JUBE Documentation

Release 2.0.1

2014, JUBE Developer Team,
Forschungszentrum Jülich GmbH

November 25, 2014

CHAPTER

ONE

JUBE TUTORIAL

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

1.1 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 `PHYTONPATH` must contain the position of the *JUBE* package

- 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.

- You can also add **parent folder path** of the *JUBE* package-folder to the `PHYTONPATH` environment variable:

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

- 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
```

1.2 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="parameterset">
      <parameter name="hello_str">Hello World</parameter>
    </parameterset>

    <!-- Operation -->
```

```
    <step name="say_hello">
      <use>parameterset</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 using 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 doesn't 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 comment 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">` parameter. 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. 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 parameterspace is named *workpackage*.

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

```
$parametername
```

or

```
${parametername}
```

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

To run the benchmark just type:

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

This benchmark will produce the follwing output:

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

A simple hello world
################################################################################

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

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

```
>>>> Benchmark information and further useful commands:
>>>>       id: 0
>>>>      dir: bench_run
>>>>  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 run one shell command `echo $hello_str` which will be expanded to `echo Hello World`.

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

Inside the benchmark directory you will see the follwing 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 sanbox 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.

1.3 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:

```
# display a list of existing benchmarks
>>> jube info <benchmark-directory>
# display information about given benchmark
>>> jube info <benchmark-directory> -- id <id>
# display information about a step inside the given benchmark
>>> jube info <benchmark-directory> -- id <id> --step <stepname>
```

The third, but 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`.

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

There is also a debugging mode inside of *JUBE*:

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

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

1.4 Parameterspace creation

In this example we will show you an important feature of *JUBE*: The automatic parameterspace 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="parameterset">
      <!-- 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>parameterset</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 possible parameter combinations. In this example the step `say_hello` will run 6 times:

```
 stepname | all | open | wait | done
----------+-----+------+------+-----
say_hello |   6 |    0 |    0 |    6
```

Every parameter combination will run in its own sandbox directory.

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

1.5 Step dependencies

If you start writing a complex benchmark structure, you 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 parameterspace.

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

    <!-- Operations -->
    <step name="first_step">
      <use>parameterset</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. You are able to 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 a different `first_step`-directory:

```
   stepname | all | open | wait | done
------------+-----+------+------+-----
 first_step |   3 |    0 |    0 |    3
second_step |   3 |    0 |    0 |    3
```

1.6 Loading files and substitution

Every step runs inside a unique sandbox directory. In normal cases you need 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 parameterspace. 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="parameterset">
      <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>parameterset</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 (relativly seen towards the current input file, also absolute pathes are allowed) of files is given. `<copy>` specify 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 filename. The path is relativly seen towards the sandbox directory. Because we do/should not know that location we used the fileset to copy `file.in` to this directory.

The `<sub>` specify the substitution. All occurrence of `source` will be substituted by `dest`. As you can see, you can use Parameter 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 use the corresponding `<parameterset>` inside the `<step>` where you use the `<substituteset>`!

In the `sub_step` we use all available sets. The use order isn't relevant. The normal execution process will be:

1. Parameterspace 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 ($number = 1)
      |
      +- done           # workpackage finished marker
      +- work           # user sanbox folder
         |
         +- stderr      # standard error messages of used shell commands
         +- stdout      # standard output of used shell commands (Number: 1)
         +- file.in     # the file copy
         +- file.out    # the substituted file
   +- 000001_say_hello  # the workpackage ($number = 2)
      |
      +- ...
   +- ...
```

1.7 Creating a result table

Finally, after running the benchmark, you will get several directories. *JUBE* allows you to parse your result files 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="parameterset">
      <!-- Create a parameterspace out of two 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>parameterset</use> <!-- use existing parameterset -->
      <do>echo "Number: $number"</do> <!-- shell command -->
    </step>

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

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

```
39    </benchmark>
40  </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 had 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 pattern or parameter. The example uses `$jube_pat_int` which is a *JUBE* given *default pattern* matching integer values. The pattern must contain a group, given by brackets `(...)`, to declare the extraction part (`$jube_pat_int` already contains these brackets).

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

To use your <patternset> you had to specify the files which should be parsed. This can be done using the <analyzer>. It uses relevant patternsets. 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 analyzer automatically knows all parameter which where used in the given step and in depending steps. There is no <use> option to add additonal completely new parametersets.

To run the anlayse you had 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 had to used an existing analyzer. 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 had to write:

```
>>> jube result bench_run
```

The result table will be written to `STDOUT` and into a `result.dat` file inside `bench_run/<id>/result`.

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, jobsystem representation and scripting parameter.

CHAPTER

TWO

ADVANCED TUTORIAL

This tutorial will show you more detailed functions and tools of *JUBE*. If you want a basic overview you should read the general *JUBE tutorial* first.

2.1 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:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE benchmarks SYSTEM "<jube.dtd path>">
<benchmark>
...
```

Schema usage:

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

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

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

2.2 Scripting parameter

Sometimes you want to create a parameter which based on the value of another paramter. In this case you can use 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" outpath="bench_run">
    <comment>A scripting parameter example</comment>

    <!-- Configuration -->
    <parameterset name="parameterset">
      <!-- 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*${number}) for a
      <!-- A scripting parameter -->
      <parameter name="number_mult" mode="python" type="float">${number}*${additional_number}</pa
      <!-- Normal template -->
      <parameter name="text">Number: $number</parameter>
    </parameterset>

    <!-- Operation -->
    <step name="operation">
      <use>parameterset</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 parameter.

- `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` and `perl` are allowed). The additional `type` is optional and declare 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 parameter. Because of this second template we will get six different *workpackages*.
- `number_mult` is a small calculation. You can use any other existing parameter (which is used inside the same step).
- `text` a normal parameter which uses the content of another parameter. For simple concatenation parameter you do not need 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 had to create a new parameter.

2.3 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 jobscript):

```
#!/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="parameterset">
      <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>
```

```
21        <parameter name="err_file">stderr</parameter>
22        <parameter name="out_file">stdout</parameter>
23        <parameter name="exec">echo $number</parameter>
24      </parameterset>
25
26      <!-- Load jobfile -->
27      <fileset name="files">
28        <copy>${job_file}.in</copy>
29      </fileset>
30
31      <!-- Substitute jobfile -->
32      <substituteset name="sub_job">
33        <iofile in="${job_file}.in" out="$job_file" />
34        <sub source="#NODES#" dest="$nodes" />
35        <sub source="#PROCS_PER_NODE#" dest="$ppn" />
36        <sub source="#WALLTIME#" dest="$walltime" />
37        <sub source="#ERROR_FILEPATH#" dest="$err_file" />
38        <sub source="#OUT_FILEPATH#" dest="$out_file" />
39        <sub source="#MAIL_ADDRESS#" dest="$mail_address" />
40        <sub source="#MAIL_MODE#" dest="$mail_mode" />
41        <sub source="#EXEC#" dest="$exec" />
42        <sub source="#READY#" dest="$ready_file" />
43      </substituteset>
44
45      <!-- Operation -->
46      <step name="submit" work_dir="$$WORK/jobsystem_bench_${jube_benchmark_id}_${jube_wp_id}" >
47        <use>parameterset</use>
48        <use>executeset</use>
49        <use>files,sub_job</use>
50        <do done_file="$ready_file">$submit_cmd $job_file</do> <!-- shell command -->
51      </step>
52    </benchmark>
53  </jube>
```

As you can see the jobfile is very general and several parameter 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 set a filename/path to a file which should be used by the jobfile to mark the end of execution. *JUBE* doesn't 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 benchmark_run
  ```

  This will continue your benchmark execution (`benchmark_run` is the benchmarks directory in this example). The position of the `done_file` is relativly 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 get 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 parameter* and will expand to the current benchmark and workpackage id. Files and directories out of a given `<fileset>` will be copied to 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 | done
---------+-----+------+------+-----
```

```
  submit |   3 |    0 |    3 |    0
```

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

```
stepname | all | open | wait | done
---------+-----+------+------+-----
  submit |   3 |    0 |    0 |    3
```

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

2.4 Include external data

As you 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 inside of *JUBE*.

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

  <parameterset name="parameterset2">
    <parameter name="text">Hello</parameter>
  </parameterset>

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

All files which contains data that should 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="parameterset" init_with="include_data.xml">
      <parameter name="foo">bar</parameter>
    </parameterset>

    <!-- Operation -->
    <step name="say_hello">
      <use>parameterset</use> <!-- use existing parameterset -->
      <use from="include_data.xml">parameterset2</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 inlcude types.

The `init_with` can be used inside any set definition. Inside the given file the searching 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 can overwrite existing ones. If your include-set use 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 setname 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 `path` is optional and can be used to select a specific nodeset (otherwise the root-node 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 pathnames, 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 the same directory of your `main.xml`
- inside a directory given over the command line:

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

- inside any path given with the `JUBE_INCLUDE_PATH` environment variable:

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

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

```
<?xml version="1.0" encoding="UTF-8"?>
<benchmarks>
  <include-path>
    <path>some_path</path>
    <path>another_path</path>
  </include-path>
  ...
```

2.5 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="parameterset">
      <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>parameterset</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>).

2.6 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 includeable 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 inlcude-path usage

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

To avoid writting long include-pathes 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:

```
<?xml version="1.0" encoding="UTF-8"?>
<jube>
  <include-path>
    <path tag="plat1">some path</path>
    <path tag="plat2">another path</path>
    ...
  </include-path>
```

```
  ...
</jube>
```

Now you can run your benchmark using:

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

2.7 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:

```
<?xml version="1.0" encoding="UTF-8"?>
<jube>
  <benchmark name="a" outpath="bench_runs">...</benchmark>
  <benchmark name="b" outpath="bench_runs">...</benchmark>
  ...
</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:

```
<?xml version="1.0" encoding="UTF-8"?>
<jube>
  <selection>
    <only>a</only>
    <not>b</not>
  </selection>
  ...
</jube>
```

2.8 Shared operations

Sometimes you want to communicate between the different workpackages of a single step or you want a single operation only run one time 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="parameterset">
      <parameter name="number" type="int">1,2,4</parameter>
    </parameterset>
```

```
10
11      <!-- Operation -->
12      <step name="a_step" shared="shared">
13        <use>parameterset</use>
14        <!-- shell command will run three times -->
15        <do>echo $jube_wp_id >> shared/all_ids</do>
16        <!-- shell command will run one time -->
17        <do shared="true">cat all_ids</do>
18      </step>
19    </benchmark>
20  </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 sanbox 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
```

2.9 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`:

```
1   <?xml version="1.0" encoding="UTF-8"?>
2   <jube>
3     <benchmark name="environment" outpath="bench_run">
4       <comment>An environment handling example</comment>
5
6       <!-- Configuration -->
7       <parameterset name="parameterset">
8         <parameter name="EXPORT_ME" export="true">VALUE</parameter>
9       </parameterset>
```

```
10
11      <!-- Operations -->
12      <step name="first_step" export="true">
13        <do>export SHELL_VAR=Hello</do> <!-- export a Shell var -->
14        <do>echo "$$SHELL_VAR world"</do><!-- use exported Shell var -->
15      </step>
16
17      <!-- Create a dependency between both steps -->
18      <step name="second_step" depend="first_step">
19        <use>parameterset</use>
20        <do>echo $$EXPORT_ME</do>
21        <do>echo "$$SHELL_VAR again"</do> <!-- use exported Shell var out of privious step -->
22      </step>
23    </benchmark>
24  </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 additonal 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>.

2.10 Convert option

For the *JUBE* version 1 file conversion *JUBE* seeks in specific directories for the files which need to be converted. For instance, the default directory for platform.xml is relative to the *JUBE* version 1 benchmark directory, i.e. jube seeks in ../../..platform/ for this file. If it can't be found *JUBE* tries to find it in the current directory.

Generally, if the convertion fails because files can't be found it is recommend to copy them to the current directory. jube creates two files, namely benchmarks_jube2.xml and platform_jube2.xml. Have a look in benchmarks_jube2.xml and take note of the given comments.

CHAPTER

THREE

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.

3.1 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] [-m COMMENT] 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)

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

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

3.2 convert

Convert jube version 1 files to jube version 2 files.

```
jube convert [-h] [-i INPUT_PATH] main_xml_file
```

-h, --help show command help information

-i INPUT_PATH main_xml_file select root directory of jube version 1 benchmark along with the corresponding main XML file

3.3 continue

Continue an existing benchmark.

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

-h, --help show command help information

-i ID [ID ...], --id ID [ID ...] select benchmark id, 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)

DIRECTORY directory which contain benchmarks, default: **.**

3.4 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, default: last found inside the benchmarks directory

-u UPDATE_FILE, --update UPDATE_FILE use given input *XML* file to update `patternsets`, `analyzer` 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: **.**

3.5 result

Run the result creation.

```
jube result [-h] [-i ID [ID ...]] [-a] [-u UPDATE_FILE]
            [--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, default: last found inside the benchmarks directory

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

-u UPDATE_FILE, --update UPDATE_FILE use given input *XML* file to update `patternsets`, `analyzer` 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: **.**

3.6 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, 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: **.**

3.7 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, default: last found inside the benchmarks directory

-f, --force do not prompt

DIRECTORY directory which contain benchmarks, default: **.**

3.8 info

Get benchmark specific information

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

DIRECTORY show directory specific information

3.9 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, 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: .

..index:: status

3.10 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, default: last found inside the benchmarks directory

DIRECTORY directory which contain benchmarks, default: .

3.11 help

Command help

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

-h, --help show command help information

command command to get help about

CHAPTER

FOUR

GLOSSARY

analyse Analyse an existing benchmark. The analyzer 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.

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

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

- you can use different patternsets to analyse a set of files
- only patternsets are useable
- 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 seperatly

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

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.

convert Convert jube version 1 files to jube version 2 files.

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

```
<copy directory="..." name="..." rel_path_ref="..." separator="...">...</copy>
```

- `directory` is optional, it can be used if you want to copy several files inside the same directory
- `name` is optional, it can be used to rename the file inside your work directory
- `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)
- 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 can't 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
```

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"?>
<jube>
  <!-- optional additional include pathes -->
  <include-path>
    <path>...</path>
    ...
  </include-path>
  <!-- optional benchmark selection -->
  <selection>
    <only>...</only>
    <not>...</not>
    ...
  </selection>
  <!-- global sets -->
  <parameterset>...</parameterset>
  <substitutionset>...</substitutionset>
  <fileset>...</fileset>
  <patternset>...</patternset>
  ...
  <benchmark>
    <!-- optional benchmark comment -->
    <comment>...</comment>
    <!-- local benchmark parametersets -->
    <parameterset>...</parameterset>
    ...
    <!-- files, which should be used -->
    <fileset>...</fileset>
    ...
    <!-- substitution rules -->
    <substituteset>...</substituteset>
    ...
    <!-- pattern -->
    <patternset>...</patternset>
```

```
    ...
    <!-- commands -->
    <step>...</step>
    ...
    <!-- analyse -->
    <analyzer>...</analyzer>
    ...
    <!-- 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="..." />
```

- `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` must be different

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_nwrd`: 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

  - `$jube_benchmark_home`: original input file location
- Step:
  - `$jube_step_name`: current step name
  - `$jube_step_iteratuions`: number of step iterations (default: 1)
- Workpackage:
  - `$jube_wp_id`: current workpackage id
  - `$jube_wp_iteration`: current iteration number (default: 0)
  - `$jube_wp_parent_<parent_name>_id`: workpackage id of selected parent step
  - `$jube_wp_abspath`: absolute path to workpackage work directory
  - `$jube_wp_envstr`: a string containing all exported parameter in shell syntax:

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

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 directory="..." name="..." rel_path_ref="..." separator="...">...</link>
```

- `directory` is optional, it can be used if you want to link several files inside the same directory
- `name` is optional, it can be used to rename the file inside your work directory
- `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)
- 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_tag A parameter is a usable configuration option.

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

- 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 multiply 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 useable for substitution!
- Scripting modes allowed:
  - `mode="python"`: allow python snippets (using `eval <cmd>`)
  - `mode="perl"`: allow perl snippets (using `perl -e "print <cmd>"`)
- Templates can be created, using scripting e.g.: `",".join([str(2**i) for i in range(3)])`

parameterset_tag A parameterset is a container to store a bundle of parameter.

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

- parameterset-name must be unique (can't be reuse 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 aren't 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="..." unit="..." mode="..." type="..." reduce="...">...</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*)
- `type` is optional, specify datatype (for sort operation)
  - default: `string`
  - allowed: `int`, `float` or `string`
- `reduce` is optional, specify handling of multiple matches
  - `first`: use first match (default)
  - `last`: use last match
  - `min`: use min (using float casting)
  - `max`: use max (using float casting)
  - `avg`: use avg (using float casting)
  - `sum`: use sum (using float casting)
  - `cnd`: use counter
  - `all`: use all of the options
  - reduce can contain a list
  - reduced variables can be used by name_<reduce_option>

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

```
<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 analyzer-command
- different sets, which are used inside the same analyzer, 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 coresspoding 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="...">...</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 (relativly seen towards the original work directory)

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, last benchmark found in directory will be used. If benchmark directory is missing, current directory will be used.

result_tag Container for different output types.

```
<result result_dir="...">
  <use>...</use>
  ...
  <table>...</table>
  ...
</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 analyzer are useable
- using analyzer `<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 `,`

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="..." work_dir="..." shared="..." export="...">
  <use from="">...</use>
  ...
  <do stdout="..." stderr="..." active="...">...</do>
  <do done_file="...">...</do>
  <do shared="true">...</do>
  <do work_dir="...">...</do>
  ...
</step>
```

- parametersets, filesets and substitutionsets are useable
- using filesets and substitutesets `<use>set1,set2</use>` is the same as `<use>set1</use><use>set2</use>`
- using parametersets `<use>set1</use><use>set2</use>` means: use both; `<use>set1,set2</use>` means: use in one case the first set and in second case the other set
- 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
- `work_dir` is optional and can be used to switch to an alternative work directory
  - the user had to handle **uniqueness of this directory** by his own
  - no automatic parent/children link creation
- `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!
- `do` can contain any *Shell*-syntax-snippet (parameter will be replaced ... $nameofparameter ...)
- `stdout`- and `stderr`-filename are optional (default: `stdout` and `stderr`)
- `work_dir` is optional, it can be used to change the work directory of this single command (relativly seen towards the original work directory)
- `active` is optional
  - can be set to `true` or `false` to enable or disable the single command
  - parameter are allowed inside this attribute
- `done_file`-filename is optional
  - by using done_file the user can mark async-steps. The operation will stop until the script will create the named file inside the work directory.
- `shared="true"`
  - can be used inside a step using a shared folder
  - cmd will be **executed inside the shared folder**
  - cmd will run once (synchronize all workpackages)
  - `$jube_wp_...` - parameter can't be used inside the shared command
- `export="true"`
  - the environment of the current step will be exported to an dependent step

sub_tag A substition 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 subs.

```
<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

table_tag A simple ASCII based table ouput.

```
<table name="..." style="..." sort="..." separator="...">
  <column colw="..." format="..." title="...">...</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 patterntype or parametertype will be used for sorting
- `<column>` must contain an single parameter- or patternname
- `colw` is optional: column width
- `title` is optional: column title
- `format` can contain a C like format string: e.g. format=".2f"

tagging Tagging is a simple way to include or exclude parts of your input file.

- 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 doesn't contain one of these names will be hidden inside the include file
  - <tags> which doesn't 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`

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

Every workpackage will run inside its own sandbox directory!

V

W