column>` contains a pattern or a parameter name. `sort` is the optional sorting order (separated by `,`). The `style` attribute can be `csv` or `pretty` to get different ASCII representations.

To create the result table you have to write:

```
>>> jube result bench_run -i last
```

The result table will be written to STDOUT and into a `result.dat` file inside `bench_run/<id>/result`. The `last` can also be replaced by a specific benchmark id. If the id selection is missing a combined result table of all available benchmark runs from the `bench_run` directory will be created.

Output of the given example:

number	number_pat
1	1
2	2
4	4

This was the last example of the basic *JUBE* tutorial. Next you can start the [advanced tutorial](#) to get more information about including external sets, jobssystem representation and scripting parameter.

ADVANCED TUTORIAL

This tutorial demonstrates more detailed functions and tools of *JUBE*. If you want a basic overview you should read the general [JUBE tutorial](#) first.

Schema validation

To validate your input files you can use DTD or schema validation. You will find `jube.dtd`, `jube.xsd` and `jube.rnc` inside the `schema` folder. You have to add these schema information to your input files which you want to validate.

DTD usage:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE jube SYSTEM "<jube.dtd path>">
3 <jube>
4 ...
```

Schema usage:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <jube xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3   xsi:noNamespaceSchemaLocation="<jube.xsd path>">
4 ...
```

RELAX NG Compact Syntax (RNC for emacs nxml-mode) usage:

In order to use the provided rnc schema file `schema/jube.rnc` in emacs open an xml file and use `C-c C-s C-f` or `M-x rng-set-schema-file-and-validate` to choose the rnc file. You can also use `M-x customize-variable rng-schema-locating-files` after you loaded nxml-mode to customize the default search paths to include `jube.rnc`. After successful parsing emacs offers to automatically create a `schema.xml` file which looks like

```
1 <?xml version="1.0"?>
2 <locatingRules xmlns="http://thaiopensource.com/ns/locating-rules/1.0">
3   <uri resource="jube-file.xml" uri="../schema/jube.rnc"/>
4 </locatingRules>
```

The next time you open the same xml file emacs will find the correct rnc for the validation based on `schema.xml`.

Example validation tools:

- eclipse (using DTD or schema)
- emacs (using RELAX NG)
- xmllint:
 - For validation (using the DTD):

```
>>> xmllint --noout --valid <xml input file>
```

- For validation (using the DTD and Schema):

```
>>> xmllint --noout --valid --schema <schema file> <xml input file>
```

Scripting parameter

In some cases it is needed to create a parameter which is based on the value of another parameter. In this case you can use a scripting parameter.

The files used for this example can be found inside `examples/scripting_parameter`.

The input file `scripting_parameter.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<jube>
  <benchmark name="scripting_parameter" outputpath="bench_run">
    <comment>A scripting parameter example</comment>

    <!-- Configuration -->
    <parameterset name="param_set">
      <!-- Normal template -->
      <parameter name="number" type="int">1,2,4</parameter>
      <!-- A template created by a scripting parameter -->
      <parameter name="additional_number" mode="python" type="int">
        ", ".join(str(a*${number}) for a in [1,2])
      </parameter>
      <!-- A scripting parameter -->
      <parameter name="number_mult" mode="python" type="float">
        ${number}*${additional_number}
      </parameter>
      <!-- Normal template -->
      <parameter name="text">Number: $number</parameter>
    </parameterset>

    <!-- Operation -->
    <step name="operation">
      <use>param_set</use> <!-- use existing parameterset -->
      <!-- shell commands -->
      <do>echo "number: $number, additional_number: $additional_number"</do>
      <do>echo "number_mult: $number_mult, text: $text"</do>
    </step>
  </benchmark>
</jube>
```

In this example we see four different parameters.

- `number` is a normal template which will be expanded to three different *workpackages*.
- `additional_number` is a scripting parameter which creates a new template and bases on `number`. The mode is set to the scripting language (python, perl and shell are allowed). The additional type is optional and declares the result type after evaluating the expression. The type is only used by the sort algorithm in the result step. It is not possible to create a template of different scripting parameters. Because of this second template we will get six different *workpackages*.
- `number_mult` is a small calculation. You can use any other existing parameters (which are used inside the same step).
- `text` is a normal parameter which uses the content of another parameter. For a simple concatenation parameter you do not need a scripting parameter.

For this example we will find the following output inside the `run.log`-file:

```

===== operation =====
>>> echo "number: 1, additional_number: 1"
>>> echo "number_mult: 1, text: Number: 1"
===== operation =====
>>> echo "number: 1, additional_number: 2"
>>> echo "number_mult: 2, text: Number: 1"
===== operation =====
>>> echo "number: 2, additional_number: 2"
>>> echo "number_mult: 4, text: Number: 2"
===== operation =====
>>> echo "number: 2, additional_number: 4"
>>> echo "number_mult: 8, text: Number: 2"
===== operation =====
>>> echo "number: 4, additional_number: 4"
>>> echo "number_mult: 16, text: Number: 4"
===== operation =====
>>> echo "number: 4, additional_number: 8"
>>> echo "number_mult: 32, text: Number: 4"

```

Implicit Perl or Python scripting inside the `<do>` or any other position is not possible. If you want to use some scripting expressions you have to create a new parameter.

Scripting pattern

Similar to the *Scripting parameter* also different patterns, or patterns and parameters can be combined. For this a scripting pattern can be created by using the `mode=` attribute in the same way as it is used for the *Scripting parameter*.

All scripting patterns are evaluated at the end of the analyse part. Each scripting pattern is evaluated once. If there are multiple matches as described in the *Statistic pattern values* section, only the resulting statistical pattern is available (not each individual value). Scripting pattern do not create statistic values by themselves.

In addition the `default=` attribute can be used to set a default pattern value, if the value can't be found during the analysis.

The files used for this example can be found inside `examples/scripting_pattern`.

The input file `scripting_pattern.xml`:

```

<?xml version="1.0" encoding="UTF-8"?>
<jube>
  <benchmark name="scripting_pattern" outpath="bench_run">
    <comment>A scripting_pattern example</comment>

    <!-- Configuration -->
    <parameterset name="param_set">
      <parameter name="value" type="int">0,1,2</parameter>
    </parameterset>

    <!-- Operation -->
    <step name="operation">
      <use>param_set</use>
      <do>echo "$value"</do>
    </step>

    <!-- Pattern to extract -->
    <patternset name="pattern_set">
      <!-- A normal pattern -->
      <pattern name="value_pat" type="int">$jube_pat_int</pattern>
      <!-- A combination of a pattern and a parameter -->
      <pattern name="dep_pat" type="int" mode="python">$value_pat+$value</pattern>
      <!-- This pattern is not available -->

```

```

<pattern name="missing_pat" type="int">
  pattern_not_available: $jube_pat_int
</pattern>
<!-- The combination will fail (create NaN) -->
<pattern name="missing_dep_pat" type="int" mode="python">
  $missing_pat*$value
</pattern>
<!-- Default value for missing pattern -->
<pattern name="missing_pat_def" type="int" default="0">
  pattern_not_available: $jube_pat_int
</pattern>
<!-- Combination of default value and parameter -->
<pattern name="missing_def_dep_pat" type="int" mode="python">
  $missing_pat_def*$value
</pattern>
</patternset>

<analyser name="analyse">
  <use>pattern_set</use>
  <analyse step="operation">
    <file>stdout</file>
  </analyse>
</analyser>

<!-- result table creation -->
<result>
  <use>analyse</use>
  <table name="result" style="pretty">
    <column>value</column>
    <column>value_pat</column>
    <column>dep_pat</column>
    <column>missing_pat</column>
    <column>missing_dep_pat</column>
    <column>missing_pat_def</column>
    <column>missing_def_dep_pat</column>
  </table>
</result>
</benchmark>
</jube>

```

It will create the following output:

value	value_pat	dep_pat	missing_pat	missing_dep_pat	missing_pat_def	missing_def_dep_pat
0	0	0		nan	0	
1	1	2		nan	0	
2	2	4		nan	0	

Statistic pattern values

Normally a pattern should only match a single entry in your result files. But sometimes there are multiple similar entries (e.g. if the benchmark uses some iteration feature).

JUBE will create the statistical values last, min, max, avg, std, cnt and sum automatically. To use these values, the user had to specify the pattern name followed by `<statistic_option>`, e.g. `pattern_name_last` (the `pattern_name` itself will always be the first match).

An example for multiple matches and the statistic values can be found in `examples/statistic`.

The input file `statistic.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<jube>
  <benchmark name="reduce_example" outpath="bench_run">
    <comment>A result reduce example</comment>

    <!-- Regex pattern -->
    <patternset name="pattern">
      <pattern name="number_pat" type="int">$jube_pat_int</pattern>
    </patternset>

    <!-- Operation -->
    <step name="write_some_numbers">
      <do>echo "1 2 3 4 5 6 7 8 9 10"</do> <!-- shell command -->
    </step>

    <!-- Analyse -->
    <analyser name="analyse">
      <use>pattern</use> <!-- use existing patternset -->
      <analyse step="write_some_numbers">
        <file>stdout</file> <!-- file which should be scanned -->
      </analyse>
    </analyser>

    <!-- Create result table -->
    <result>
      <use>analyse</use> <!-- use existing analyser -->
      <table name="result" style="pretty">
        <column>number_pat</column> <!-- first match -->
        <column>number_pat_last</column> <!-- last match -->
        <column>number_pat_min</column> <!-- min of all matches -->
        <column>number_pat_max</column> <!-- max of all matches -->
        <column>number_pat_sum</column> <!-- sum of all matches -->
        <column>number_pat_cnt</column> <!-- number of matches -->
        <column>number_pat_avg</column> <!-- avg of all matches -->
        <column format=".2f">number_pat_std</column> <!-- std of all matches -->
      </table>
    </result>
  </benchmark>
</jube>
```

It will create the following output:

number_pat	number_pat_last	number_pat_min	number_pat_max	number_pat_sum	number_pat_cnt
1	10	1	10	55	10

Jobsystem

In most cases you want to submit jobs by *JUBE* to your local jobsystem. You can use the normal file access and substitution system to prepare your jobfile and send it to the jobsystem. *JUBE* also provide some additional features.

The files used for this example can be found inside `examples/jobsystem`.

The input jobsystem file `job.run.in` for *Torque/Moab* (you can easily adapt your personal jobscrip):

```
#!/bin/bash -x
#MSUB -l nodes=#NODES#:ppn=#PROCS_PER_NODE#
#MSUB -l walltime=#WALLTIME#
#MSUB -e #ERROR_FILEPATH#
#MSUB -o #OUT_FILEPATH#
```

```
#MSUB -M #MAIL_ADDRESS#
#MSUB -m #MAIL_MODE#

### start of jobscript

#EXEC#
touch #READY#
```

The *JUBE* input file `jobsystem.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<jube>
  <benchmark name="jobsystem" outpath="bench_run">
    <comment>A jobsystem example</comment>

    <!-- benchmark configuration -->
    <parameterset name="param_set">
      <parameter name="number" type="int">1,2,4</parameter>
    </parameterset>

    <!-- Job configuration -->
    <parameterset name="executeset">
      <parameter name="submit_cmd">msub</parameter>
      <parameter name="job_file">job.run</parameter>
      <parameter name="nodes" type="int">1</parameter>
      <parameter name="walltime">00:01:00</parameter>
      <parameter name="ppn" type="int">4</parameter>
      <parameter name="ready_file">ready</parameter>
      <parameter name="mail_mode">abe</parameter>
      <parameter name="mail_address"></parameter>
      <parameter name="err_file">stderr</parameter>
      <parameter name="out_file">stdout</parameter>
      <parameter name="exec">echo $number</parameter>
    </parameterset>

    <!-- Load jobfile -->
    <fileset name="files">
      <copy>${job_file}.in</copy>
    </fileset>

    <!-- Substitute jobfile -->
    <substituteset name="sub_job">
      <iofile in="${job_file}.in" out="$job_file" />
      <sub source="#NODES#" dest="$nodes" />
      <sub source="#PROCS_PER_NODE#" dest="$ppn" />
      <sub source="#WALLTIME#" dest="$walltime" />
      <sub source="#ERROR_FILEPATH#" dest="$err_file" />
      <sub source="#OUT_FILEPATH#" dest="$out_file" />
      <sub source="#MAIL_ADDRESS#" dest="$mail_address" />
      <sub source="#MAIL_MODE#" dest="$mail_mode" />
      <sub source="#EXEC#" dest="$exec" />
      <sub source="#READY#" dest="$ready_file" />
    </substituteset>

    <!-- Operation -->
    <step name="submit" work_dir="$WORK/jobsystem_bench_${jube_benchmark_id}_${jube_wp_id}" >
      <use>param_set</use>
      <use>executeset</use>
      <use>files,sub_job</use>
      <do done_file="$ready_file">$submit_cmd $job_file</do> <!-- shell command -->
    </step>
  </benchmark>
</jube>
```

As you can see the jobfile is very general and several parameters will be used for replacement. By using a general jobfile and the substitution mechanism you can control your jobsystem directly out of your *JUBE* input file.

The submit command is a normal *Shell* command so there are no special *JUBE* tags to submit a job.

There are two new attributes:

- `done_file` inside the `<do>` allows you to set a filename/path to a file which should be used by the jobfile to mark the end of execution. *JUBE* does not know when the job ends. Normally it will return when the *Shell* command was finished. When using a jobsystem we had to wait until the jobfile was executed. If *JUBE* found a `<do>` containing a `done_file` attribute *JUBE* will return directly and will not continue automatically until the `done_file` exists. If you want to check the current status of your running steps and continue the benchmark process if possible you can type:

```
>>> jube continue bench_run
```

This will continue your benchmark execution (`bench_run` is the benchmarks directory in this example). The position of the `done_file` is relatively seen towards the work directory.

- `work_dir` can be used to change the sandbox work directory of a step. In normal cases *JUBE* checks that every work directory gets a unique name. When changing the directory the user must select a unique name by his own. For example he can use `$jube_benchmark_id` and `$jube_wp_id`, which are *JUBE internal parameters* and will be expanded to the current benchmark and workpackage ids. Files and directories out of a given `<fileset>` will be copied into the new work directory. Other automatic links, like the dependency links, will not be created!

You will see this Output after running the benchmark:

stepname	all	open	wait	error	done
submit	3	0	3	0	0

and this output after running the `continue` command (after the jobs where executed):

stepname	all	open	wait	error	done
submit	3	0	0	0	3

You have to run `continue` multiple times if not all `done_file` were written when running `continue` for the first time.

Include external data

As you have seen in the example before a benchmark can become very long. To structure your benchmark you can use multiple files and reuse existing sets. There are three different include features available.

The files used for this example can be found inside `examples/include`.

The include file `include_data.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<jube>
  <parameterset name="param_set">
    <parameter name="number" type="int">1,2,4</parameter>
  </parameterset>

  <parameterset name="param_set2">
    <parameter name="text">Hello</parameter>
  </parameterset>

  <dos>
    <do>echo Test</do>
    <do>echo $number</do>
  </dos>
</jube>
```

```
</dos>
</jube>
```

All files which contain data to be included must use the *XML*-format. The include files can have a user specific structure (there can be none valid *JUBE* tags like `<dos>`), but the structure must be allowed by the searching mechanism (see below). The resulting file must have a valid *JUBE* structure.

The main file `main.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<jube>
  <benchmark name="include" outpath="bench_run">
    <comment>A include example</comment>

    <!-- use parameterset out of an external file and add a additional parameter -->
    <parameterset name="param_set" init_with="include_data.xml">
      <parameter name="foo">bar</parameter>
    </parameterset>

    <!-- Operation -->
    <step name="say_hello">
      <use>param_set</use> <!-- use existing parameterset -->
      <use from="include_data.xml">param_set2</use> <!-- out of an external file -->
      <do>echo $foo</do> <!-- shell command -->
      <include from="include_data.xml" path="dos/do" /> <!-- include all available tag -->
    </step>
  </benchmark>
</jube>
```

In these file there are three different include types:

The `init_with` can be used inside any set definition. Inside the given file the search mechanism will search for the same set (same type, same name), will parse its structure (this must be *JUBE* valid) and copy the content to `main.xml`. Inside `main.xml` you can add additional values or overwrite existing ones. If your include-set uses a different name inside your include file you can use `init_with="filename.xml:new_name"`.

The second method is the `<use from="...">`. This is mostly the same like the `init_with` structure, but in this case you are not able to add or overwrite some values. The external set will be used directly. There is no set-type inside the `<use>`, because of that, the set's name must be unique inside the include-file.

The last method is the most generic include. By using `<include />` you can copy any *XML*-nodes you want to your main-*XML* file. The included file can provide tags which are not *JUBE*-conform but it must be a valid *XML*-file (e.g. only one root node allowed). The resulting main configuration file must be completely *JUBE* valid. The path is optional and can be used to select a specific node set (otherwise the root-node itself will be included). The `<include />` is the only include-method that can be used to include any tag you want. The `<include />` will copy all parts without any changes. The other include types will update path names, which were relative to the include-file position.

To run the benchmark you can use the normal command:

```
>>> jube run main.xml
```

It will search for include files inside four different positions (in the following order):

- inside a directory given over the command line interface:

```
>>> jube run --include-path some_path another_path -- main.xml
```

- inside any path given by a `<include-path>`-tag:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <benchmarks>
3   <include-path>
4     <path>some_path</path>
5     <path>another_path</path>
```



```
6     </include-path>
7     ...
```

- inside any path given with the JUBE_INCLUDE_PATH environment variable (see *Configuration*):

```
>>> export JUBE_INCLUDE_PATH=some_path:another_path
```

- inside the same directory of your main.xml

Tagging

Tagging is a easy way to hide selectable parts of your input file.

The files used for this example can be found inside `examples/tagging`.

The input file `tagging.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<jube>
  <benchmark name="tagging" outpath="bench_run">
    <comment>A simple tagging example</comment>

    <!-- Configuration -->
    <parameterset name="param_set">
      <parameter name="hello_str" tag="eng">Hello</parameter>
      <parameter name="hello_str" tag="deu">Hallo</parameter>
      <parameter name="world_str" tag="!deu">World</parameter>
    </parameterset>

    <!-- Operation -->
    <step name="say_hello">
      <use>param_set</use> <!-- use existing parameterset -->
      <do>echo '$hello_str $world_str'</do> <!-- shell command -->
    </step>
  </benchmark>
</jube>
```

When running this example:

```
>>> jube run tagging.xml
```

all <tags> which contain a special tag="..." attribute will be hidden. !deu stands for not deu so this tag will not be hidden when running the command.

The result inside the stdout file will be

```
$hello_str World
```

because there was no alternative to select the \$hello_str.

When running this example using a specific tag:

```
>>> jube run tagging.xml --tag eng
```

the result inside the stdout file will be

```
Hello World
```

The tag attribute or the command line expression can also contain a list of different names. A hidden <tag> will be ignored completely! If there is no alternative this can produce a wrong execution behaviour!

The tag attribute can be used inside every <tag> inside the input file (except the <jube>).

Platform independent benchmarking

If you want to create platform independent benchmarks you can use the include features inside of *JUBE*.

All platform related sets must be declared in a includable file e.g. `platform.xml`. There can be multiple `platform.xml` in different directories to allow different platforms. By changing the `include-path` the benchmark changes its platform specific data.

An example benchmark structure bases on three include files:

- The main benchmark include file which contain all benchmark specific but platform independent data
- A mostly generic platform include file which contain benchmark independent but platform specific data (this can be created once and placed somewhere central on the system, it can be easily accessed using the `JUBE_INCLUDE_PATH` environment variable.
- A platform specific and benchmark specific include file which must be placed in a unique directory to allow `include-path` usage

Inside the `platform` directory you will find some example benchmark independent platform configuration files for the supercomputers at Forschungszentrum Jülich.

To avoid writing long include-paths every time you run a platform independent benchmark, you can store the `include-path` inside your input file. This can be mixed using the tagging-feature:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <jube>
3   <include-path>
4     <path tag="plat1">some path</path>
5     <path tag="plat2">another path</path>
6     ...
7   </include-path>
8   ...
9 </jube>
```

Now you can run your benchmark using:

```
>>> jube run filename.xml --tag plat1
```

Multiple benchmarks

Often you only have one benchmark inside your input file. But it is also possible to store multiple benchmarks inside the same input file:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <jube>
3   <benchmark name="a" outpath="bench_runs">...</benchmark>
4   <benchmark name="b" outpath="bench_runs">...</benchmark>
5   ...
6 </jube>
```

All benchmarks can use the same global (as a child of `<jube>`) declared sets. Often it might be better to use an include feature instead. *JUBE* will run every benchmark in the given order. Every benchmark gets an unique benchmark id.

To select only one benchmark you can use:

```
>>> jube run filename.xml --only-bench a
```

or:

```
>>> jube run filename.xml --not-bench b
```

This information can also be stored inside the input file:

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <jube>
3   <selection>
4     <only>a</only>
5     <not>b</not>
6   </selection>
7   ...
8 </jube>

```

Shared operations

Sometimes you want to communicate between the different workpackages of a single step or you want a single operation to run only once for all workpackages. Here you can use shared steps.

The files used for this example can be found inside `examples/shared`.

The input file `shared.xml`:

```

<?xml version="1.0" encoding="UTF-8"?>
<jube>
  <benchmark name="shared" outpath="bench_run">
    <comment>A shared folder example</comment>

    <!-- Configuration -->
    <parameterset name="param_set">
      <parameter name="number" type="int">1,2,4</parameter>
    </parameterset>

    <!-- Operation -->
    <step name="a_step" shared="shared">
      <use>param_set</use>
      <!-- shell command will run three times -->
      <do>echo $jube_wp_id >> shared/all_ids</do>
      <!-- shell command will run one time -->
      <do shared="true">cat all_ids</do>
    </step>
  </benchmark>
</jube>

```

The step must be marked using the `shared` attribute. The name, given inside this attribute, will be the name of a symbolic link, which will be created inside every single sandbox work directory pointing to a single shared folder. Every Workpackage can access this folder by using its own link. In this example every workpackage will write its own id into a shared file (`$jube_wp_id` is an internal variable, more of these you will find [here](#)).

To mark an operation to be a shared operation `shared="true"` inside the `<do>` must be used. The shared operation will start after all workpackages reached its execution position. The work directory for the shared operation is the shared folder itself.

You will get the following directory structure:

```

bench_run          # the given outpath
|
+- 000000          # the benchmark id
|
+- configuration.xml # the stored benchmark configuration
+- workpackages.xml  # workpackage information
+- 000000_a_step    # the first workpackage
|
+- done            # workpackage finished marker
+- work            # user sandbox folder

```

```
|
+- stderr      # standard error messages of used shell commands
+- stdout      # standard output of used shell commands
+- shared      # symbolic link pointing to shared folder
+- 000001_a_step # workpackage information
+- 000002_a_step # workpackage information
+- a_step_shared # the shared folder
|
+- stdout      # standard output of used shell commands
+- stderr      # standard error messages of used shell commands
+- all_ids     # benchmark specific generated file
```

Environment handling

Shell environment handling can be very important to configure pathes or parameter of your program.

The files used for this example can be found inside `examples/environment`.

The input file `environment.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<jube>
  <benchmark name="environment" outpath="bench_run">
    <comment>An environment handling example</comment>

    <!-- Configuration -->
    <parameterset name="param_set">
      <parameter name="EXPORT_ME" export="true">VALUE</parameter>
    </parameterset>

    <!-- Operations -->
    <step name="first_step" export="true">
      <do>export SHELL_VAR=Hello</do> <!-- export a Shell var -->
      <do>echo "$$SHELL_VAR world"</do><!-- use exported Shell var -->
    </step>

    <!-- Create a dependency between both steps -->
    <step name="second_step" depend="first_step">
      <use>param_set</use>
      <do>echo $$EXPORT_ME</do>
      <do>echo "$$SHELL_VAR again"</do> <!-- use exported Shell var out of privious step -->
    </step>
  </benchmark>
</jube>
```

In normal cases all `<do>` within one `<step>` shares the same environment. All **exported** variables of one `<do>` will be available inside the next `<do>` within the same `<step>`.

By using `export="true"` inside of a `<parameter>` you can export additional variables to your *Shell* environment. Be aware that this example uses `$$` to explicitly use *Shell* substitution instead of *JUBE* substitution.

You can also export the complete environment of a step to a dependent step by using `export="true"` inside of `<step>`.

Parameter dependencies

Sometimes you need parameters which based on other parameters or only a specific parameter combination make sense and other combinations are useless or wrong. For this there are several techniques inside of *JUBE* to create such a more complex workflow.

The files used for this example can be found inside `examples/parameter_dependencies`.

The input file `parameter_dependencies.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<jube>
  <benchmark name="parameter_dependencies" outpath="bench_run">
    <comment>A parameter_dependencies example</comment>

    <!-- Configuration -->
    <parameterset name="param_set">
      <parameter name="index" type="int">0,1</parameter>
      <parameter name="text" mode="python">["hello","world"][$index]</parameter>
    </parameterset>

    <parameterset name="depend_param_set0">
      <parameter name="number" type="int">3,5</parameter>
    </parameterset>

    <parameterset name="depend_param_set1">
      <parameter name="number" type="int">1,2,4</parameter>
    </parameterset>

    <!-- Operation -->
    <step name="operation">
      <use>param_set</use> <!-- use basic parameterset -->
      <use>depend_param_set$index</use> <!-- use dependent parameterset -->
      <use from="include_file.xml:depend_param_set0:depend_param_set1">
        depend_param_set$index
      </use>
      <do>echo "$text $number $number2"</do>
    </step>
  </benchmark>
</jube>
```

The include file `include_file.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<jube>
  <parameterset name="depend_param_set0">
    <parameter name="number2" type="int">10</parameter>
  </parameterset>

  <parameterset name="depend_param_set1">
    <parameter name="number2" type="int">20</parameter>
  </parameterset>
</jube>
```

The easiest way to handle dependencies is to define an index-parameter which can be used in other scripting parameters to combine all dependent parameter combinations.

Also complete sets can be marked as dependent towards a specific parameter by using this parameter in the `<use>`-tag. When using parametersets out of an other file the correct set-name must be given within the `from` attribute, because these sets will be loaded in a pre-processing step before the corresponding parameter will be evaluated. Also sets out of different files can be combined within the same `<use>` by using the `file1:set1,file2:set2` syntax. The sets names must be unique.

Parameter update

Once a parameter is specified and evaluated the first time, its value will not change. Sometimes this behaviour can produce the wrong behaviour:

```
<parameter name="foo">$jube_wp_id</parameter>
```

In this example `foo` should hold the `$jube_wp_id`. If you have two steps, where one step depends on the other one `foo` will be available in both, but it will only be evaluated in the first one.

There is a simple work-around to change the update behaviour of a parameter by using the attribute `update_mode`:

- `update_mode="never"` No update (default behaviour)
- `update_mode="use"` Re-evaluate the parameter if the parameterset is explicitly used
- `update_mode="step"` Re-evaluate the parameter for each new step
- `update_mode="cycle"` Re-evaluate the parameter for each new cycleloop, but not at the begin of a new step
- `update_mode="always"` Combine step and cycle

Within a cycle loop no new workpackages can be created. Templates will be reevaluated, but they can not increase the number of existing workpackages within a cycle.

Within the result generation, the parameter value, which is presented in the result table is the value of the selected analysed step. If another parameter representation is needed as well, all other steps can be reached by using `<parameter_name>_<step_name>`.

Step iteration

Especially in the context of benchmarking an application should be executed multiple times to generate some meaningful statistical values. The handling of statistical values is described in *Statistic pattern values*. This allows you to aggregate multiple result lines if your application automatically supports to run multiple times.

In addition there is also an iteration feature within JUBE to run a specific step and its parametrization multiple times.

The files used for this example can be found inside `examples/iterations`.

The input file `iterations.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<jube>
  <benchmark name="iterations" outpath="bench_run">
    <comment>A Iteration example</comment>

    <!-- Configuration -->
    <parameterset name="param_set">
      <parameter name="foo" type="int">1,2,4</parameter>
      <parameter name="bar" mode="text" update_mode="step">$foo iter:$jube_wp_iteration</parameter>
    </parameterset>

    <step name="first_step" iterations="2">
      <use>param_set</use> <!-- use existing parameterset -->
      <do>echo $bar</do> <!-- shell command -->
    </step>

    <step name="second_step" depend="first_step" iterations="2">
      <do>echo $bar</do> <!-- shell command -->
    </step>

    <!-- analyse without reduce -->
    <analyser name="analyse_no_reduce" reduce="false">
      <analyse step="second_step" />
    </analyser>
  </benchmark>
</jube>
```

```

<!-- Analyse with reduce -->
<analyser name="analyse" reduce="true">
  <analyse step="second_step" />
</analyser>

<result>
  <use>analyse</use>
  <use>analyse_no_reduce</use>
  <table name="result" style="pretty">
    <column>jube_res_analyser</column>
    <column>jube_wp_id_first_step</column>
    <column>jube_wp_id</column>
    <column>jube_wp_iteration_first_step</column>
    <column>jube_wp_iteration</column>
    <column>foo</column>
  </table>
</result>
</benchmark>
</jube>

```

In this example either step 1 as well as step 2 are executed 2 times for each parameter and dependency configuration. Because of the given parameter step 1 is executed 6 times in total (3 parameter combinations x 2). Step 2 is executed 12 times (6 from the dependent step x 2). Each run will be executed in the normal way using its individual sandbox folder.

\$jube_wp_iteration holds the individual iteration id. The update_mode is needed here to reevaluate the parameter bar in step 2.

In the analyser reduce=true or reduce=false can be enabled, to allow you to see all individual results or to aggregate all results of the same parameter combination. for the given step. If reduce=true is enabled (the default behaviour) the output of the individual runs, which uses the same parametrization, are treated like a big continuous file before applying the statistical patterns.

jube_res_analyser	jube_wp_id_first_step	jube_wp_id	jube_wp_iteration_first_step	jube_wp_iteration
analyse_no_reduce	0	6	0	0
analyse_no_reduce	0	7	0	1
analyse_no_reduce	1	8	1	0
analyse_no_reduce	1	9	1	1
analyse_no_reduce	2	10	0	2
analyse_no_reduce	2	11	0	3
analyse_no_reduce	3	12	1	0
analyse_no_reduce	3	13	1	1
analyse_no_reduce	4	14	0	2
analyse_no_reduce	4	15	0	3
analyse_no_reduce	5	16	1	0
analyse_no_reduce	5	17	1	1
analyse	5	16	1	2
analyse	0	7	0	3
analyse	1	8	1	0
analyse	2	10	0	1
analyse	3	12	1	2
analyse	4	15	0	3

Step cycle

Instead of having a new workpackage you can also redo the <do> commands inside a step using the cycle-feature.

The files used for this example can be found inside `examples/cycle`.

The input file `cycle.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<jube>
  <benchmark name="cycle" outpath="bench_run">
    <comment>A cycle example</comment>

    <step name="a_step" cycles="5">
      <do break_file="done">echo $jube_wp_cycle</do>
      <do active="$jube_wp_cycle==2">touch done</do>
    </step>

  </benchmark>
</jube>
```

The `cycles` attribute allows to repeat all `<do>` commands within a step multiple times. The `break_file` can be used to cancel the loop and all following commands in the current cycle (the command itself is still executed). In the given example the output will be:

```
0
1
2
3
```

In contrast to the iterations, all executions for the cycle feature take place inside of the same folder.

FREQUENTLY ASKED QUESTIONS

Parameter groups

Within *JUBE* you can define parameter groups to allow only specific parameter combinations.

E.g. you have are two parameters:

```
<parameter name="foo">10,100</parameter>
<parameter name="bar">20,200</parameter>
```

Without any additional change, *JUBE* will run four paramater combinations (foo=10,bar=20, foo=100,bar=20, foo=10,bar=200, foo=100,bar=200). But maybe within your configuration only foo=10,bar=20 and foo=100,bar=200 make sense. For this you can use the parameter dependencies feature and small *Python* snippets (*Parameter dependencies*) to split the four combinations into two groups, by using a dummy index value:

```
<parameter name="i">0,1</parameter>
<parameter name="foo" mode="python">[10,100][$i]</parameter>
<parameter name="bar" mode="python">[20,200][$i]</parameter>
```

Instead of using a numerical index, you can also use a string value for selction:

```
<parameter name="key">tick,tock</parameter>
<parameter name="foo" mode="python">
  { "tick" : 10,
    "tock" : 100 } [ "${key}" ]
</parameter>
<parameter name="bar" mode="python">
  { "tick" : 20,
    "tock" : 200 } [ "${key}" ]
</parameter>
```

Also default values are possible:

```
<parameter name="foo" mode="python">
  { "tick" : 10,
    "tock" : 100 }.get ( "${key}" , 0 )
</parameter>
```

Workdir change

Sometimes you want to execute a step outside of the normal *JUBE* directory structure. This can be done by using the `work_dir`-attribute inside the `<step>`-tag. If you use the `work_dir` *JUBE* does not create a unique directory structure. So you have to create this structure on your own if you need unique directories e.g. by using the *jube_variables*.

```
<step name="a_step" work_dir="path_to_dir/${jube_benchmark_papid}/${jube_wp_papid}_${jube_step_name}"
...
</step>
```

Using the `*_papid` variables will help to create a sorted directory structure.

JUBE does not create any symbolic links inside the changed work directories. If you want to access files, out of a dependend step, you can use a `<fileset>` and the `rel_path_ref`-attribute.

```
<fileset name="needed_files">
  <link rel_path_ref="internal">dependent_step_name/a_file</link>
</fileset>
```

This will create a link inside your alternative working dir and the link target path will be seen relative towards the original *JUBE* directory structure. So here you can use the normal automatic created link to access all dependend files.

To access files out of a alternative working directory in a following step and if you created this working directory by using the *jube_variables*, you can use `jube_wp_parent_<parent_name>_id` to get the id of the parent step to use it within a path definition.

XML character handling

The *JUBE* input format bases on the general *XML* rules. Here some hints for typical *XML* problems:

Linebreaks are not allowed inside a tag-option (e.g. `<sub ... dest="...\n...">` is not possible). Inside a tag multiple lines are now problem (e.g. inside of `<parameter>...</parameter>`). Often multiple lines are also needed inside a `<sub>`. Linebreaks are possible for the `dest=""` part, by switching to the alternative `<sub>` syntax:

```
<sub source="...">
...
</sub>
```

Whitespaces will only be removed in the beginning and in the end of the whole string. So indentation of a multiline string can create some problems.

Some characters are not allowed inside a *XML* script or at least not inside a tag-option. Here some of the typical replacments:

- `<: <`;
- `>: >`;
- `&: &`;
- `" : "`;
- `' : '`;

Analyse multiple output files

This FAQ entry is only relevant for *JUBE* versions prior version 2.2. Since version 2.2 *JUBE* automatically creates a combined result table.

Within an `<analyser>` you can analyse multiple files. Each `<analyser>` `<analyse>` combination will create independent result entries:

```
<analyser name="analyse">
  <use>a_patternset</use>
  <analyse step="step_A">
    <file>stdout</file>
```

```

</analyse>
<analyse step="step_B">
  <file>stdout</file>
</analyse>
</analyser>

```

In this example the `<patternset>` `a_patternset` will be used for both files. This is ok if there are only pattern which matches either the `step_A` stdout file or the `step_B` stdout file.

If you want to use a file dependent patternset you can move the use to an `<file>` attribute instead:

```

<analyser name="analyse">
  <analyse step="step_A">
    <file use="a_patternset_A">stdout</file>
  </analyse>
  <analyse step="step_B">
    <file use="a_patternset_B">stdout</file>
  </analyse>
</analyser>

```

This avoids the generation of uncorrect result entries. A `from=...` option is not available in this case. Instead you can copy the patternset first to your local file by using the `init_with` attribute.

Due to the independet result_entries, you will end up with the following result table if you mix the extracted pattern:

pattern1_of_A	pattern2_of_A	pattern1_of_B
1	A	
2	B	
		10
		11
		12
		13

The different `<analyse>` were not combined. So you end up with independet result lines for each workpackage. *JUBE* does not see possible step dependencies in this point the user has to set the dependencies manually:

```

<analyser name="analyse">
  <analyse step="step_B">
    <file use="a_patternset_B">stdout</file>
    <file use="a_patternset_A">step_A/stdout</file>
  </analyse>
</analyser>

```

Now we only habe one `<analyse>` and we using the autogenerated link to access the dependet step. This will create the correct result:

pattern1_of_A	pattern2_of_A	pattern1_of_B
1	A	10
2	B	11
1	A	12
2	B	13

COMMAND LINE DOCUMENTATION

Here you will find a list of all available *JUBE* command line options. You can also use:

```
jube -h
```

to get a list of all available commands.

Because of the *shell* parsing mechanism take care if you write your optional arguments after the command name before the positional arguments. You **must** use `--` to split the ending of an optional (if the optional argument takes multiple input elements) and the start of the positional argument.

When using *BASH* you can use the `jube complete` mechanism to enable a command line autocompletion.

general

General commandline options (can also be used in front of a subcommand)

```
jube [-h] [-V] [-v] [--debug] [--force] [--strict] [--devel] {...}
```

-h, --help show general help information

-V, --version show version information

-v, --verbose enable verbose console output (use `-vv` to show stdout during execution and `-vvv` to show log and stdout)

--debug use debugging mode (no shell script execution)

--force ignore any *JUBE* version conflict

--strict force strict *JUBE* version check

--devel developer mode (show complete error messages)

run

Run a new benchmark.

```
jube run [-h] [--only-bench ONLY_BENCH [ONLY_BENCH ...]]  
          [--not-bench NOT_BENCH [NOT_BENCH ...]] [-t TAG [TAG ...]]  
          [--hide-animation] [--include-path INCLUDE_PATH [INCLUDE_PATH ...]]  
          [-a] [-r] [-e] [-m COMMENT] [--id ID [ID ...]] FILE [FILE ...]
```

-h, --help show command help information

--only-bench ONLY_BENCH [ONLY_BENCH ...] only run specific benchmarks given by benchmark name

--not-bench NOT_BENCH [NOT_BENCH ...] do not run specific benchmarks given by benchmark name

-t TAG [TAG ...], --tag TAG [TAG ...] use specific tags when running this file. This will be used for *tagging*

--hide-animation hide the progress bar animation (if you want to use *JUBE* inside a scripting environment)

--include-path INCLUDE_PATH [INCLUDE_PATH ...] add additional include pathes where to search for include files

-a, --analyse run analyse after finishing run command

-r, --result run result after finishing run command (this will also start analyse)

-e, --exit run will exit if there is an error

-m COMMENT, --comment COMMENT overwrite benchmark specific comment

-i ID [ID ...], --id ID [ID ...] use specific benchmark id (must be ≥ 0)

FILE [FILE ...] input *XML* file

continue

Continue an existing benchmark.

```
jube continue [-h] [-i ID [ID ...]] [--hide-animation] [-a] [-r] [-e] [DIRECTORY]
```

-h, --help show command help information

-i ID [ID ...], --id ID [ID ...] select benchmark id, negative ids count backwards from the end, default: last found inside the benchmarks directory

--hide-animation hide the progress bar animation (if you want to use *JUBE* inside a scripting environment)

-a, --analyse run analyse after finishing run command

-r, --result run result after finishing run command (this will also start analyse)

-e, --exit run will exit if there is an error

DIRECTORY directory which contain benchmarks, default: .

analyse

Run the analyse procedure.

```
jube analyse [-h] [-i ID [ID ...]] [-u UPDATE_FILE]
             [--include-path INCLUDE_PATH [INCLUDE_PATH ...]]
             [-t TAG [TAG ...]] [DIRECTORY]
```

-h, --help show command help information

-i ID [ID ...], --id ID [ID ...] select benchmark id, negative ids count backwards from the end, default: last found inside the benchmarks directory

-u UPDATE_FILE, --update UPDATE_FILE use given input *XML* file to update patternsets, analyser and result before running the analyse

--include-path INCLUDE_PATH [INCLUDE_PATH ...] add additional include pathes where to search for include files (when using `--update`)

-t TAG [TAG ...], --tag TAG [TAG ...] use specific tags when running this file. This will be used for *tagging* (when using `--update`)

DIRECTORY directory which contain benchmarks, default: .

result

Run the result creation.

```
jube result [-h] [-i ID [ID ...]] [-a] [-r] [-u UPDATE_FILE] [-n NUM]
            [--include-path INCLUDE_PATH [INCLUDE_PATH ...]]
            [-t TAG [TAG ...]] [-o RESULT_NAME [RESULT_NAME ...]] [DIRECTORY]
```

-h, --help show command help information

-i ID [ID ...], --id ID [ID ...] select benchmark id, if no id is given, output of all available benchmarks will be shown

-a, --analyse run analyse before running result command

-r, --reverse reverse benchmark output order when multiple benchmarks are given

-n, --num show only last N benchmarks

-u UPDATE_FILE, --update UPDATE_FILE use given input *XML* file to update patternsets, analyser and result before running the analyse

--include-path INCLUDE_PATH [INCLUDE_PATH ...] add additional include pathes where to search for include files (when using **--update**)

-t TAG [TAG ...], --tag TAG [TAG ...] use specific tags when running this file. This will be used for *tagging* (when using **--update**)

-o RESULT_NAME [RESULT_NAME ...], --only RESULT_NAME [RESULT_NAME ...] only create specific results given by name

DIRECTORY directory which contain benchmarks, default: .

comment

Add or manipulate the benchmark comment.

```
jube comment [-h] [-i ID [ID ...]] [-a] comment [DIRECTORY]
```

-h, --help show command help information

-i ID [ID ...], --id ID [ID ...] select benchmark id, negative ids count backwards from the end, default: last found inside the benchmarks directory

-a, --append append new comment instead of overwrite existing one

comment new comment

DIRECTORY directory which contain benchmarks, default: .

remove

Remove an existing benchmark

```
jube remove [-h] [-i ID [ID ...]] [-f] [DIRECTORY]
```

-h, --help show command help information

-i ID [ID ...], --id ID [ID ...] select benchmark id, negative ids count backwards from the end, default: last found inside the benchmarks directory

-f, --force do not prompt

DIRECTORY directory which contain benchmarks, default: .

info

Get benchmark specific information

```
jube info [-h] [-i ID [ID ...]] [-s STEP [STEP ...]] [-p] [DIRECTORY]
```

-h, --help show command help information

-i ID [ID ...], --id ID [ID ...] show benchmark specific information

-s STEP [STEP ...], --step STEP [STEP ...] show step specific information

-c, --csv-parametrization display only parametrization of given step using csv format

-p, --parametrization display only parametrization of given step

DIRECTORY show directory specific information

log

Show logs for benchmark

```
jube log [-h] [-i ID [ID ...]] [-c COMMAND [COMMAND ...]] [DIRECTORY]
```

-h, --help show command help information

-i ID [ID ...], --id ID [ID ...] select benchmark id, negative ids count backwards from the end, default: last found inside the benchmarks directory

-c COMMAND [COMMAND ...], --command COMMAND [COMMAND ...] show only logs for specified commands

DIRECTORY directory which contain benchmarks, default: .

status

Show benchmark status RUNNING or FINISHED.

```
jube status [-h] [-i ID [ID ...]] [DIRECTORY]
```

-h, --help show command help information

-i ID [ID ...], --id ID [ID ...] select benchmark id, negative ids count backwards from the end, default: last found inside the benchmarks directory

DIRECTORY directory which contain benchmarks, default: .

complete

Generate shell completion. Usage: eval "\$(jube complete)"

```
jube complete [-h] [--command-name COMMAND_NAME]
```

-h, --help show command help information

--command-name COMMAND_NAME, -c COMMAND_NAME name of command to be complete, default: programname which was used to run the complete command

help

Command help

```
jube help [-h] [command]
```

-h, --help show command help information

command command to get help about

update

Check *JUBE* version

```
jube update [-h]
```

-h, --help show command help information

GLOSSARY

analyse Analyse an existing benchmark. The analyser will scan through all files given inside the configuration by using the given patternsets.

If no benchmark id is given, last benchmark found in directory will be used. If benchmark directory is missing, current directory will be used.

analyser_tag The analyser describe the steps and files which should be scanned using a set of pattern.

```
<analyser name="..." reduce="...">
  <use from="...">...</use>
  ...
  <analyse step="...">
    <file use="...">...</file>
  </analyse>
  ...
</analyser>
```

- you can use different patternsets to analyse a set of files
- only patternsets are usable
- using patternsets `<use>set1,set2</use>` is the same as `<use>set1</use><use>set2</use>`
- the from-attribute is optional and can be used to specify an external set source
- any name must be unique, it is not allowed to reuse a set
- the step-attribute contains an existing stepname
- each file using each workpackage will be scanned separately
- the use argument inside the `<file>` tag is optional and can be used to specify a file specific patternset;
 - the global `<use>` and this local use will be combined and evaluated at the same time
 - a from''subargument is not possible in this local ''use
- reduce is optional (default: true)
 - true : Combine result lines if iteration-option is used
 - false : Create single line for each iteration

benchmark_tag The main benchmark definition

```
<benchmark name="..." outpath="...">
  ...
</benchmark>
```

- container for all benchmark information
- benchmark-name must be unique inside input file
- outpath contains the path to the root folder for benchmark runs

- multiple benchmarks can use the same folder
- every benchmark and every (new) run will create a new folder (named by an unique benchmark id) inside this given `outpath`
- the path will be relative to input file location

column_tag A line within a ASCII result table. The `<column>`-tag can contain the name of a pattern or the name of a parameter.

```
<column colw="..." format="..." title="...">...</column>
```

- `colw` is optional: column width
- `title` is optional: column title
- `format` can contain a C like format string: e.g. `format="%.2f"`

comment Add or manipulate the comment string.

If no benchmark id is given, last benchmark found in directory will be used. If benchmark directory is missing, current directory will be used.

comment_tag Add a benchmark specific comment. These comment will be stored inside the benchmark directory.

```
<comment>...</comment>
```

continue Continue an existing benchmark. Not finished steps will be continued, if they are leaving pending mode.

If no benchmark id is given, last benchmark found in directory will be used. If benchmark directory is missing, current directory will be used.

copy_tag A copy can be used to copy a file or directory from your normal filesystem to your sandbox work directory.

```
<copy source_dir="..." target_dir="..." name="..." rel_path_ref="..." separator="..." active=
```

- `source_dir` is optional, will be used as a prefix for the source filenames
- `target_dir` is optional, will be used as a prefix for the target filenames
- `name` is optional, it can be used to rename the file inside your work directory (will be ignored if you use shell extensions in your pathname)
- `rel_path_ref` is optional
 - `external` or `internal` can be chosen, default: `external`
 - `external`: rel.-pathes based on position of xml-file
 - `internal`: rel.-pathes based on current work directory (e.g. to link files of another step)
- `active` is optional
 - can be set to `true` or `false` or any *Python* parsable bool expression to enable or disable the single command
 - *parameter* are allowed inside this attribute
- each copy-tag can contain a list of filenames (or directories), separated by `,`, the default separator can be changed by using the `separator` attribute
 - if `name` is present, the lists must have the same length
- you can copy all files inside a directory by using `directory/*`
 - this cannot be mixed using `name`
- in the execution step the given files or directories will be copied

directory_structure

- every (new) benchmark run will create its own directory structure
- every single workpackage will create its own directory structure
- user can add files (or links) to the workpackage dir, but the real position in filesystem will be seen as a blackbox
- general directory structure:

```
benchmark_runs (given by "outpath" in xml-file)
|
+- 000000 (determined through benchmark-id)
|
|   +- 000000_compile (step: just an example, can be arbitrary chosen)
|   |
|   |   +- work (user environment)
|   |   +- done (workpackage finished information file)
|   |   +- ... (more jube internal information files)
|   +- 000001_execute
|   |
|   |   +- work
|   |   |
|   |   |   +- compile -> ../../000000_compile/work (automatic generated link for depending
|   |   |   +- wp_done_00 (single "do" finished, but not the whole workpackage)
|   |   |   +- ...
|   +- 000002_execute
|   +- result (result data)
|   +- configuration.xml (benchmark configuration information file)
|   +- workpackages.xml (workpackage graph information file)
|   +- analyse.xml (analyse data)
+- 000001 (determined through benchmark-id)
|
|   +- 000000_compile (step: just an example, can be arbitrary chosen)
|   +- 000001_execute
|   +- 000002_postprocessing
```

do_tag A do contain a executable *Shell* operation.

```
<do stdout="..." stderr="..." active="...">...</do>
<do done_file="...">...</do>
<do break_file="...">...</do>
<do shared="true">...</do>
<do work_dir="...">...</do>
```

- do can contain any *Shell*-syntax-snippet (*parameter* will be replaced ... \$nameofparameter ...)
- stdout- and stderr-filename are optional (default: stdout and stderr)
- work_dir is optional, it can be used to change the work directory of this single command (relatively seen towards the original work directory)
- active is optional
 - can be set to true or false or any *Python* parsable bool expression to enable or disable the single command
 - *parameter* are allowed inside this attribute
- done_file-filename is optional
 - by using done_file the user can mark async-steps. The operation will stop until the script will create the named file inside the work directory.
- break_file-filename is optional
 - by using break_file the user can stop further cycle runs. the current step will be directly marked with finalized and further <do> will be ignored.

- `shared="true"`
 - can be used inside a step using a shared folder
 - cmd will be **executed inside the shared folder**
 - cmd will run once (synchronize all workpackages)
 - `$jube_wp_...` - parameter cannot be used inside the shared command

fileset_tag A fileset is a container to store a bundle of links and copy commands.

```
<fileset name="..." init_with="...">
  <link>...</link>
  <copy>...</copy>
  <prepare>...</prepare>
  ...
</fileset>
```

- `init_with` is optional
 - if the given filepath can be found inside of the `JUBE_INCLUDE_PATH` and if it contains a fileset using the given name, all link and copy will be copied to the local set
 - the name of the external set can differ to the local one by using `init-with="filename.xml:external_name"`
- link and copy can be mixed within one fileset (or left)
- filesets can be used inside the step-command

general_structure

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Basic top level JUBE structure -->
<jube>
  <!-- optional additional include pathes -->
  <include-path>
    <path>...</path>
    ...
  </include-path>
  <!-- optional benchmark selection -->
  <selection>
    <only>...</only>
    <not>...</not>
    ...
  </selection>
  <!-- global sets -->
  <parameterset name="">...</parameterset>
  <substitutionset name="">...</substitutionset>
  <fileset name="">...</fileset>
  <patternset name="">...</patternset>
  ...
  <benchmark name="" outpath="">
    <!-- optional benchmark comment -->
    <comment>...</comment>
    <!-- local benchmark parametersets -->
    <parameterset name="">...</parameterset>
    ...
    <!-- files, which should be used -->
    <fileset name="">...</fileset>
    ...
    <!-- substitution rules -->
    <substituteset name="">...</substituteset>
    ...
    <!-- pattern -->
    <patternset name="">...</patternset>
```

```

...
<!-- commands -->
<step name="">...</step>
...
<!-- analyse -->
<analyser name="">...</analyser>
...
<!-- result -->
<result>...</result>
...
</benchmark>
...
</jube>

```

include-path_tag Add some include pathes where to search for include files.

```

<include-path>
  <path>...</path>
...
</include-path>

```

- the additional path will be scanned for include files

include_tag Include *XML*-data from an external file.

```
<include from="..." path="..." />
```

- `<include>` can be used to include an external *XML*-structure into the current file
- can be used at every position (inside the `<jube>`-tag)
- path is optional and can be used to give an alternative xml-path inside the include-file (default: root-node)

info Show info for the given benchmark directory, a given benchmark or a specific step.

If benchmark directory is missing, current directory will be used.

iofile_tag A iofile declare the name (and path) of a file used for substitution.

```
<iofile in="..." out="..." out_mode="..." />
```

- `in` and `out` filepath are relative to the current work directory for every single step (not relative to the path of the inputfile)
- `in` and `out` can be the same
- `out_mode` is optional, can be `w` or `a` (default: `w`)
 - `w`: out-file will be overridden
 - `a`: out-file will be appended

jube_pattern List of available jube pattern:

- `$jube_pat_int`: integer number
- `$jube_pat_nint`: integer number, skip
- `$jube_pat_fp`: floating point number
- `$jube_pat_nfp`: floating point number, skip
- `$jube_pat_wrd`: word
- `$jube_pat_nwr`: word, skip
- `$jube_pat_bl`: blank space (variable length), skip

jube_variables List of available jube variables:

- **Benchmark:**
 - `$jube_benchmark_name`: current benchmark name
 - `$jube_benchmark_id`: current benchmark id
 - `$jube_benchmark_papid`: current benchmark id with preceding zeros
 - `$jube_benchmark_home`: original input file location
 - `$jube_benchmark_rundir`: main benchmark specific execution directory
 - `$jube_benchmark_start`: benchmark starting time
- **Step:**
 - `$jube_step_name`: current step name
 - `$jube_step_iterations`: number of step iterations (default: 1)
 - `$jube_step_cycles`: number of step cycles (default: 1)
- **Workpackage:**
 - `$jube_wp_id`: current workpackage id
 - `$jube_wp_papid`: current workpackage id with preceding zeros
 - `$jube_wp_iteration`: current iteration number (default: 0)
 - `$jube_wp_parent_<parent_name>_id`: workpackage id of selected parent step
 - `$jube_wp_relpath`: relative path to workpackage work directory (relative towards configuration file)
 - `$jube_wp_abspath`: absolute path to workpackage work directory
 - `$jube_wp_envstr`: a string containing all exported parameter in shell syntax:

```
export par=$par
export par2=$par2
```

- `$jube_wp_envlist`: list of all exported parameter names
- `$jube_wp_cycle`: id of current step cycle (starts at 0)

key_tag A syslog result key. `<key>` must contain an single parameter- or patternname.

```
<key format="..." title="...">...</key>
```

- `title` is optional: alternative key title
- `format` can contain a C like format string: e.g. `format="%.2f"`

link_tag A link can be used to create a symbolic link from your sandbox work directory to a file or directory inside your normal filesystem.

```
<link source_dir="..." target_dir="..." name="..." rel_path_ref="..." separator="..." active=
```

- `source_dir` is optional, will be used as a prefix for the source filenames
- `target_dir` is optional, will be used as a prefix for the target filenames
- `name` is optional, it can be used to rename the file inside your work directory (will be ignored if you use shell extensions in your pathname)
- `rel_path_ref` is optional
 - `external` or `internal` can be chosen, default: `external`
 - `external`: rel.-pathes based on position of xml-file
 - `internal`: rel.-pathes based on current work directory (e.g. to link files of another step)

- `active` is optional
 - can be set to `true` or `false` or any *Python* parsable bool expression to enable or disable the single command
 - *parameter* are allowed inside this attribute
- each link-tag can contain a list of filenames (or directories), separated by `,`, the default separator can be changed by using the `separator` attribute
 - if `name` is present, the lists must have the same length
- in the execution step the given files or directories will be linked

log Show logs for the given benchmark directory or a given benchmark.

If no benchmark id is given, last benchmark found in directory will be used. If benchmark directory is missing, current directory will be used.

parameter_space The parameter space for a specific benchmark run is the bundle of all possible parameter combinations. E.g. there are two different parameters: `a = 1,2` and `b= "p","q"` then you will get four different parameter combinations: `a=1, b="p"`; `a=1, b="q"`; `a=2, b="p"`; `a=2, b="q"`.

The parameter space of a specific step will be one of these parameter combinations. To fulfill all combinations the step will be executed multiple times (each time using a new combination). The specific combination of a step and an expanded parameter space is named *workpackage*.

parameter_tag A parameter can be used to store benchmark configuration data. A set of different parameters will create a specific parameter environment (also called *parameter space*) for the different steps of the benchmark.

```
<parameter name="..." mode="..." type="..." separator="..." export="..." update_mode="...">...
```

- a parameter can be seen as variable: Name is the name to use the variable, and the text between the tags will be the real content
- name must be unique inside the given parameterset
- type is optional (only used for sorting, default: `string`)
- mode is optional (used for script-types, default: `text`)
- separator is optional, default: `,`
- export is optional, if set to `true` the parameter will be exported to the shell environment when using `<do>`
- if the text contains the given (or the implicit) separator, a template will be created
- use of another parameter:
 - inside the parameter definition, a parameter can be reused: `... $nameofparameter ...`
 - the parameter will be replaced multiple times (to handle complex parameter structures; max: 5 times)
 - the substitution will be run before the execution step starts with the current *parameter space*. Only parameters reachable in this step will be usable for substitution!
- Scripting modes allowed:
 - `mode="python"`: allow *Python* snippets (using `eval <cmd>`)
 - `mode="perl"`: allow *Perl* snippets (using `perl -e "print <cmd>"`)
 - `mode="shell"`: allow *Shell* snippets
- Templates can be created, using scripting e.g.: `" , ".join([str(2**i) for i in range(3)])`
- `update_mode` is optional (default: `never`)

- can be set to never, use, step and cycle
- depending on the setting the parameter will be reevaluated:
 - * never: no reevaluation, even if the parameterset is used multiple times
 - * use: reevaluation if the parameterset is explicitly used
 - * step: reevaluation in each new step
 - * cycle: reevaluation in each cycle (number of workpackages will stay unchanged)
 - * always: reevaluation in each step and cycle

parameterset_tag A parameterset is a container to store a bundle of *parameters*.

```
<parameterset name="..." init_with="...">
  <parameter>...</parameter>
  ...
</parameterset>
```

- parameterset-name must be unique (cannot be reused inside substitutionsets or filesets)
- init_with is optional
 - if the given filepath can be found inside of the JUBE_INCLUDE_PATH and if it contains a parameterset using the given name, all parameters will be copied to the local set
 - local parameters will overwrite imported parameters
 - the name of the external set can differ to the local one by using init-with="filename.xml:external_name"
- parametersets can be used inside the step-command
- parametersets can be combined inside the step-tag, but they must be compatible:
 - Two parametersets are compatible if the parameter intersection (given by the parameter-name), only contains parameter based on the same definition
 - These two sets are compatible:

```
<parameterset name="set1">
  <parameter name="test">1,2,4</parameter>
  <parameter name="test2">foo</parameter>
</parameterset>
<parameterset name="set2">
  <parameter name="test">1,2,4</parameter>
  <parameter name="test3">bar</parameter>
</parameterset>
```

- These two sets are not compatible:

```
<parameterset name="set1">
  <parameter name="test">1,2,4</parameter>
  <parameter name="test2">foo</parameter>
</parameterset>
<parameterset name="set2">
  <parameter name="test">2</parameter> <!-- Template in set1 -->
  <parameter name="test2">bar</parameter> <!-- Other content in set2 -->
</parameterset>
```

pattern_tag A pattern is used to parse your output files and create your result data.

```
<pattern name="..." default="..." unit="..." mode="..." type="...">...</pattern>
```

- unit is optional, will be used in the result table
- mode is optional, allowed modes:

- pattern: a regular expression (default)
- text: simple text and variable concatenation
- perl: snippet evaluation (using *Perl*)
- python: snippet evaluation (using *Python*)
- shell: snippet evaluation (using *Shell*)
- type is optional, specify datatype (for sort operation)
 - default: string
 - allowed: int, float or string
- default is optional: Specify default value if pattern cannot be found or if it cannot be evaluated

patternset_tag A patternset is a container to store a bundle of patterns.

```
<patternset name="..." init_with="...">
  <pattern>...</pattern>
  ...
</patternset>
```

- patternset-name must be unique
- init_with is optional
 - if the given filepath can be found inside of the JUBE_INCLUDE_PATH and if it contains a patternset using the given name, all pattern will be copied to the local set
 - local pattern will overwrite imported pattern
 - the name of the external set can differ to the local one by using init-with="filename.xml:external_name"
- patternsets can be used inside the analyser tag
- different sets, which are used inside the same analyser, must be compatible

prepare_tag The prepare can contain any *Shell* command you want. It will be executed like a normal `<do>` inside the step where the corresponding fileset is used. The only difference towards the normal do is, that it will be executed **before** the substitution will be executed.

```
<prepare stdout="..." stderr="..." work_dir="..." active="...">...</prepare>
```

- stdout- and stderr-filename are optional (default: stdout and stderr)
- work_dir is optional, it can be used to change the work directory of this single command (relatively seen towards the original work directory)
- active is optional
 - can be set to true or false or any *Python* parsable bool expression to enable or disable the single command
 - *parameter* are allowed inside this attribute

remove The given benchmark will be removed.

If no benchmark id is given, last benchmark found in directory will be removed.

Only the *JUBE* internal directory structure will be deleted. External files and directories will stay unchanged.

If no benchmark id is given, last benchmark found in directory will be used. If benchmark directory is missing, current directory will be used.

result Create a result table.

If no benchmark id is given, a combined result view of all available benchmarks in given directory will be created. If benchmark directory is missing, current directory will be used.

result_tag The result tag is used to handle different visualisation types of your analysed data.

```
<result result_dir="...">
  <use>...</use>
  ...
  <table>...</table>
  <syslog>...</syslog>
  ...
</result>
```

- `result_dir` is optional. Here you can specify an different output directory. Inside of this directory a subfolder named by the current benchmark id will be created. Default: `benchmark_dir/result`
- only analyser are usable
- using analyser `<use>set1,set2</use>` is the same as `<use>set1</use><use>set2</use>`

run Start a new benchmark run by parsing the given *JUBE* input file.

selection_tag Select benchmarks by name.

```
<selection>
  <only>...</only>
  <not>...</not>
  ...
</selection>
```

- select or unselect a benchmark by name
- only selected benchmarks will run (when using the `run` command)
- multiple `<only>` and `<not>` are allowed
- `<only>` and `<not>` can contain a name list divided by ,

statistical_values If there are multiple pattern matches within one file, multiple files or when using multiple iterations. *JUBE* will create some statistical values automatically:

- `first`: first match (default)
- `last`: last match
- `min`: min value
- `max`: max value
- `avg`: average value
- `std`: standard deviation
- `sum`: sum
- `cnt`: counter

These variabels can be accessed within the the result creation or to create derived pattern by `variable_name_<statistic_option>` e.g. `${nodes_min}`

The variable name itself always matches the first match.

status Show status string (RUNNING or FINISHED) for the given benchmark.

If no benchmark id is given, last benchmark found in directory will be used. If benchmark directory is missing, current directory will be used.

step_tag A step give a list of *Shell* operations and a corresponding parameter environment.

```
<step name="..." depend="..." work_dir="..." suffix="..." shared="..." active="..."
  export="..." max_async="..." iterations="..." cycles="...">
  <use from="">...</use>
  ...
```

```
<do></do>
...
</step>
```

- parametersets, filesets and substitutionsets are usable
- using sets `<use>set1, set2</use>` is the same as `<use>set1</use><use>set2</use>`
- parameter can be used inside the `<use>`-tag
- the `from` attribute is optional and can be used to specify an external set source
- any name must be unique, it is **not allowed to reuse** a set
- `depend` is optional and can contain a list of other step names which must be executed before the current step
- `max_async` is optional and can contain a number (or a parameter) which describe how many *work-packages* can be executed asynchronously (default: 0 means no limitation). This option is only important if a *do* inside the step contains a `done_file` attribute and should be executed in the background (or managed by a jobsystem). In this case *JUBE* will manage that there will not be too many instances at the same time. To update the benchmark and start further instances, if the first ones were finished, the *continue* command must be used.
- `work_dir` is optional and can be used to switch to an alternative work directory
 - the user has to handle **uniqueness of this directory** by his own
 - no automatic parent/children link creation
- `suffix` is optional and can contain a string (parameters are allowed) which will be attached to the default workpackage directory name
- `active` is optional
 - can be set to `true` or `false` or any *Python* parsable bool expression to enable or disable the single command
 - *parameter* are allowed inside this attribute
- `shared` is optional and can be used to create a shared folder which can be accessed by all workpackages based on this step
 - a link, named by the attribute content, is used to access the shared folder
 - the shared folder link will not be automatically created in an alternative working directory!
- `export="true"`
 - the environment of the current step will be exported to an dependent step
- `iterations` is optional. All workpackages within this step will be executed multiple times if the `iterations` value is used.
- `cycles` is optional. All `<do>` commands within the step will be executed `cycles`-times

sub_tag A substitution expression.

```
<sub source="..." dest="..." />
```

- source-string will be replaced by dest-string
- both can contain parameter: `... $nameofparameter ...`

substituteset_tag A substituteset is a container to store a bundle of *sub* commands.

```
<substituteset name="..." init_with="...">
  <iofile/>
  ...
</sub/>
```

```
...
</substituteset>
```

- `init_with` is optional
 - if the given filepath can be found inside of the `JUBE_INCLUDE_PATH` and if it contains a `substituteset` using the given name, all `iofile` and `sub` will be copied to the local set
 - local `iofile` will overwrite imported ones based on `out`, local `sub` will overwrite imported ones based on `source`
 - the name of the external set can differ to the local one by using `init-with="filename.xml:external_name"`
- `substitutesets` can be used inside the `step-command`

syslog_tag A syslog result type

```
<syslog name="..." address="..." host="..." port="..." sort="..." format="..." filter="...">
  <key>...</key>
  ...
</syslog>
```

- Syslog daemon can be given by a `host` and `port` combination (default `port`: 541) or by a socket address e.g.: `/dev/log` (mixing of `host` and `address` is not allowed)
- `format` is optional: can contain a log format written in a pythonic way (default: `jube[% (process) s]: % (message) s`)
- `sort` is optional: can contain a list of parameter- or patternnames (separated by `.`). Given `pattern`type or `parameter`type will be used for sorting
- `<key>` must contain an single parameter- or patternname
- `filter` is optional, it can contain a bool expression to show only specific result entries

table_tag A simple ASCII based table output.

```
<table name="..." style="..." sort="..." separator="..." transpose="..." filter="...">
  <column>...</column>
  ...
</table>
```

- `style` is optional; allowed styles: `csv`, `pretty`; default: `csv`
- `separator` is optional; only used in `csv`-style, default: `,`
- `sort` is optional: can contain a list of parameter- or patternnames (separated by `.`). Given `pattern`type or `parameter`type will be used for sorting
- `<column>` must contain an single parameter- or patternname
- `transpose` is optional (default: `false`)
- `filter` is optional, it can contain a bool expression to show only specific result entries

tagging Tagging is a simple way to mark parts of your input file to be includable or excludable.

- Every available `<tag>` (not the root `<jube>`-tag) can contain a tag-attribute
- The tag-attribute can contain a list of names: `tag="a, b, c"` or “not” names: `tag="a, !b, c"`
- When running *JUBE*, multiple tags can be send to the input-file parser:

```
jube run <filename> --tag a b
```

- `<tags>` which does not contain one of these names will be hidden inside the include file
- `<tags>` which does not contain any tag-attribute will stay inside the include file

- “not” tags are more important than normal tags: `tag="a, !b, c"` and running with `a b` will hide the `<tag>` because the `!b` is more important than the `a`

types *Parameter* and *Pattern* allow a type specification. This type is either used for sorting within the result table and is also used to validate the parameter content. The types are not used to convert parameter values, e.g. a floating value will stay unchanged when used in any other context even if the type `int` was specified.

allowed types are:

- `string` (this is also the default type)
- `int`
- `float`

update Check if a newer JUBE version is available.

update_mode The update mode is parameter attribute which can be used to control the reevaluation of the parameter content.

These update modes are available:

- `never`: no reevaluation, even if the parameterset is used multiple times
- `use`: reevaluation if the parameterset is explicitly used
- `step`: reevaluation in each new step
- `cycle`: reevaluation in each cycle (number of workpackages will stay unchanged)
- `always`: reevaluation in each step and cycle

workpackage A workpackage is the combination of a *step* (which contains all operations) and one parameter setting out of the expanded *parameter space*.

Every workpackage will run inside its own sandbox directory!

A

advanced tutorial, 11
 analyse, 9, 34, 39
 analyse multiple files, 30
 analyser_tag, 39

B

benchmark_tag, 39

C

column_tag, 40
 commandline, 31
 comment, 35, 40
 comment_tag, 40
 complete, 36
 configuration, 3
 continue, 34, 40
 copy_tag, 40
 cycle, 27

D

dependencies, 7
 directory_structure, 40
 do_tag, 41
 dtd, 13

E

environment handling, 24
 external files, 8

F

faq, 28
 files, 8
 fileset_tag, 42

G

general commandline options, 33
 general_structure, 42

H

hello world, 4
 help, 6, 36

I

include, 19

include-path_tag, 43
 include_tag, 43
 info, 35, 43
 installation, 3
 introduction, 1
 iofile_tag, 43
 iteration, 26

J

jobssystem, 17
 jube_pattern, 43
 jube_variables, 43

K

key_tag, 44

L

link_tag, 44
 loading files, 8
 log, 36, 45
 logging, 6

M

multiple benchmarks, 22

P

parameter dependencies, 24
 parameter groups, 29
 parameter space creation, 6
 parameter update, 25
 parameter_space, 45
 parameter_tag, 45
 parameterset_tag, 46
 pattern, 15
 pattern_tag, 46
 patternset_tag, 47
 perl, 14, 15
 platform independent, 21
 prepare_tag, 47
 python, 14, 15

R

remove, 35, 47
 result, 9, 34, 47
 result_tag, 48
 run, 33, 48

S

- schema validation, 13
- scripting, 14, 15
- selection_tag, 48
- shared operations, 23
- shell, 14, 15
- statistic values, 16
- statistical_values, 48
- status, 36, 48
- step dependencies, 7
- step_tag, 48
- sub_tag, 49
- substituteset_tag, 49
- substitution, 8
- syslog_tag, 50

T

- table, 9
- table_tag, 50
- tagging, 21, 50
- tutorial, 1
- types, 51

U

- update, 37, 51
- update_mode, 51

V

- validation, 13

W

- workdir change, 29
- workpackage, 51

X

- XML character handling, 30