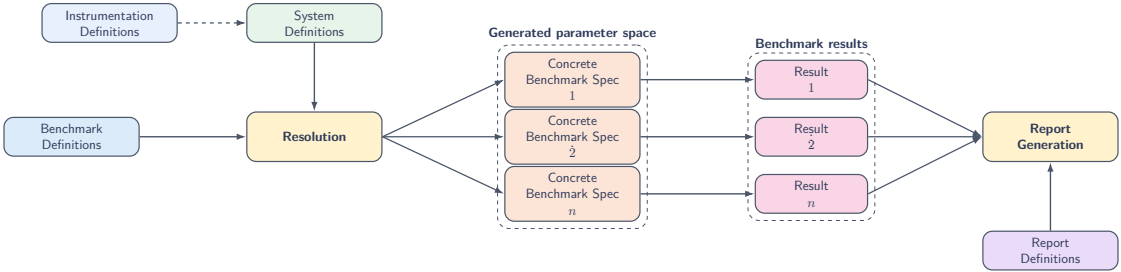
UNIFIED BENCHMARK DESCRIPTIONS

An Initial and Partial Approach

July 6, 2026 | Jayesh Badwaik, Andreas Herten | JSC

JÜLICH Forschungszentrum

STRUCTURE OF BENCHMARKS



JÜLICH Forschungszentrum

THREE LAYERS OF SPECIFICATION

1. **Benchmark specification**
   - application parameters,
   - software parameters,
   - machine parameters.
2. **Generated benchmark space**
   - concrete benchmark specs,
   - cases, variants, node counts, inputs, repetitions.
3. **Resolution specification**
   - rules that generate the fanout,
   - parameter spaces,
   - system- and framework-dependent expansion.

LAYER 1: BENCHMARK SPECIFICATIONS

JUBE, YAML-like

```yaml
benchmark: app-case-single
application: app
case: case
experiment: single

system: jurecadc
partition: dc-gpu

parameters:
  nodes: 1
  taskspernode: 4
  threadspertask: 1
  input: case.yaml

execution:
  launcher: srun
  executable: ./app
  arguments:
    - --input
    - $input
```

ReFrame

```python
class
↪  AppCaseSingle(rfm.RegressionTest)
    descr = 'app case single'

    valid_systems =
    ↪  ['jurecadc:dc-gpu']
    valid_prog_environs =
    ↪  ['builtin']

    num_nodes = 1
    num_tasks_per_node = 4
    num_cpus_per_task = 1

    input = 'case.yaml'

    executable = 'srun'
    executable_opts = [
        './app',
        '--input', input
    ]
```

Ramble workspace

```yaml
benchmark: app-case-single
application: app
workload: case
experiment: single

system: jurecadc
partition: dc-gpu

variables:
  n_nodes: 1
  processes_per_node: 4
  threads_per_process: 1
  input: case.yaml

execution:
  launcher: srun
  executable: ./app
  arguments:
    - --input
    - "{input}"
```

LAYER 2: DESCRIBING THE GENERATED FANOUT

JUBE-style

```
parameterset:
  name: strong_scaling
  parameter:
    - name: nodes
      mode: python
      _: "[1, 2, 4, 8]"
    - name: taskspernode
      _: 4
    - name: problem_size
      _: large

step:
  name: run
  use: strong_scaling
```

ReFrame

```
class StrongScaling(rfm.RegressionTest):
    nodes = parameter([1, 2, 4, 8])

    @run_before('run')
    def set_resources(self):
        self.num_nodes = self.nodes
        self.num_tasks_per_node = 4

    problem_size = 'large'
    executable = 'srun'
    executable_opts = [
        './app',
        '--size', problem_size
    ]
```

Ramble-style

```
experiments:
  strong_scaling:
    matrix:
      - n_nodes: [1, 2, 4, 8]

    variables:
      processes_per_node: 4
      problem_size: large
      input: case.yaml
```

LAYER 3: MACHINE AND SYSTEM DEFINITIONS

JUBE Slurm system definition

```
systemParameter:
  nodes:
    type: int
    default: 1
  taskspernode:
    type: int
    default: 1
  threadspertask:
    type: int
    default: 1

  tasks:
    type: int
    mode: python
    value: nodes * taskspernode

  OMP_NUM_THREADS:
    export: true
    value: threadspertask

  queue: batch
  gres: NONE
  starter: srun
  timelimit: "00:30:00"
```

Ramble-style system variables

```
variables:
  scheduler: slurm
  launcher: srun
  system: jurecadc
  queue: dc-gpu

  n_nodes: 1
  processes_per_node: 4
  threads_per_process: 1

  mpi_command:
    srun -N {n_nodes}
         -n {n_ranks}
```

INSTRUMENTATION ACROSS FRAMEWORKS

JUBE, YAML-like

```yaml
instrumentation: runtime

injection:
  before_run:
    - module load nsys
  launcher: nsys
  launcher_args:
    - profile

success:
  file: job.out
  match: "SUCCESS"

metric:
  name: runtime
  file: job.out
  regex: "Time: (\\S+)"
  type: float
  units: s
```

ReFrame

```python
instrumentation = 'runtime'

@run_before('run')
def inject(self):
    self.prerun_cmds += [
        'module load nsys'
    ]
    self.executable = 'nsys'
    self.executable_opts = [
        'profile', './app'
    ]

success = sn.assert_found(
    r'SUCCESS', self.stdout)

runtime = sn.extractsingle(
    r'Time: (\S+)',
    ↪  self.stdout, 1, float)
```

Ramble workspace

```yaml
instrumentation: runtime

modifiers:
  - name: nsys
    mode: profile

success_criteria:
  completed:
    file:
    ↪  "{experiment_run_dir}/job.out"
    match: "SUCCESS"

figures_of_merit:
  runtime:
    file:
    ↪  "{experiment_run_dir}/job.out"
    regex: "Time:
    ↪  (?P<runtime>\\S+)"
    type: float
    units: s
```

LAYER 1: BENCHMARK SPECIFICATIONS

JUBE, YAML-like

```yaml
benchmark: app-case-single
application: app
case: case
experiment: single

system: jurecadc
partition: dc-gpu

parameters:
  nodes: 1
  taskspernode: 4
  threadspertask: 1
  input: case.yaml

execution:
  launcher: srun
  executable: ./app
  arguments:
    - --input
    - $input
```

ReFrame

```python
class
↪  AppCaseSingle(rfm.RegressionTest)
    descr = 'app case single'

    valid_systems =
    ↪  ['jurecadc:dc-gpu']
    valid_prog_environs =
    ↪  ['builtin']

    num_nodes = 1
    num_tasks_per_node = 4
    num_cpus_per_task = 1

    input = 'case.yaml'

    executable = 'srun'
    executable_opts = [
        './app',
        '--input', input
    ]
```

Ramble workspace

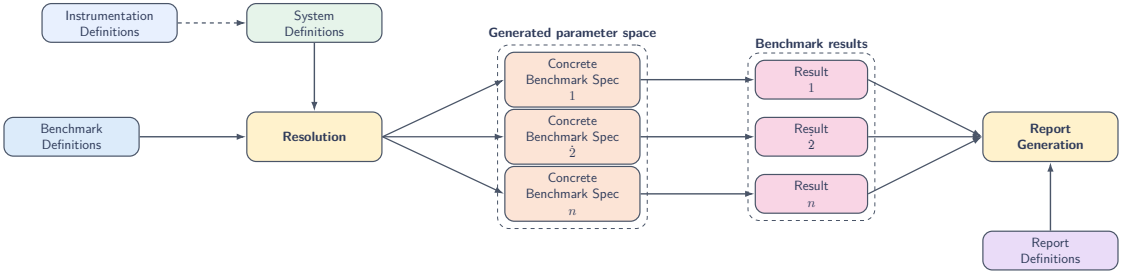
```yaml
benchmark: app-case-single
application: app
workload: case
experiment: single

system: jurecadc
partition: dc-gpu

variables:
  n_nodes: 1
  processes_per_node: 4
  threads_per_process: 1
  input: case.yaml

execution:
  launcher: srun
  executable: ./app
  arguments:
    - --input
    - "{input}"
```

STRUCTURE OF BENCHMARKS



JÜLICH Forschungszentrum

LAYER 1: BENCHMARK SPECIFICATIONS

JUBE, YAML-like

```yaml
benchmark: app-case-single
application: app
case: case
experiment: single

system: jurecadc
partition: dc-gpu

parameters:
  nodes: 1
  taskspernode: 4
  threadspertask: 1
  input: case.yaml

execution:
  launcher: srun
  executable: ./app
  arguments:
    - --input
    - $input
```

ReFrame

```python
class
↪  AppCaseSingle(rfm.RegressionTest)
    descr = 'app case single'

    valid_systems =
    ↪  ['jurecadc:dc-gpu']
    valid_prog_environs =
    ↪  ['builtin']

    num_nodes = 1
    num_tasks_per_node = 4
    num_cpus_per_task = 1

    input = 'case.yaml'

    executable = 'srun'
    executable_opts = [
        './app',
        '--input', input
    ]
```

Ramble workspace

```yaml
benchmark: app-case-single
application: app
workload: case
experiment: single

system: jurecadc
partition: dc-gpu

variables:
  n_nodes: 1
  processes_per_node: 4
  threads_per_process: 1
  input: case.yaml

execution:
  launcher: srun
  executable: ./app
  arguments:
    - --input
    - "{input}"
```

LAYER 1: TWO THINGS ARE MIXED TOGETHER

Application command interface

```yaml
application: app
case: case

arguments:
  input: case.yaml

execution:
  executable: ./app
  arguments:
    - --input
    - case.yaml
```

What does the application need?

Machine / hardware configuration

```yaml
system: jurecadc
partition: dc-gpu

resources:
  nodes: 1
  tasks_per_node: 4
  threads_per_task: 1

launcher: srun
```

Where and how is it run?

ISOLATING THE COMMAND-LINE INTERFACE

Kaira / OpenCLI-like specification

```yaml
command: app

options:
  input:
    spelling: --input
    type: path
    required: true

  backend:
    spelling: --backend
    type: enum
    values: [cpu, gpu]

  steps:
    spelling: --steps
    type: integer
    default: 100
```

Generated command line

```
./app \
  --input case.yaml \
  --backend gpu \
  --steps 100
```

Predictable command lines are desirable.

LAYER 2: DESCRIBING THE GENERATED FANOUT

JUBE-style

```
parameterset:
  name: strong_scaling
  parameter:
    - name: nodes
      mode: python
      _: "[1, 2, 4, 8]"
    - name: taskspernode
      _: 4
    - name: problem_size
      _: large

step:
  name: run
  use: strong_scaling
```

ReFrame

```
class StrongScaling(rfm.RegressionTest):
    nodes = parameter([1, 2, 4, 8])

    @run_before('run')
    def set_resources(self):
        self.num_nodes = self.nodes
        self.num_tasks_per_node = 4

    problem_size = 'large'
    executable = 'srun'
    executable_opts = [
        './app',
        '--size', problem_size
    ]
```

Ramble-style

```
experiments:
  strong_scaling:
    matrix:
      - n_nodes: [1, 2, 4, 8]

    variables:
      processes_per_node: 4
      problem_size: large
      input: case.yaml
```

LAYER 2: A COMMON GENERATED SPACE

- Tools may keep their own native parameter-space notation.
- Before execution, they emit a common generated space.
- Matrix like notation is the generated artifact to be used by the execution engine.

```yaml
generated_space:
  _union:
    - _product:
        variant: [single_node]
        nodes: [1]
        tasks_per_node: [4]
        problem_size: [small]
        input: [case.yaml]

    - _product:
        variant: [strong_scaling]
        nodes: [1, 2, 4, 8]
        tasks_per_node: [4]
        problem_size: [large]
        input: [case.yaml]
```

LAYER 3: MACHINE AND SYSTEM DEFINITIONS

JUBE Slurm system definition

```
systemParameter:
  nodes:
    type: int
    default: 1
  taskspernode:
    type: int
    default: 1
  threadspertask:
    type: int
    default: 1

  tasks:
    type: int
    mode: python
    value: nodes * taskspernode

  OMP_NUM_THREADS:
    export: true
    value: threadspertask

  queue: batch
  gres: NONE
  starter: srun
  timelimit: "00:30:00"
```

Ramble-style system variables

```
variables:
  scheduler: slurm
  launcher: srun
  system: jurecadc
  queue: dc-gpu

  n_nodes: 1
  processes_per_node: 4
  threads_per_process: 1

  mpi_command:
    srun -N {n_nodes}
         -n {n_ranks}
```

LAYER 3: MACHINE CONFIGURATION IS DIFFERENT

Problem

System descriptions often mix together three different things:

1. **Machine specs**
   partitions, node types, queues, limits, launchers
2. **Concrete executor parameters**
   nodes, tasks per node, CPUs per task, GPUs per node, time limit
3. **Derived or inferred values**
   total tasks, default mappings, implicit launcher choices

Direction

A concrete benchmark spec should contain explicit executor parameters, but not generated or derived parameter logic.

LAYER 3: CONCRETE SPEC

Avoid

- derived values,
- implicit defaults,
- framework-specific inference,
- parameter-generation rules.

Prefer

- independent executor variables,
- explicit resource choices,
- full concrete configuration,
- validation against a system definition.

Generation happens at a higher level than the concrete spec.
The concrete spec records one resolved benchmark run.

BRINGING THE PIECES TOGETHER

1. **Specify one benchmark**
   Benchmark config, application interface, and Slurm/system config.

2. **Describe combinations explicitly**
   Matrix-like products and unions describe benchmark spaces.

3. **Let tools generate the combinations**
   JUBE, ReFrame, Ramble, or exaCB emit the common space before execution.

WHERE TO START?

Start with the case of a single benchmark.

1. **Executor specification**
   Slurm or Flux command line spec

2. **Machine configuration specification**
   What machine, partition, node type?

Scaling studies and benchmark collections can be built on top of this.