DACCOSIM deployer documentation:

**DacRun user’s guide and technical manual**

Stephane Vialle, CentraleSupelec

*Stephane.Vialle@centralesupelec.fr*

February 23, 2017
Contents

1 What is DacRun? 5

2 DacRun user’s guide 7
   2.1 User account configuration ................................. 7
   2.2 DacRun syntax and options ................................ 7
   2.3 From FMUs to computing nodes mapping mechanism .... 11
   2.4 Command line and usage examples .......................... 12
   2.5 Contents of the output files ................................. 14

3 DacRun technical manual 17
   3.1 Software architecture ..................................... 17
   3.2 Details of remote processes launching ..................... 17
   3.3 Deployment algorithms .................................... 17
Chapter 1

What is DacRun?

DacRun is a Python software designed to easily compile, run and collect results of a DACCOSIM\textsuperscript{1} application on a PC cluster, and also on a local machine.

- DacRun is compliant with OAR and SLURM cluster managers, and can run on a single machine without any cluster manager installed.
- DacRun allows to control the collection and aggregation of distributed execution logs and results.
- DacRun achieves many checks of any operations run on a computing node, and can generate detailed logs and error reports, which is very helpful to debug a distributed execution.
- DacRun is developed under Python 2.7, and requires the installation of ppserver.py package on all used computing nodes.

DacRun has been developed by CentraleSupelec (Metz Campus), in collaboration with EDF (Saclay Lab), in the framework of the RISEGrid Institute\textsuperscript{2}.

\textsuperscript{1}https://daccosim.foundry.supelec.fr
\textsuperscript{2}http://www.supelec.fr/342p_36889/risegrid.html
CHAPTER 1. WHAT IS DACRUN?
Chapter 2

DacRun user’s guide

2.1 User account configuration

First, you need to have access to python 2.7 to use DacRun, and when any shell script or program runs a Python code then Python 2.7 has to be automatically run instead of Python 3.

Second, when using DACCOSIM and DacRun, you need to define the DACCOSIM_ROOT environment variable, referencing the DACCOSIM directory. If using Bourne shell, add the following line at the end of your .bashrc and .profile files:

```bash
export DACCOSIM_ROOT=/....../DACCOSIM5.5.0
```

Third, to use DacRun you need to set the directory DACCOSIM_ROOT/fr.daccosim.dacrun into your PATH. If using Bourne shell, add the following line at the end of your .bashrc and .profile files:

```bash
export PATH=$DACCOSIM_ROOT/fr.daccosim.dacrun:$PATH
```

Remark: when using a cluster, many processes will be run on remote nodes, and configuration of .profile file is highly important and should match configuration of .bashrc file.

2.2 DacRun syntax and options

To run DacRun tool you must enter `dacrun.py <option list>` on the command line, as it is a Python 2.7 program. Many options exist, and five are mandatory:

```bash
dacrun.py
  # Mandatory path options
  -p PROJECT
  -fp FMUFOLDERPATH
  -br BLOCKRESULTS -bl BLOCKLOGS -vl VIRTUALNODELOGS

  # Automatic creation of missing directories
  [-mkdir]

  # Co-simulation computation setting
  [-m METHOD]
  [-ab ABORDER]
  [-e EXTRAPOLATION]
  [-iss INITIALSTEPSIZE]
  [-minss MINIMALSTEPSIZE] [-maxss MAXIMALSTEPSIZE]
  [-sd SIMULATIONDURATION]
```
CHAPTER 2. DACRUN USER’S GUIDE

# Co-simulation execution setting
[-ic INTERNALCOMMUNICATION]
[-om ORCHESTRATIONMODE]
[-cs COHORTSIZE]
[--noresult] [--nolog] [--noio]
[--trace]

# Adaptation to the cluster manager used
[-cm CLUSTERMANAGER]

# Information retrieval
[--daccosimversion]
[--dacrunversion]
[-V VERBOSE]
[-h]

# Work in progress (do not use)
[--cleanbefore]
[-gp GATHERPATH] [--parallelgather PARALLELGATHER]
[--zip] [--merge] [--eraselog]

Mandatory path options

-p PROJECT
-fp FMUFOLDERPATH -lp LIBRARYPATH
-br BLOCKRESULTS -bl BLOCKLOGS -vl VIRTUALNODELOGS

These six paths can be absolute or relative to the current working directory (where is run dacrun.py).

- **-p PROJECT** sets the directory path including a Java src subdirectory, a bin subdirectory where all compiled Java programs will be stored, and any other related subdirectory (like result and log file directories, or input file directory).

  Warning: the src subdirectory MUST exist, while the bin subdirectory is automatically created if missing when DacRun is launched.

- **-fp FMUFOLDERPATH** sets the path to a repository of the FMU files required by the co-simulation.

- **-br BLOCKRESULTS and -bl BLOCKLOGS** set the directory paths where each FMU (or block of FMU) will store its CVS result files and its log files. Moreover, each virtual node aggregates its FMU result files into one large virtual node CVS file (first step of the co-simulation result aggregation) also stored in the BLOCKRESULTS directory.

- **-vl VIRTUALNODELOGS** sets the directory path where each virtual node stores three files: an overview of its configuration and execution time (useful for performance analysis), and two trace files (error messages and normal output messages) function of the verbose level (see verbose control option).

  Warning: these result and log file directories are cleaned at each DacRun launch: all stored files are erased (to avoid confusions with result and log files of previous runs).

When running on a cluster, we need to decide to use mounted and shared directories, or node local directories (like /tmp). Some clusters do not provide mounted and shared directories, then PROJECT and FMUFOLDERPATH directories have to be replicated on each computing nodes. When
mounted directories are available, they should be used for these three directories, without significant impact on performance.

But function of the IO configuration and performances of the cluster, user can choose comfortable mounted directories to store large result files and/or log files, or can choose local directories to use local disk on each computing node. So, `BLOCKRESULTS`, `BLOCKLOGS` and `VIRTUALNODELOGS` paths can be local directories to increase co-simulation performances during computations, but then results (and logs) will need to be collected from each node after computations. Some `DacRun` options are under development to achieve this collection.

**Automatic creation of missing directories**

`[-mkdir]`

This option enables the automatic creation of missing result and log directories, specified with previous options. Its main objective is to simplify development of benchmark repositories and benchmarking shell scripts.

**Co-simulation computation setting**

`[-m METHOD]`

`[-ab ABORDER]`

`[-e EXTRAPOLATION]`

`[-iss INITIALSTEPSIZE]`

`[-minss MINIMALSTEPSIZE] [-maxss MAXIMALSTEPSIZE]`

`[-sd SIMULATIONDURATION]`

These six options control the computations of the co-simulation.

- `/-m METHOD` sets the name of the computation method: constant time steps with 'constant' method (default value), or variable time steps with 'Euler', 'Richardson' and 'Adams' (Adams-Bashforth) methods. Case is not sensitive.

- `ab ABORDER` sets the order of the Adams-Bashforth method, in the range $[3 - 9]$. Default value is 3.

- `-e EXTRAPOLATION` enables the extrapolation of input values inside a time step ('1') or disables it ('0', default value).

- `-iss INITIALSTEPSIZE` sets the initial value of the step size for variable time step methods and the definitive step size for the constant time step method.

- `-minss MINIMALSTEPSIZE` and `-maxss MAXIMALSTEPSIZE` set the boundaries of the step size interval for variable time step methods.

These three values are expressed in seconds, and have floating point type. Default values are set by user when designing the co-simulation graph.

- `-sd SIMULATIONDURATION` sets the duration of the simulation, i.e. the elapsed simulated time we aim to simulate.

It is expressed in seconds and has floating point type. Default values are set by user when designing the co-simulation graph.
Co-simulation execution setting

[-ic INTERNALCOMMUNICATION]
[-om ORCHESTRATIONMODE]
[-cs COHORTSIZE]
[--noresult] [--nolog] [--noio]
[--trace]

These seven options allow to control the execution of the co-simulation, in order to improve performances and to make easier some performance measurement.

• -ic INTERNALCOMMUNICATION sets the intra-virtual-node communication mechanism. All FMUs of a virtual node and their control task (their local Master) are located on a same physical node. The 'InProc' communication mode will ensure local communications will use shared message queues, instead of crossing complex middleware layers. The '0MQ' communication mode will use 0MQ middleware communication mechanism for both intra- and inter-virtual-node communications. Default value is set when designing the co-simulation graph.

• -om ORCHESTRATIONMODE sets the orchestration mode of the co-simulation to 'Overlapped': overlapping FMU communications and control operations and late FMU computations, or 'Ordered': the three sub-steps are not overlapped. Overlapped mode is faster with constant time steps without any rollbacks. Default value is set when designing the co-simulation graph.

• -cs COHORTSIZE sets the size \( n \geq 0 \) of the FMU cohort. Default is 0 meaning there is no cohort mechanism, and all FMUs of the node are concurrently run in a thread at each computation substep. Then, the OS schedules the threads, that share the different cores of the node. When a cohort size is specified \( n > 0 \) a maximum of \( n \) FMU are concurrently activated, limiting the race to access cores, cache memory, memory bandwidth...

Warning: this option is deprecated, and could disappear in future versions of DacRun.

• --noresult, --nolog and --noio are boolean flags allowing to imped writing of results, or logs, or both logs and results. These options are independent of the required result and log paths (see the mandatory path options).

These options have been introduced to easily measure the gain of time when writing no results and/or no logs, and to deduce the time elapsed in the IO operations. They facilitate performance analysis of DACCOSIM distributed runs.

• --trace generates a trace file for each FMU, in the directory of the virtual node logs, and in the /tmp directory of each used computing node (see section ??). Trace files are considered part of the log files, and managed with the log files.

Adaptation to the cluster manager used

[-cm CLUSTERMANAGER]

This option specifies a cluster management environment: OAR or Slurm (well known cluster managers), or: 'local' when using a single PC, not a cluster. The case is not sensitive.

In any case DacRun needs to access the list of available nodes to run a DACCOSIM co-simulation. When not using a cluster but a basic laptop, the 'local' value allows DacRun to consider this machine like one cluster node. So, DacRun can be used on any platform, and can be adapted to future clusters with different management environments.
Information retrieval

[\(--daccosimversion\)]
[\(--dacrunversion\)]
[\([-V\) VERBOSE\)]
[\([-h\)]

These options allows to get informations on DacRun and DACCOSIM executions.

- \(--daccosimversion\) runs a DACCOSIM co-simulation but prints the DACCOSIM version number in each vnode log file and abort computations. However, mandatory options (path options) must be supplied and valid.

- \(--dacrunversion\) prints the DacRun version number on screen and abort deployment. However, mandatory options (path options) must be supplied and valid.

- \([-V\) VERBOSE\)] specifies the verbose level of DacRun: '0’ leads to silent execution (prints only the execution time on screen), '4’ prints many DacRun execution details on screen and in vnode log and trace files. Mandatory options (path options) must be supplied and valid.

- \([-h\)] prints a DacRun help on the screen.

2.3 From FMUs to computing nodes mapping mechanism

There are two steps in the mapping process of the FMUs on the computing nodes: FMUs are mapped on DACCOSIM virtual nodes (vnodes), and next these virtual nodes are mapped on physical cluster nodes (or PCs).

FMUs/virtual nodes mapping

When designing a co-simulation graph with the DACCOSIM GUI or DSL, the number of vnodes \((n_v)\) is set and each FMU (or FMU block) is associated to a vnode. In fact, a vnode is a JVM process, and FMUs associated to a vnode are only threads inside the JVM of the vnode. Different mapping strategies are available:

1. Both the number of vnodes and the FMU/vnode mapping can be specified by user.

2. The number of vnodes or a maximal vnode number can be set by user, and the mapping can be done by DACCOSIM.

   (a) The mapping algorithm can be a basic round robin mechanism on a FMU list,
   (b) or it can be a round robin mechanism on a list of FMU ordered according to their computation load,
   (c) or it can be a smart heuristic taking into account FMU ordered according to their computation load.

3. Considering the most advanced smart heuristics, the number of vnodes can be computed by the smart mapping algorithm of DACCOSIM, in order to minimize the global execution time \((n_v^{ideal})\), assuming there will be available computing nodes to host these vnodes.

   Current DACCOSIM GUI and DSL support full explicit solution (1), and DACCOSIM DSL supplies a basic round robin mapping on a number of vnodes set by user (2a). More advanced smart mapping strategies are under development and will be available soon.
Virtual nodes/computation node mapping

Mapping several vnodes on one node would lead to install several JVM processes on a PC. This could increase performances on big nodes with several chips and Non Uniform Memory Architecture architecture (NUMA). However, the current version 2016.0.3 of DacRun installs only one vnode per (physical) node.

How many computation nodes are required?

Then, DacRun is used to deploy this $n_v$ DACCOSIM co-simulation on $n_{cn}$ (physical) computing nodes. Current version of DacRun maps only one vnode per node. So, DacRun version 2016.0.3 requires you allocate at least $n_v$ physical nodes to deploy your DACCOSIM co-simulation ($n_{cn} \geq n_v$):

- If you allocate more physical nodes than required ($n_{cn} > n_v$), then some physical nodes will remain unused but DacRun will succeed to deploy your co-simulation.

- If you allocate less physical nodes than required ($n_{cn} < n_v$), then DacRun will abort.

- When using only one PC (like your laptop), you must design a co-simulation graph mapped on only one virtual node and use the local cluster management mode (and use the -cm local option).

2.4 Command line and usage examples

Structure of a DACCOSIM project directory

A traditional DACCOSIM project consists in a main directory, like myproject (name is free), initially including two subdirectories:

- myproject/src: mandatory directory, storing Java source files generated by DACCOSIM co-simulation graph automatic analysis,

- myproject/bin: directory automatically created by DacRun if does not exists, or automatically cleaned by DacRun if already exists.

One result file and two log file directories are also required to run DacRun. They are usually subdirectories of the project one. This is not mandatory, but is the configuration of this example:
2.4. COMMAND LINE AND USAGE EXAMPLES

- myproject/bres: results of the FMUs (or blocks of FMUs).
- myproject/blogs: logs and traces of the FMUs (or blocks of FMUs).
- myproject/vlogs: logs of the virtual nodes.

These three subdirectories are automatically created by DacRun if do not exist and the -mkdir option is specified. They are automatically cleaned at each DacRun launch.

Finally, any other subdirectory can be added to the project. For example to store input data files:

- myproject/documentation: repository of input data files.

Figure 2.1 summarizes the initial content of our myproject directory. The src subdirectory contains 3 Java source files, meaning this project has been designed to create 3 virtual nodes, running on 3 computing nodes (with current vnode/node mapping strategy of DacRun).

Example of a DacRun command line

The following linux commands enter the main project directory, and execute the DacRun command. It specifies to compile and run a co-simulation on 1000s of simulated time, with 5s constant time steps, without extrapolating input values. The last two options specify the cluster used is managed by OAR (so, DacRun will interact with OAR) and to retrieve maximal logs (verbose level is set to its higher value):

cd myproject
dacrun.py -p . #project directory
   -br bres #FMU results directory
   -bl blogs #FMU logs directory
   -vl vlogs #vnode logs directory
   -fp FMUs-Linux-32-64 #FMU repository
   -mkdir #create directories if not exist
   -m Constant #use constant step size
   -e 0 #do not extrapolate input values
   -sd 1000 #compute 1000s of simulated time
   -iss 5 #use step size of 5s
   -cm OAR #interact with OAR cluster env.
   -V 4 #retrieve maximal logs
   --trace #generate a trace file for each FMU

Of course, real command line is:

dacrun.py -p . -br bres -bl blogs -vl vlogs -fp FMUs-Linux-32-64
   -lp $(DACCOSIM_ROOT)/DACCOSIM_deployment_setup -mkdir -m Constant -e 0 -sd 1000 -iss 5
   -cm OAR -V 4 --trace

Remark: Order of the option has no importance, paths can be absolute or relative, and paths can end with or without ‘/’.

After DacRun execution, many new files have been created and project directory content is illustrated on figure 2.2. The Documentation directory is an example of input data repository, and the FMUs-Linux-32-64 directory is an example of FMU repository.
CHAPTER 2. DACRUN USER’S GUIDE

src:
- DACCOSIMGlobalMaster.java
- DACCOSIMLocalMaster2.java
- DACCOSIMLocalMaster4.java
- genericMachineNames.csv

bin:
- DACCOSIMGlobalMaster.class
- DACCOSIMLocalMaster2.class
- DACCOSIMLocalMaster4.class
- DACCOSIMGlobalMaster$1.class
- DACCOSIMLocalMaster4$1.class
- DACCOSIMLocalMaster2$1.class
- mapping_file.csv

blogs:
- block_BoundaryConditions.log
- block_CrawlSpaceTemp.log
- block_IndoorWall1.log
- block_IndoorWallDown.log
- block_IndoorWallTop.log
- block_ZoneNorth1.log
- block_ZoneNorthDown.log
- block_ZoneNorthTop.log
- block_ZoneSouth1.log
- block_ZoneSouthDown.log
- block_ZoneSouthTop.log

Documentation:
- Donnees/Meteos/METEONORM/France/trappes.txt

Figure 2.2: Final content of our example project directory (myproject/)

2.5 Contents of the output files

Files of block result directory

Block (or FMU) result directory is myproject/bres in our example. It contains CSV files (ready to be interpreted by Excel). Each file store the output values of a FMU (or a FMU block) at each time step. First line contains the name of each attribute (column), and the first column is the time step. Here is the beginning of the bres/block/IndoorWall1.csv file:

```
Time; dt; IndoorWall1.FlowN; IndoorWall1.FlowS; IndoorWall1.TempN; IndoorWall1.TempS;
0.0; 0.0; -0.0; -1.311377673118841E-11; 293.15; 293.15;
5.0; 5.0; -160046.77203858254; -60657.554975598236; 293.1473102157917; 293.1473102157917;
10.0; 5.0; -158135.13070499248; -60383.62459857707; 290.18807023402144; 292.01922619622434;
...```

Files of block log directory

Block log directory is named myproject/blogs in our example. It contains text files. Each one stores some synthetic informations about one FMU (or one block) execution. Here is the content of bres/block/IndoorWall1.log file:

```
```
2.5. CONTENTS OF THE OUTPUT FILES

Run start: 2016/08/25 08:59:51
Intra-node communication protocol: Shared queue
Co-initialization method: Sequential output propagation
Step size method : Constant step size
Orchestration mode : Send First Control Last
Real input extrapolation: No
Start time: 0.0 seconds
Stop time : 1000.0 seconds
StepSize: 5.0 seconds

Number of iterations (number of calls to doStep): 200
Number of successful calculation steps : 200
Calculation time : 76.45743 milliseconds
Communication time: 170.466232 milliseconds

Warning: Calculation and communication times are measured on a multi-core computing node, running many threads. A computing node runs one computing thread per hosted FMU, and many communication and control threads. So, these elapsed times can be longer than the real CPU time consumed by the FMU. When using many nodes and running only one FMU per node, the FMU computing time stored at the end of each FMU log file can be considered right. When the node hosts and runs several FMUs, it is better to consider only the vnode computing time (see the next paragraph).

Files of vnode log directory

Vnode log directory is named myproject/vlogs in our example. It contains three text files per virtual node: a synthetic log file, a standard output trace file, and an error output trace file.

Here is the synthetic log file vnode_node2.log:

Node: Node2
Run start: 2016/08/25 08:59:51
Intra-node communication protocol: Shared queue
Co-initialization method: Sequential output propagation
Step size method : Constant step size
Orchestration mode : Send First Control Last
Real input extrapolation: No
Start time: 0.0 seconds
Stop time : 1000.0 seconds
Time elapsed for co-initialization: 0.0 milliseconds
Time elapsed for co-simulation : 354.0 milliseconds

The elapsed times at the end of each vnode log file should be very close of elapsed times of others vnode, because some synchronization barriers exist between the vnodes, and each physical computing node runs only one vnode (in current DacRun version 2016.0.3). So, a vnode execution is not interrupted/disturbed by another vnode execution on the same node, and a global vnode execution time includes the waiting time on synchronization barriers with others vnodes. However, some little differences can appear, specially when running very short co-simulations.

The standard output trace file trace_vnode_node2.out.log contains both execution trace of the vnode FMUs, of DACCOSIM software, and of DacRun (at the end of the file). The error output trace file trace_vnode_node2.err.log contains error message, still from both all FMUs.
in the vnode, DACCOSIM and DacRun. However, these files remain complex to interpret, and are devoted to internal developers of DacRun and DACCOSIM.

**Complementary log files in /tmp**

When launching compilation and execution of each Java source file on a remote computing node, there is a *black out* period. No log is possible on the standard output of the DacRun process, and vnode log files are not yet created and writable.

In order to trace the deployment process during this short *black out*, a file is created on each computing node in its /tmp directory. This mechanism has already appeared very useful to detect and fix some DacRun bugs. These files are named `DACCOSIM_trace.cosimCompute_XXX....txt` and are generated when the option `--trace` is specified on the command line.

But again, the traces printed in these files remain complex to interpret, and are devoted to internal developers of DacRun and DACCOSIM.
Chapter 3

DacRun technical manual

3.1 Software architecture
3.2 Details of remote processes launching
3.3 Deployment algorithms