Standard Workflow Components

Build 10.7.0.0202512081401_SNAPSHOT

Table of Contents

List of Tables

1. Cluster Component

1.1. Synopsis

The Cluster component allows submission of jobs to a cluster.

1.2. Rationale

The Cluster component submits jobs described by a given job script to the queuing system of a cluster. It allows to upload directories beforehand and to download directories after the job is terminated.

To check if jobs are finished, the Cluster component polls the queuing system every minute and asks for their states. The connection to the cluster is established via SSH. For the submission, a directory (sandbox-[uuid]) is created on the cluster in the user's home directory. It will serve as the current working directory for all remote command line calls.

The remote directory structure is as follows:

```
/sandbox-[id]
    /iteration-0
        /cluster-job-0
            /input
            /output
        /cluster-job-1
            /input
            /output
        …
        /cluster-job-shared-input
    /iteration-1
        /cluster-job-0
            /input
            /output
        …
        / cluster-job-shared-input
    …
    job
```

The job script is uploaded to /sandbox-[id]/job. The job submission is done from /sandbox-[id]/iteration-[n]/cluster-job-[n]/.

If the job failed and the Cluster component should be marked as failed, a file named job_failed must be created in /sandbox-[id]/iteration-[n]/cluster-job-[n]/output. The content of the file is used as error message. The output directories are not downloaded for the failed job and all remaining jobs terminated afterwards.

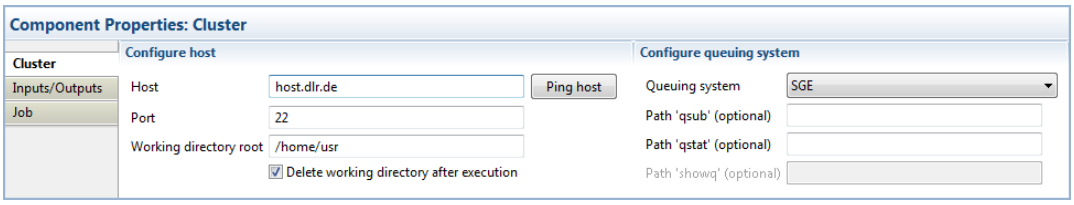
1.3. Usage

The Cluster component is configured as follows:

1.3.1. Cluster tab

In the Cluster tab define the information needed to connect to a cluster via SSH. Define host IP or resolvable host name, port number, etc. The working directory root is the folder, where the sandbox

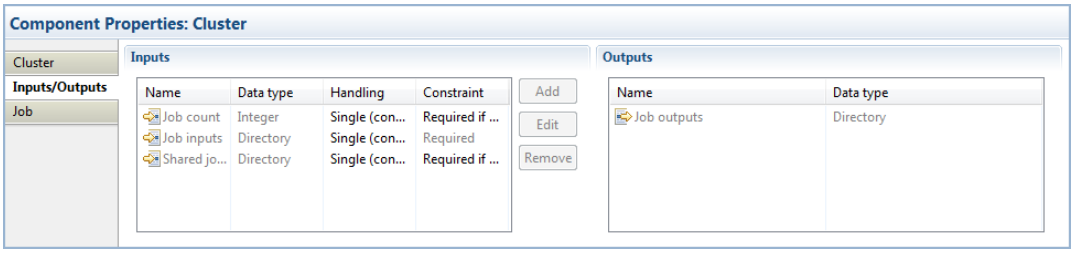
folder mentioned above is created. Also define the queuing system running on the cluster. In some cases, the queuing system console commands like qsub, qstat, etc. are not known within a non-interactive SSH shell on the cluster. For that, you can optionally define the absolute paths to the required commands explicitly. If you don't know them, just type 'which qsub' etc. on a cluster's console and you will get them.



1.3.2. Inputs/Outputs tab

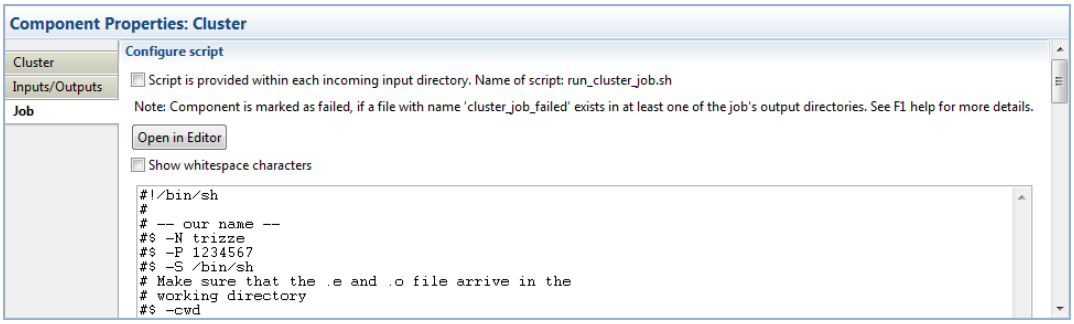
In the tab Inputs/Outputs you see the inputs and outputs of the Cluster component. The inputs and outputs are static and cannot be modified except the scheduling behavior.

1. Job count: The count of jobs to submit on each iteration
2. Job inputs: Input directories which are uploaded before each iteration to /sandbox-[id]/iteration-n/cluster-job-n/input (in the order as they arrive, 0 for first directory, 1 for second, etc.)
3. Shared job input: Input directory which is uploaded before each iteration to /sandbox-[id]/iteration-n/shared-input
4. Job outputs: Output directory which is download after each iteration from sandbox-[id]/iteration-n/cluster-job-n/output

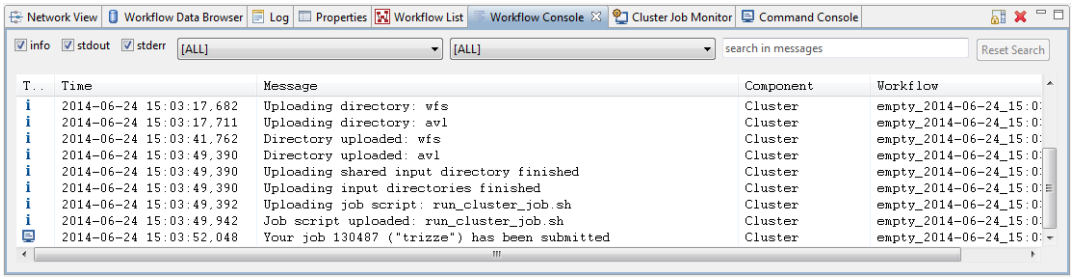


1.3.3. Job tab

The job itself is either described in the Job tab or is provided within each input directory (/sandbox-[id]/iteration-[#]/cluster-job-[#]/input). Select the check box accordingly. If it is provided within each input directory, the name of the script must be: run_cluster_job.sh



To see the native standard out and error of the job submission see the Workflow console.


Network View
Workflow Data Browser
Log
Properties
Workflow List
Workflow Console
Cluster Job Monitor
Command Console
info
stdout
stderr
[ALL]
[ALL]
search in messages
Reset Search
T..
Time
Message
Component
Workflow
2014-06-24 15:03:17,682 Uploading directory: wfs Cluster empty_2014-06-24_15:0
2014-06-24 15:03:17,711 Uploading directory: avl Cluster empty_2014-06-24_15:0
2014-06-24 15:03:41,762 Directory uploaded: wfs Cluster empty_2014-06-24_15:0
2014-06-24 15:03:49,390 Directory uploaded: avl Cluster empty_2014-06-24_15:0
2014-06-24 15:03:49,390 Uploading shared input directory finished Cluster empty_2014-06-24_15:0
2014-06-24 15:03:49,390 Uploading input directories finished Cluster empty_2014-06-24_15:0
2014-06-24 15:03:49,392 Uploading job script: run_cluster_job.sh Cluster empty_2014-06-24_15:0
2014-06-24 15:03:49,942 Job script uploaded: run_cluster_job.sh Cluster empty_2014-06-24_15:0
2014-06-24 15:03:52,048 Your job 130487 ("trizze") has been submitted Cluster empty_2014-06-24_15:0

2. Converger Component

2.1. Synopsis

The Converger component checks values of type float and integer for convergence by comparing current values with values from previous runs.

2.2. Rationale

The Converger component checks values of all of its inputs for convergence. It compares the values of the current run with the values from previous runs. Absolute and relative convergence is supported. If the absolute difference is less then a pre-defined epsilon the values are considered as converged in terms of absolute convergence. In case of relative convergence the absolute difference is divided by the maximum of the considered values. The Converger component considers a loop as converged as soon as all if the values to consider are converged.

To prevent endless-running loops, a maximum number of convergence check can be defined.

In the final run of the Converger component, the values most recently received are sent to the outputs with the suffix '_converged'.

2.3. Usage

The Converger component has four configuration tabs.

2.3.1. Converge criteria tab

Define the values for the epsilons in case of absolute and relative convergence. You can also define the number of iterations (k) which should be considered. In case of k > 1 not only the current and previous values but the minimum and maximum from the set of current plus previous k values are considered. So k = 1 means only current and previous values are considered, k = 2 current and two previous, etc. You can also limit the number of convergence checks.

2.3.2. Inputs/Outputs tab

Create an input for each value to consider. An output with the same name and an output with the suffix '_converged' are created automatically. In the "Add" dialog you can also decide whether to define a start value. In case if not, an additional input is created with the suffix '_start'.

2.3.3. Nested and Fault-tolerant Loop tab

See general section "Workflows" in the user guide.

3. CPACS Writer Component

3.1. Synopsis

The CPACS Writer component saves incoming CPACS content as an xml file on the local file system. It has a runtime GUI to show the CPACS file in TiGL Viewer.

3.2. Usage

On the “Target” tab you can select the target folder where the file will be stored. If you select the option “Overwrite CPACS file each run” the filename will be “cpacs.xml” and the file will be overwritten each time the component runs. If you do not select this option a new file will be written on each run with a filename following the scheme “cpacs_[iteration count]_[timestamp].xml”.

You can define the scheduling of the static input channels on the tab “Inputs/Outputs”. You are also able to add dynamic input or output channels here. Thus it is possible to map single values into the XML dataset via XPATH declaration. Moreover, you can select single values in order to write them in an output channel.

> **Note**
> Search for the XPathChooser help in the User Guide (3. Usage -> 2. Workflows -> 2.3 Workflow components) to garner further information about the usage of XPaths.

On the “Workflow Data” tab you can select whether to store history data for each component run or not.

3.3. Runtime GUI

Double clicking on the CPACS Writer component during or after workflow execution opens the “CPACS Geometry” view where the current CPACS content is shown. By clicking the button “Show CPACS dataset in TIGL Viewer” the corresponding view “TIGL Viewer” will show the CPACS geometry in the TiGL Viewer, if that viewer is configured. See the help entry for the TiGL Viewer component and the configuration reference for more information on how to configure the TiGL viewer.

4. Database Component

4.1. Synopsis

With the database component, MySQL and PostgreSQL databases can be accessed. Database management however is not the focus of this component.

4.2. Rationale

The database component executes one or more SQL statements and can write the result of each statement into one output if specified.

The statement can be composed using placeholders for inputs configured in the component (see "Inputs/Outputs").

Currently supported statement types are SELECT, INSERT, UPDATE, DELETE. Management tasks, like creating users and handling their privileges as well as altering the database structure are not the purpose of the component.

4.3. Usage

4.3.1. Registering a database connector

To register a JDBC database connector, you simply need to place the connector .jar file in the subfolder ".../extras/database_connectors" in your RCE installation folder. It will automatically be loaded when you restart RCE. As mentioned above: Currently, only drivers for MySQL and PostgreSQL databases are supported. If you require a different driver class feel free to contact us.

Note

In previous releases we shipped a JDBC connector with RCE. Since we no longer deliver this for security reasons, at least one JDBC database connector must be registered before using the database component. Please inform us if there are problems with the integration of current versions of connectors.

4.3.2. Defining a database connection

In the properties view of the database component there is a *"Database"* tab. Here you can define the database connection this component works on. Note that currently drivers for MySQL and PostgreSQL databases are supported. The credentials required to access the database can be entered later on when you execute the workflow.

4.3.3. Use the credentials

When you execute a workflow that contains a database component you are asked for the username and password. Note that you can store the password in an encrypted storage if you check the "save" check box.

4.3.4. Database statements

For each database component you can enter multiple statements. They are defined in the "Statement" tab. Pressing the "< + >" tab will open a new statement tab. Every tab must not contain more than one statement. For every statement you can define whether its result should be written to an output. To dynamically compose database statements by using placeholders for inputs you can make use of the "Input" group. A placeholder is added at the current caret position which will be replaced by the actual input value at runtime. Likewise the "Templates" group will insert templates for the given statement types which you can edit to fit your purpose.

4.3.5. Writing multiple times to the same output

It is possible to configure multiple statements to write their result sets to the same output. These results are queued, which can cause subsequent components to run multiple times to consume the queued values. Note that these components must allow queuing of input values for this to work.

4.3.6. Output "success"

There is a static output of type boolean named *"success"*. It is set to *true* if the given statement and result set distribution was successful. It is useful when a database statement does not yield a result set (like an INSERT statement, for instance) but should trigger the start of a succeeding component. If there are multiple statements defined in a component instance then the "success" output is written when all statements have been processed.

4.3.7. Valid statement types

The database component is designed to query and update databases on a lightweight basis. As already mentioned above, typical database management statements like creating, altering and dropping tables or working on views and user accounts is not the aim of this component. Therefore, only a set of four database statement types is supported. Meaning that each statement must begin with one of the following phrases:

- SELECT
- INSERT
- UPDATE
- DELETE

4.3.8. Handling Small Tables

Inputs of type "Small Table" can only be used in INSERT statements.

Example:

```
INSERT INTO table_name (id, col1, col2, col3) VALUES ${in:mySmallTable}
```

Outputs of type "Small Table" are filled by converting the result set from the database to RCE's data types. Note that small tables cannot be encapsulated in small tables.

4.3.9. Handling Result Sets

If a result set is empty but configured to be written to an output, this is interpreted as an error.

If a result set has exactly one row and one column, it is tried to be mapped to the respective RCE data type.

If a result set has more than one entry and the respective output channel is small table a mapping is executed.

If the respective output channel is boolean, short text, integer or float but the result set has more than one entry this is interpreted as a potential erroneous configuration and causes the component to fail.

If values in the result set are mapped to Java's data type "Big Decimal" it cannot be processed as there is currently no data type of RCE that can represent it.

If values in the result set are mapped to Java's data type Timestamp it cannot be processed as there is currently no data type of RCE that can represent it. As a workaround you can cast or transfer the timestamp to some textual representation with SQL functionalities and work on with this.

If values in the result set are null, it is mapped to RCE's data type "Empty" and has the textual representation *"nil"*.

4.3.10. Local Execution Only

Please note that the database component cannot be published and remotely used. To use the database in your local workflow make sure you can access the database from your machine and configure the database component accordingly.

4.4. Examples

The following examples refer to the "world" example which can be found at the mysql website: https://dev.mysql.com/doc/index-other.html

The following list gives examples for the statement types without inputs configured in the component:

- `SELECT * FROM City;`
- `DELETE FROM City WHERE ID = 3076;`
- `UPDATE City SET Population = 1000000 WHERE ID = 3071;`
- `INSERT INTO City (ID, Name, CountryCode, District, Population) VALUES (4080, 'Cochem', 'DEU', 'Rheinland-Pfalz', 5213);`

The following examples demonstrate the usage of inputs configured in the component:

- `SELECT * FROM City WHERE ID = ${in:id};`
- `DELETE FROM City WHERE ID = ${in:id};`

- `UPDATE City SET Population = ${in:population} WHERE ID = ${in:id};`
- ```
  INSERT INTO City (ID, Name, CountryCode, District, Population) VALUES(${in:id}, ${in:name},
   ${in:code}, ${in:district}, ${in:population});
  ```

Considering inputs of type small table, the example above in the context of the world database would look as follows, assumed the input is properly defined:

```
INSERT INTO City (ID, Name, CountryCode, District, Population) VALUES ${in:smallTable};
```

For further information you may want to refer to the MySQL documentation: https://dev.mysql.com/doc/refman/5.7/en/sql-syntax.html
```

5. Design of Experiments Component

5.1. Synopsis

The Design of Experiment component sends values (floating-point numbers) to other components. The values are either generated on the base of a design method or are provided by a custom design table. The values can be used to sample a solution within a bounded space. They are independent to each other.

5.2. Usage

First, outputs must be defined. Each output is of type float and has a lower and an upper bound. The definition of inputs is optional. If inputs are defined, the DOE component maps one set of output values to one set of input values. I.e., output values are sent as soon as input values (corresponding to previously sent output values) are received. Queuing of input values is not allowed. If no inputs are defined, the output values are sent all at once at workflow start.

Second, the design method must be selected. You can choose between four methods:

- Full factorial design [http://en.wikipedia.org/wiki/Factorial_experiment]
- Latin hypercube design [http://en.wikipedia.org/wiki/Latin_hypercube_sampling]
- Monte Carlo design [http://en.wikipedia.org/wiki/Monte_Carlo_method]
- Custom design

The first three methods generate the output values on the base of established design methods (see links above). For two of them, the values are random. You can choose a seed in order to reuse the same values later. The number of samples can be defined with the option "Number of levels"/"Desired runs" and can be communicated to other components using the "Number of samples" output, which is sent out on the first iteration of the DOE component.

The last method allows to define a custom design table. The table at the bottom is editable and values can be entered. The table can be saved as a csv file and can be re-loaded later on. It is also possible to define a custom sample range by modifying the "Start at sample #" and "End at sample #" option.

For help concerning nested and fault-tolerant loop settings, see the general section "Workflows" in the user guide.

6. Evaluation Memory Component

6.1. Synopsis

The Evaluation Memory component stores results of loop runs and re-uses them in future runs.

6.2. Rationale

The Evaluation Memory component is used within loops. It can speed up loops by reusing results of past loop runs. Usually, the Evaluation Memory component is used before/after the loop driver component (e.g., Design of Experiments, Optimizer). It takes the design values of the loop driver component. Then, it either sends stored result values directly back to the loop driver or it forwards the values to the actual evaluation loop. If the evaluation loop is done, the newly evaluated result values are fed back to the loop driver via the Evaluation Memory component, so that it can store the result values together with the design values sent before for later reuse. The values are stored in a file on the file system.

6.3. Usage

You need to configure three things: inputs/outputs, the path to the evaluation memory file wherein the values are stored respectively should be stored, and whether loop failures should be considered as valid loop results.

6.3.1. Evaluation Memory File

The path to the evaluation memory file is either configured in the 'Memory File' configuration tab or is defined at workflow start if the checkbox is checked appropriately.

6.3.2. Handling Loop Failures

If a component in the loop fails due to invalid parameters and sends a value of type 'not-a-value', the value will pass the Evaluation Memory component and will be stored as a loop result for the design values sent into the loop. By (un-)checking the checkbox 'Consider loop failures as valid loop results' you can decide wether the stored values should be considered and re-used in case of equal design values.

6.3.3. Inputs/Outputs

The inputs and outputs are configured in the "Inputs/Outputs" configuration tab. There are five tables, three of which are read-only. In the first one (seen from left to right and top to bottom), create an input for each design value sent from the evaluation loop driver to the loop. For each input an output is created as well in the second table. These are used in case the design values are just forwarded into the loop. In the third table, create one input for each result value evaluated by the loop. Again, for each

input an output is created as well, this time in the fourth table. These are used to forward the result values to the evaluation loop driver.

For each design value of type float or integer added as an input in the first step above, a tolerance on this value may be given. Incoming design values are compared to stored ones with respect to this tolerance. Say, e.g., that there exist two inputs x1 and x2, both of type float. The input x1 is configured with a tolerance of 10%, while the input x2 is configured with a tolerance of 20%. If the evaluation memory component receives the input values x1 = 10.0 and x2 = 20.0, then it first checks whether it has already stored results for these precise values. If this is not the case, it checks whether it has stored result values for inputs in the ranges 9.0 <= x1 <= 11.0 and 16.0 <= x2 <= 24.0.

- If no such value exists, then the input values are forwarded to the loop to be evaluated.
- If exactly one such value exists, then the stored result values are returned to the loop driver.
- The behavior for the case that multiple such items exist can be configured in the "Evaluation Memory" tab: Strict behavior causes the Evaluation memory component to forward the input values to the loop for evaluation, while lenient behavior causes the component to arbitrarily pick any of the stored values and return them to the loop driver. The behavior can only be chosen if there is at least one input that has some tolerance configured.

Finally, the inputs and outputs must be connected to the evaluation loop driver and to the loop properly. If done, there are actually two loops: evaluation loop driver - Evaluation Memory component and Evaluation Memory component - evaluation loop. The first one is driven by the evaluation loop driver, the second one is driven by the Evaluation Memory component.

7. Excel Component

7.1. Synopsis

The Excel Component is designed to access Microsoft Excel files within RCE and to execute macros.

7.2. Rationale

This component wraps an existing Microsoft Excel file which is linked to RCE. The general principle is:

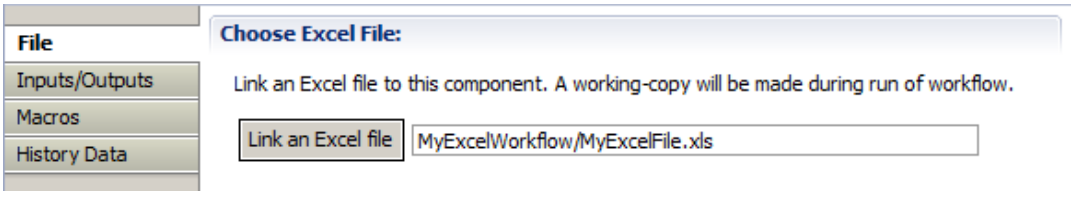
1. Copy of the existing Excel file as temporary file in a temporary folder (working copy)
2. Wait for all input channels which are needed to run the component (depending on how input handling and constraints are set)
3. Execute VBA-macro “Before Excel run”
4. Copy all input channels to their specific cell ranges. If there are multiple values in an input channel the first value occurring will be copied (“first in, first out”).
5. Update all formulas
6. Execute VBA-macro “After input variables are written”
7. Update all formulas
8. Read all output channel values from their specific cell range
9. Execute VBA-macro “After Excel run”
10. Delete temporary Excel file

7.3. Usage

The Excel component has three tabs for configuration.

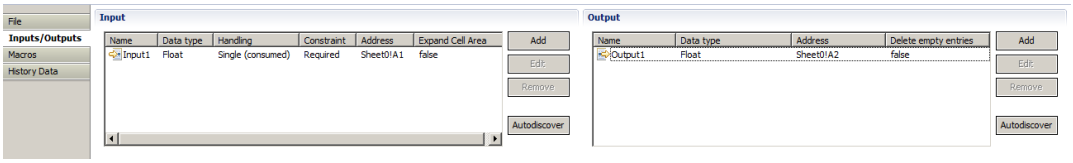
7.3.1. File

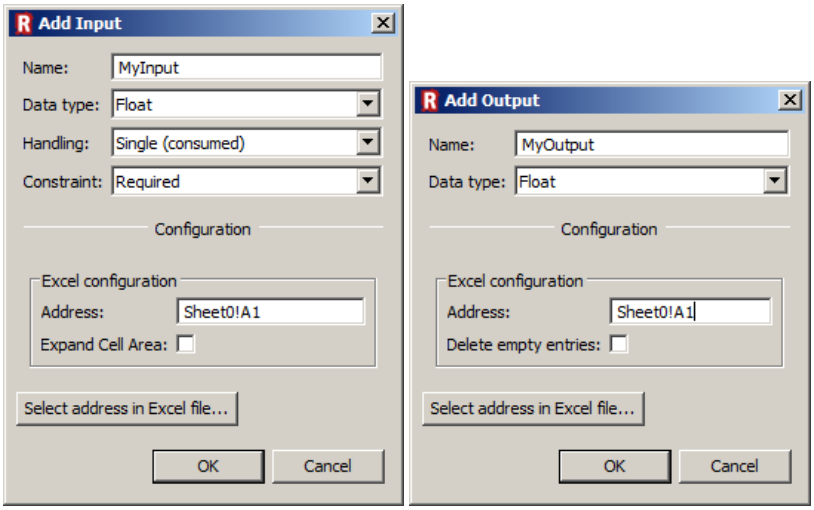
In the 'File' tab you can link an Excel file to the RCE component. Note that the Excel file must be located within the workflow's project. Click "Link an Excel file ..." and navigate to the Excel file of your choice.



7.3.2. Inputs/Outputs

The 'Inputs/Outputs' tab can be used to create inputs and outputs for the Excel component. The configuration of the channels of both types is (mostly) similar. An RCE-channel is always connected to a specific Excel cell range. The button “Autodiscover” discovers automatically all input and output channels which are described as user-specific cell areas which start with “I_” for input-channels and with “O_” for output channels respectively.



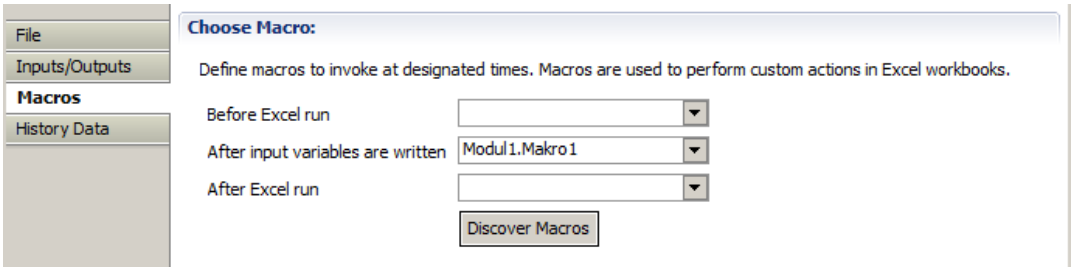


The following list gives a short description of all channel configuration parameters:

Element	**Description**
Name	The name of the RCE-channel
Data Type	See RCE user guide (Coupling Workflow Components)
Handling	See RCE user guide (Coupling Workflow Components)
Constraint	See RCE user guide (Coupling Workflow Components)
Expand Cell Area (only input channels)	If the user does not know the size of the cell area which she wants to insert, this field can be set to true. Now the upper left cell area field can be selected in “Address”-parameter. From that address on the complete table will be inserted, ignoring existing cell entries
Delete empty entries (only output channels)	If a selected cell area contains empty rows at the end these will be cut off when setting this field to true.
Button "Select address in Excel file... "	To select a cell area in Excel this button opens the file in Microsoft Excel so the user can select a specific cell area.

7.3.3. Macros

In the 'Macros' tab you can configure which VBA macros are to be run during runtime.



When in Microsoft Excel properties the access to the VBA-project object model is granted the macros are available VBA macros are discovered automatrically. All available macros can be chosen in the respective dropdown menu.

7.4. Requirements

1. Microsoft Excel must be installed. The component is being tested with Microsoft Excel 2010.

2. For an automatic detection of VBA-macros the VBA-project must be trusted. Please start Excel as administrator. In Microsoft Excel 2010 navigate to "File -> Options -> Trust center -> Trust center settings -> macro settings" and check the "trust access to the VBA project object model" option.

8. Input Provider Component

8.1. Synopsis

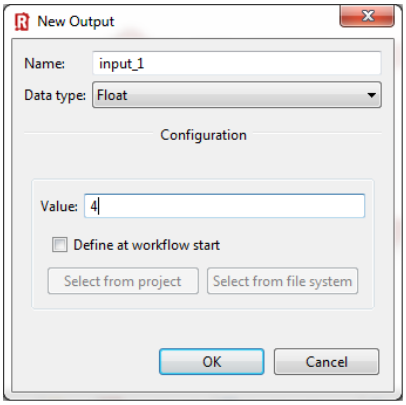
The Input Provider component sends values to inputs of other components.

8.2. Rationale

The Input Provider sends values to other components, e.g. as starting values. Therefore, the Input Provider writes specified values to its outputs. The outputs must be connected to the inputs of the other components. The values are sent once and immediately after the workflow has been started.

8.3. Usage

For each value to send you need to create an output for the Input Provider component by clicking the "Add..." button next to the outputs table. For each output you can decide whether to define the value directly or to define it at workflow start by (un-)checking the check box "Define at workflow start".

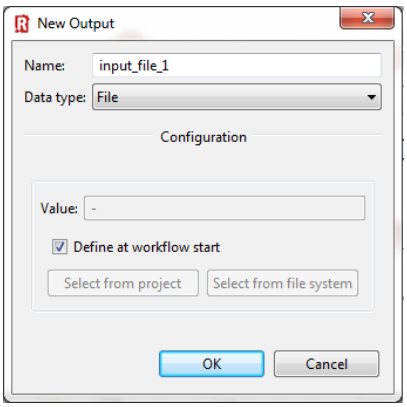


If it is defined directly it is stored in the workflow file. If you share the workflow with others the defined values will be shared as well.

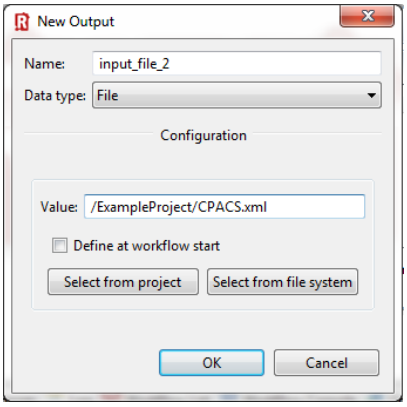
If it is defined at workflow start only a placeholder is stored in the workflow file. It will be replaced with the actual value defined at workflow start. If you share the workflow with others only the placeholder will be shared and the other person needs to define the value at workflow start by herself.

If you like to send files or directories to other components you have three options. You can choose the option that suits you best. In terms of workflow sharing, consider the following:

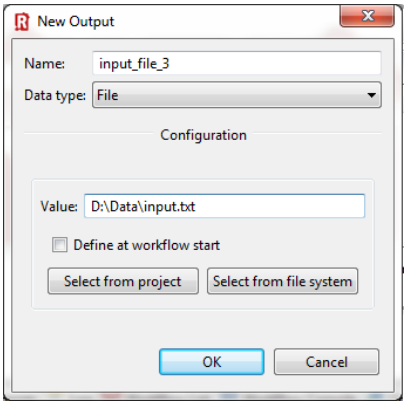
1. Select the file/directory to send at workflow start (you will be asked for selection in the workflow execution wizard). Choose this option if you like to share the workflow (.wf file) with others and don't want to share the file/directory (e.g. because it contains sensitive data).



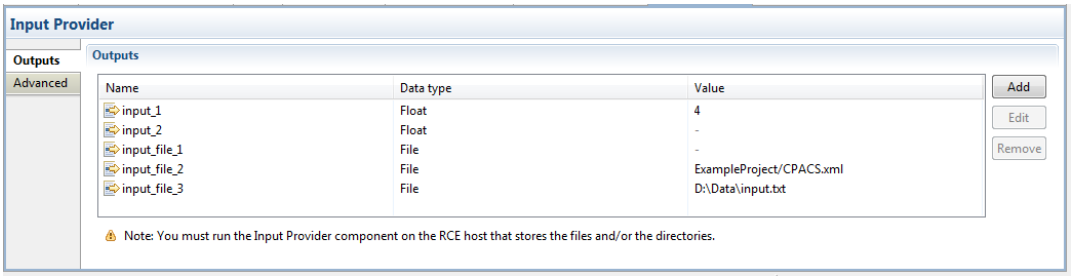
2. Select the file/directory from the project in the workspace where the workflow file is stored. Choose that option if you like to share the workflow (.wf file) as well as the file/directory with others. In that case, you need to share the whole project (e.g. as an archive file - right-click on the project and export it as an archive file). The other person needs to import that project into the RCE workspace. The workflow will run out-of-box using the file/directory of the shared project.



3. Select the file/directory from file system. Choose that options if you don't like to share the workflow with others at all and if the file/directory needs to remain at a specified place in the local file system.



You will see your outputs in the table similar like this:



After that, you can connect the outputs from the Input Provider to any other compatible input from other components. The values (either defined directly or defined at workflow start) will be sent to the connected inputs of the other components immediately after the workflow has been started.

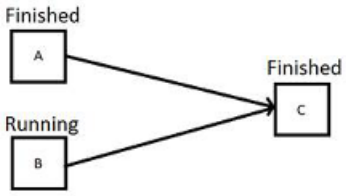
9. Joiner Component

9.1. Synopsis

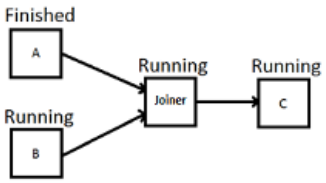
The Joiner component joins multiple connections to a single one.

9.2. Rationale

It is forbidden to connect two outputs to one single input. The reason for that is the current approach how it is determined whether a workflow is finished. If a workflow component is finished it sends a dedicated data package to all of the inputs connected. As soon as such a data package is received the input is closed. A workflow component is considered as finish if all of its inputs are closed. If an input is connected to two outputs its workflow component might be considered as finished by mistake:



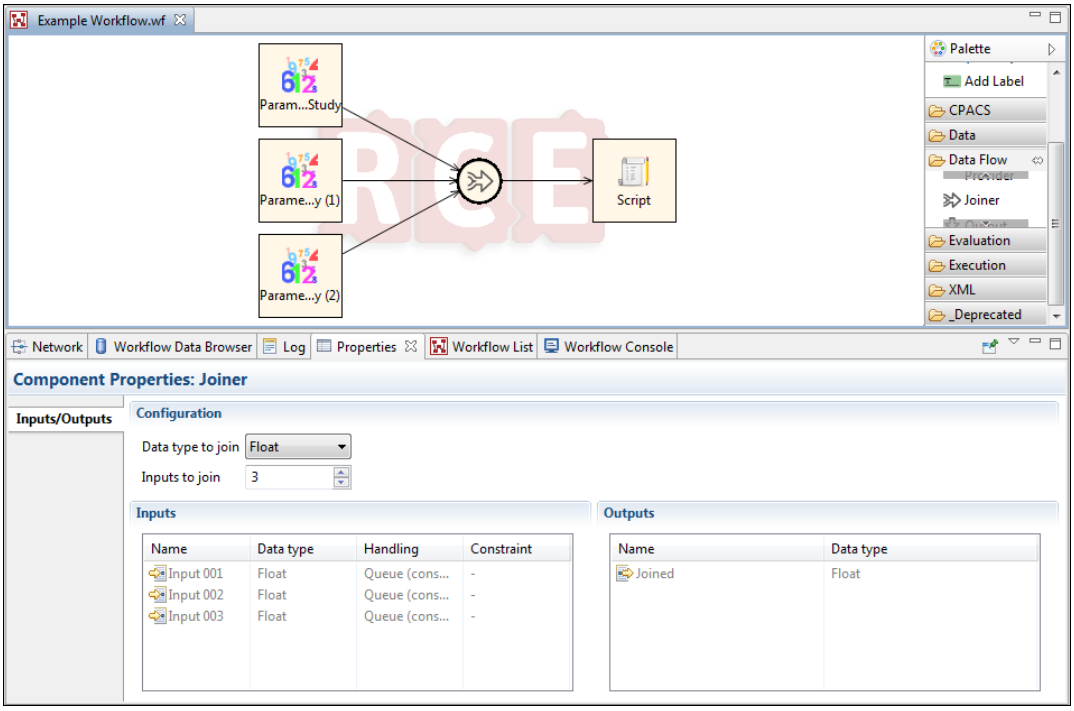
Here, workflow component C finishes as soon as A is finished. The Joiner component solves the problem by joining multiple connections into a single one. It will send the dedicated data package closing the inputs of C only if it has received the data package from A and B:



Workflow component C will finish as soon as A and B are finished.

9.3. Usage

In the Inputs/Outputs configuration tab choose the data type and the number of inputs to join. Note that the connections in RCE are typed. Thus, only connections of the same type can be joined.


Example Workflow.wf
Param...Study
Parame...y (1)
Parame...y (2)
Script
Palette
Add Label
CPACS
Data
Data Flow
Joiner
Evaluation
Execution
XML
_Deprecated
Network
Workflow Data Browser
Log
Properties
Workflow List
Workflow Console
Component Properties: Joiner
Inputs/Outputs
Configuration
Data type to join
Float
Inputs to join
3
Inputs
Name
Data type
Handling
Constraint
Input 001
Float
Queue (cons...
-
Input 002
Float
Queue (cons...
-
Input 003
Float
Queue (cons...
-
Outputs
Name
Data type
Joined
Float

10. Optimizer Component

10.1. Synopsis

The Optimizer component allows the optimization of design variables in a workflow.

10.2. Rationale

The Optimizer component uses a black-box optimization software library. By starting the component, an input file for the selected optimization algorithm is created and the software will be started in the background. Different optimization packages can be installed.

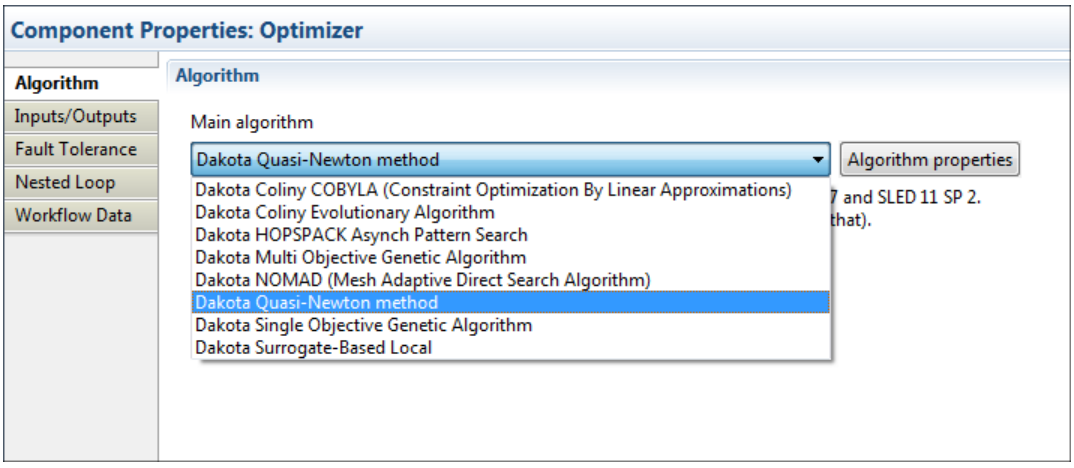
One package that is delivered with RCE on Windows x64 and Linux x64 machines is Dakota. Dakota was tested on the following distributions: Windows 10, Windows Server 2019, Ubuntu 18.04, and CentOS 8.

For more information about the Dakota Package see the Dakota Project page [https://dakota.sandia.gov/].

10.3. Usage

To use the optimizer component you need to do the following steps:

- In the algorithm tab you can choose an optimization algorithm that fits your problem. There are several algorithms from the Dakota package available.



There are properties for each algorithm. For editing, click on the 'Algorithm Properties' button. The appearing dialog shows the properties of the chosen algorithm. The properties differ from algorithm to algorithm. For more information about the properties see the documentation of the package.

If you have an operating system, on which the default Dakota does not work, you have the option to choose a custom Dakota binary by checking the box 'Use custom dakota binary'. You will be asked for the Dakota executable path at workflow start. This can be either a downloaded version from the

dakota website or a self compiled binary with the source code from the Dakota website. For more information about compiling Dakota see: https://software.sandia.gov/trac/dakota/wiki/Developer

- The next step is to define the inputs and outputs for the component. There are three types of data you can configure.

  1. The objective function variables are the one to be optimized. For each variable you can specify if it should be minimized, be maximized or be searched for a specific value (solve for). If you have more than one objective function, you can define their weight in the optimization process. If there is only one objective function, the weight will be ignored. Note that some algorithms support single- and multi-objective optimization.

     If you have defined some design variables, you also can choose if the objective function you create has gradients or not. If you select this, new inputs will appear in the connection editor, which are intended for the values from the gradients. Note that for every design variable you have, a new input for the objective function exists.

  2. The constraint variables are used to bound particular variables to a region or value. If a solution is found but it causes a constraint variable to be out of bounds, the solution is not valid. Again, the constraints can have gradients.

  3. The design variables are the values that are modified by the algorithm to find an optimal solution. For each variable you have to define a startvalue which will be the initial value for the optimization algorithm. You also have to define the lower and upper bound for each variable.

  All data have to be either a float or a vector data type.

  Note

  Some Dakota algorithms do not support discrete optimization. For discrete design variables the following algorithms are available:

  - Dakota Coliny Evolutionary Algorithm
  - Dakota Multi Objective Genetic Algorithm
  - Dakota NOMAD
  - Dakota Single Objective Genetic Algorithm

  All other Dakota methods will ignore discrete design variables during optimization.

  There are three more tables for endpoints. They are just read-only and can not be modified. They are created automatically when configuring the Optimizer.

  1. The 'Start value inputs' table shows which design variables need start values before running. Start values can be the starting value for this design variable, if the option 'Has start value' in the design variable dialog is not chosen.

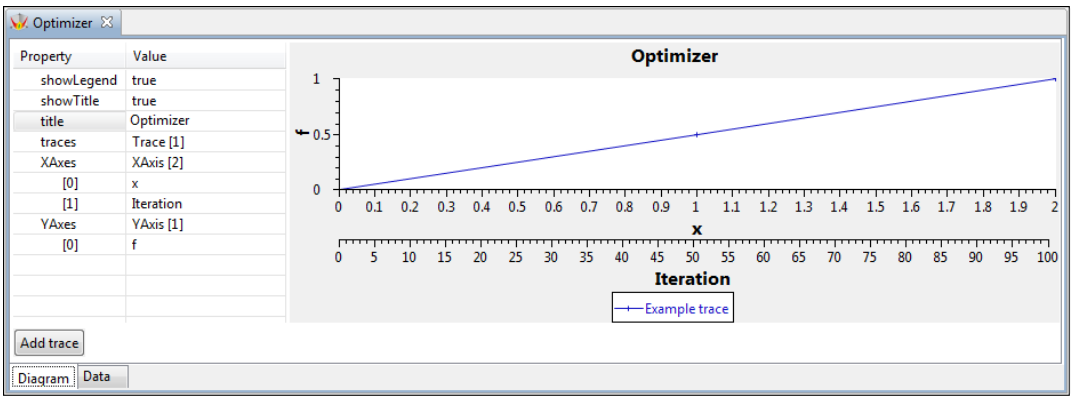
     Other possible start values are the lower and upper bounds of a design variable, if the option 'Has unified bounds' is not selected.

  2. In the 'Optimal design variables outputs' table, the outputs for the values of the optimized point are shown.

  3. The 'Gradients' table shows which objective functions should have gradient inputs as well. This is chosen in the dialog for the objective functions ('Has gradient').

After these steps the optimizer component is ready to start. In a running workflow, you are able to see the output from the Dakota optimizer in the Workflow console.

By double clicking the Optimizer component in the runtime view of the workflow, you will get the values the optimizer produces and the possibility to export these values to an Excel file. You are also able to plot a graph with the given results in the diagram tab.



For help concerning nested and fault-tolerant loop settings, see the section 'Usage/Workflows' in the user guide.

10.4. Optimization Algorithm API

Manual on how to use the Optimization Algorithm API.

10.4.1. Basic Concept

The RCE Optimization Algorithm API enables the user to integrate their own optimization algorithms into RCE and use them in the common Optimizer Component. The API is based on Python, so the user's algorithm must be callable from phyton, e.g. through a system call.

10.4.2. How to integrate an algorithm into RCE

The location where the API looks for integrated algorithms is in RCE's profile folder: `<profile>/integration/optimizer/`. Below this path, every subfolder must have a specific structure in order to be recognized as an optimizer integration. Each subfolder must contain two folders with the names "gui_properties_configuration" and "source". An example integration "example_algorithm" is available at the installation path of RCE in the subfolder "examples/optimization_algorithm_api/optimizer". Copy it to your profile and you can use this algorithm in RCE.

10.4.2.1. GUI Properties Definition

Within the "gui_properties_configuration" folder, the GUI of the Optimizer Component must be configured for the integrated algorithms. At first, there has to be a file named "algorithms.json". In this file, RCE looks for the algorithms to be defined. The file is structured as follows:

```
{
"Name of algorithm":"name of json file for algorithm"
}
```

For example:

```
{
"Name of method1" : "method1",
"Name of method2" : "method2"
}
```

where "method1.json" and "method2.json" exist in the same directory.

The method files also have to be in a certain format which looks like this:

```
{
    "methodName":"Name of method",
    "optimizerPackage":"generic",
    "specificSettings":{
        "propertyName":{
            "GuiName":"Name shown in optimizer GUI",
            "dataType": ["Int" | "Real" | "String" | "Bool"],
            "SWTWidget": ["Text" | "Combo" | "Check"],
            "DefaultValue": "",
            "Value":"",
            "Validation":""
        }
    }
}
```

The "optimizerPackage" must always have the value "generic" and the "methodName" must have the same value as defined in the "algorithms.json". In the section "specificSettings", properties can be defined in order to make them configurable in the RCE GUI. You can choose between three different types of GUI widgets and four different data types. The properties will be displayed when you open the "Algorithm properties..." view in the RCE's Algorithm section of the Optimizer component. Three categories are available to organize the properties on different tabs: "commonSettings","specificSettings" and "responsesSettings".

Every property must have the following fields:

GuiName: The name that is displayed in the "Algorithm properties..." view and describes the property.

dataType: The data type of the current property. Valid values are:

- **Int**: an integer number
- **Real**: a float number
- **String**: a text

SWTWidget: This value defines what kind of GUI element is used for the property. Valid values are:

- **Text**: a text box where the user can enter any string
- **Combo**: a dropdown menu with pre defined values. When using the Combo, you have to define the values to be shown, using:
  - **Choices**: A comma separated list of the values, e.g. "Option 1, Option 2"
- **Check**: a checkbox to select an option

DefaultValue: The default value that is chosen if the user does not manually enter a value for the property. For Combos, this must be one of the "Choices".

Value: The value must always be an empty string ("").

Validation: For Int or Real data types you can add a validation. Possible validations are:

- >, >=, <, <= followed by a number, e.g. ">=0"
- "required" or "optional" if a value must be entered or can be empty
- empty string "" if no validation is required

All required and configurable properties for the integrated algorithm should be defined in a json file using the format described above. Apart from that, no further adjustments are necessary in the "gui_properties_configuration" folder.

You will find a more detailed example json file at the end of this manual. (cf. Section 10.4.2.3, “Example GUI configuration json”)

10.4.2.2. Source Folder

In the "source" folder, the algorithm logic must be defined. Two files are mandatory, which will be the entry point for the Optimizer Component. One file must be named "python_path", which only contains one single line that points to the executable of a python installation. The other file must be named "generic_optimizer.py". This script must call your own optimizer method. In this script you can use the Optimizer Algorithm API. The API contains three modules. Import the modules as follows:

```
from RCE_Optimizer_API import configuration
from RCE_Optimizer_API import evaluation
from RCE_Optimizer_API import result
```

Module Description:

- *configuration*: This module contains all information that is needed to configure the optimization method. You can get the design variables names and counts and the objective names and weigths. Furthermore you can access the property values configured by the user in the GUI.

- *evaluation*: Use this module you start an evaluation run in RCE and get the result of each evaluation and end the optimizer.

- *result*: If an evaluation is done, it generates a new result object. It contains objective and constraint values, their gradients and the failed inputs from RCE and provides methods to access them. The result object is lost at the next evaluation unless it is explicitly saved somewhere.

  For detailed information on the modules and the included methods see Section Section 10.4.3, “Module Description”

10.4.2.3. Example GUI configuration json

```
{
    "methodName" : "Name of method",
    "optimizerPackage" : "generic",
    "commonSettings" : {
        ...
    },
    "specificSettings" : {
        "textExample" : {
            "GuiName" : "Enter Value:",
            "GuiOrder" : "1",
            "dataType" : "Real",
            "SWTWidget" : "Text",
            "DefaultValue" : "1.0",
            "Value" : "",
            "Validation" : "<=1.0"
        },
        "comboExample" : {
            "GuiName" : "Select parameter",
            "GuiOrder" : "2",
            "dataType" : "String",
            "SWTWidget" : "Combo",
            "Choices" : "choice1,choice2,choice3",
            "DefaultValue" : "choice1",
            "Value" : ""
        },
        "checkboxExample" : {
            "GuiName" : "Any flag:",
            "GuiOrder" : "3",
            "dataType" : "Bool",
            "SWTWidget" : "Check",
            "DefaultValue" : "false",
            "Value" : ""
        }
    },
    "responsesSettings" : {
        ...
    }
}
```

Note

Property names must be unique on each tab. Otherwise the last configuration is used.

Note

The field "GuiOrder" is optional. Use this field to specify or change easily the order of the widgets in the GUI.

10.4.3. Module Description

Table 10.1. configuration.py

Method	Description
def get_algorithm()	Returns the selected algorithm
def get_design_variable_count()	Returns the number of design variables
def get_design_variable_names()	Returns a list of variable names
def get_design_variable_max_values()	Returns a dictionary of the variables and their corresponding upper bound
def get_design_variable_min_values()	Returns a dictionary of the variables and their corresponding lower bound
def get_design_variable_max_value(variable_name)	Returns the upper bound of the given variable "variable_name"
def get_design_variable_min_value(variable_name)	Returns the lower bound of the given variable "variable_name"
def get_start_values()	Returns a dictionary of the variables and their corresponding start values
def get_start_value(variable_name)	Returns the start value of the given variable "variable_name"
def get_step_values()	Returns a dictionary of the variables and their corresponding start values
def is_discrete_variable(variable_name)	Returns whether the given variable "variable_name" is discrete or not
def get_constraint_names()	Returns a list of constraint names
def get_constraint_max_values()	Returns a dictionary of the constraints and their corresponding upper bound
def get_constraint_min_values()	Returns a dictionary of the constraints and their corresponding lower bound
def get_constraint_max_value(constraint_name)	Returns the upper bound of the given constraint "constraint_name"
def get_constraint_min_value(constraint_name)	Returns the lower bound of the given constraint "constraint_name"
def get_objective_names()	Returns a list of objectives names
def get_objective_weights()	Returns a dictionary of the objectives and their corresponding weight
def get_optimization_targets()	Returns a dictionary of the objectives and their corresponding optimization target
def get_optimization_target(name)	Returns the optimization target of the given objective "name" or None if an objective "name" does not exists
def get_common_properties()	Returns a dictionary of all common properties and their corresponding values

Method	Description
def get_common_property(name)	Returns the value of the given common property "name" or None if a property "name" does not exists
def get_common_property_keys()	Returns a list of all common property keys
def get_specific_properties()	Returns a dictionary of all specific properties and their corresponding values
def get_specific_property(name)	Returns the value of the given specific property "name" or None if a property "name" does not exists
def get_specific_property_keys()	Returns a list of all specific property keys
def get_responses_properties()	Returns a dictionary of all responses properties and their corresponding values
def get_responses_property(name)	Returns the value of the given responses property "name" or None if a property "name" does not exists
def get_responses_property_keys()	Returns a list of all responses property keys

Table 10.2. evaluation.json

Method	Description
def evaluate(number_evaluation, design_variables, grad_request = False)	Starts the evaluation run with the given run number, designs variables and a boolean whether gradients are requested or not (default value is False). The result object of the current run is returned. Note: The design variables are handed over in an alphabetically sorted list as displayed in the Properties view of RCE's GUI. Be aware that uppercase is treated before lowercase variable names. Vectors are passed entry-wise. Example: Given the following design variables "Var1", "Vec" and "var2" with the values 1, [2,3,4] and 2 in evaluation run number 5. Start the evaluation run with *evaluate(5, [1,2,3,4,2])*.
def finalize(optimal_evaluation_number)	Ends the Optimization with the given run number as the optimal run. The optimal values are written to the outputs.

Table 10.3. result.py

Method	Description
def get_constraint_gradient(constraint_name)	Returns the gradient value of the given constraint "constraint_name" or None if the constraint does not exists or no gradient is defined
def get_constraint_value(constraint_name)	Returns the value of the given constraint "constraint_name" or None if a constraint "constraint_name" does not exists
def get_failed()	Returns a list of the failed optimization runs
def get_objective_gradient(objective_name)	Returns the gradient value of the given objective "objective_name" or None if the objective does not exists or no gradient is defined
def get_objective_value(objective_name)	Returns the value of the given objective "objective_name" or None if the objective does not exists
def has_gradient(name)	Returns True if the given input "name" has a gradient defined, False otherwise

11. Output Writer Component

11.1. Synopsis

The Output Writer stores outputs from other components on the local file system.

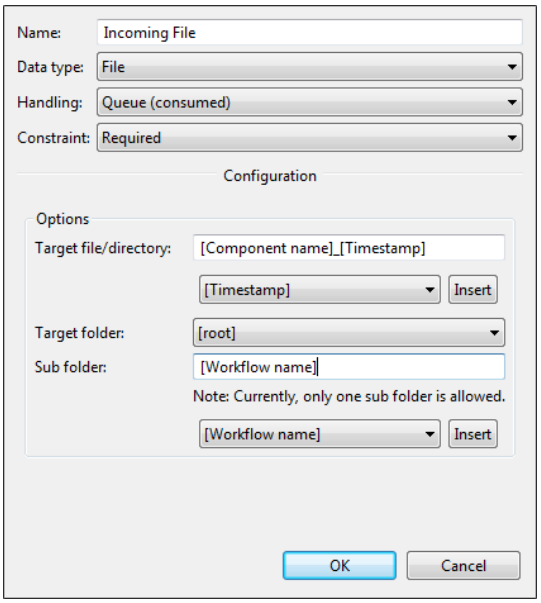
11.2. Rationale

The Output Writer receives inputs from other workflow components and saves them in a pre-configured folder. The inputs can either be files or directories, which are written directly to the local file system, or inputs of simple data types (Boolean, Float, Integer, Short Text), which are written into text files of a user-defined format.

11.3. Usage

The "Root Location" tab allows the user to define the location where the outputs should be stored on the file system. Here the user can also configure whether existing files or directories on the file system are to be overwritten or not.

Output to save must be send to the Output Writer via inputs that can be defined on the "Inputs" tab. If you add a new input of type file or directory, you must define the desired target name and path (of the file or directory). Both of them can be created using placeholders:



Target file/directory: The name the file will be given on the local file system. You can insert different placeholders by clicking the “Insert” button. The provided placeholders are:

- [Component name]: Name of the Output Writer component in the workflow.
- [Input name]: Name of the input you define at the top of this dialog.

- [Timestamp]: Date and time at file creation.
- [Timestamp at workflow start]: Date and time at workflow start.
- [Workflow name]: Name of the workflow.
- [Execution count]: Execution count of the Output Writer.
- [Original filename]: The name the file/directory had before it was sent to the Output Writer.

It is possible to combine placeholders within a single name.

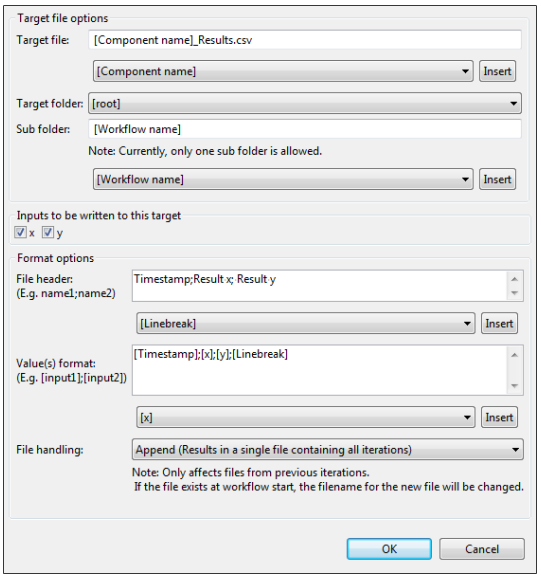
Target folder: The folder, where the file or directory should be stored. It is relative to the [root] folder (see below). Currently, only one sub folder below the [root] folder is supported. You can either select a folder, which was already used for other inputs or you can "create" a new one just by defining a new folder name within in the text box. Again, the folder name can contain placeholders.

Root folder: The files and directories received via the inputs will be saved to the [root] folder. You can either select the [root] folder at workflow start or you can define a "static" one within the component's configuration tab below the inputs table. This folder is used on every workflow execution. Note: Defining a static folder might cause problems, if you execute the workflow on another RCE node (e.g. the defined hard drive doesn't exist on the other machine).

For inputs of simple data types, you only have to specify the type of the input in the "add input" dialog. If you are using such inputs, you have to specify targets on the "Data Sheet" tab. A target receives the values of one or more simple data inputs and writes them into a text file of user-specified format. Several targets can be specified in one Output Writer. However, each input can only be written into one target.

> **Note**
>
> The Output Writer only writes output into a target file when values for ALL of the selected inputs have arrived. I.e. it is expected that for each iteration exactly one value for each of the inputs arrives.

In the "Add target" dialog you have to specify the following:



Target file: The name the file will be given on the local file system. You can insert different placeholders by clicking the "Insert" button. The provided placeholders are:

- [Component name]: Name of the Output Writer component in the workflow.
- [Timestamp at workflow start]: Date and time at workflow start.

- [Workflow name]: Name of the workflow.

Target folder: The folder where the file should be stored. It is relative to the [root] folder (see below). Currently, only one sub folder below the [root] folder is supported. You can either select a folder, which was already used for other inputs or you can “create” a new one just by defining a new folder name within in the text box. Again, the folder name can contain placeholders.

Inputs involved: Here you can select which inputs should be written into the target file. Only inputs of simple data types are shown here. Inputs that are already selected for another target are not selectable.

File header: Here you can define the header of the output file, which will be written once at the beginning of the target file (only if “Append” is selected in the “file handling” dropdown). You can insert different placeholders by clicking the “Insert” button. The provided placeholders are:

- [Linebreak]: A linebreak.
- [Timestamp]: Date and time at file creation.
- [Execution count]: Execution count of the Output Writer.

Value(s) format: Here you can define the format of the input values for one iteration. You can insert different placeholders by clicking the “Insert” button. The provided placeholders are:

- [xy]: The received value for a selected input xy.
- [Linebreak]: A linebreak. (Linebreaks will not be inserted automatically between iterations).
- [Timestamp]: Date and time when the inputs were received .
- [Execution count]: Execution count of the Output Writer.

File handling: Here you can select one of the following options:

- Append: The standard option where the inputs of all iterations are written to the same file.
- Auto-Rename: For each iteration, a new file will be created for each iteration.
- Override: Like Auto-Rename, but the file from the previous iteration will be overwritten, so you have one file that only contains the inputs from the last iteration.

12. Parametric Study Component

12.1. Synopsis

The Parametric Study Component is for running a workflow through a set of values for one input variable.

12.2. Rationale

The component starts at a given value and puts it into the workflow. When the workflow is finished, another value is given to the workflow, which is the first value plus a given step size. This iterates until a third given value, the upper bound, is reached. The resulted data can be seen if an input variable for the component is defined.

12.3. Usage

To use the parametric study component you need to define the range and the step size, the study will iterate over. There are two different ways to do that. Either by defining the parameters pre execution or by using inputs that will define the parameters during execution.

Pre Execution

The settings to define the study parameters before execution can be found in the properties tab of the parametric study. There is a pre-defined endpoint named Design Variable which has three metadata values. By clicking on the endpoint and then on the "Edit"-Button, these values may be defined.

During Execution

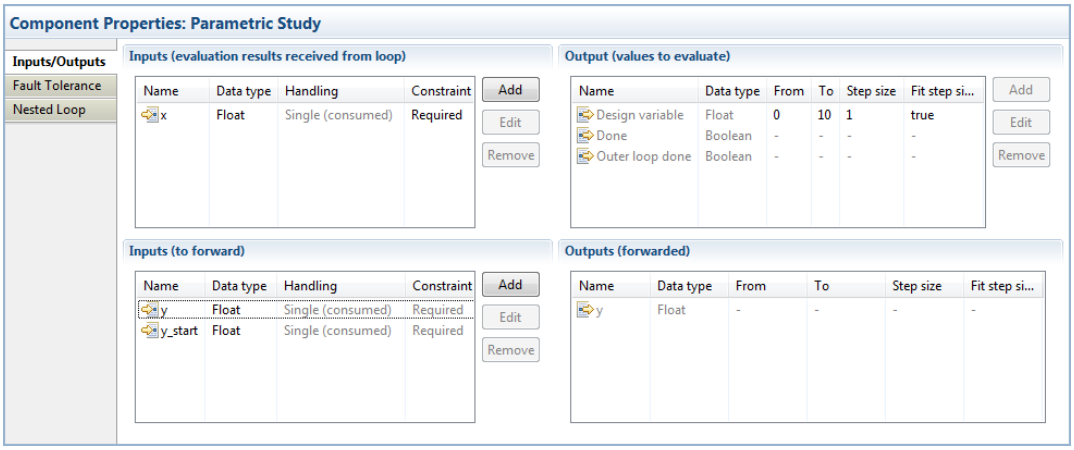
At the same place where you can define the metadata values you may choose to use an input for the values. If you do so, more inputs will appear that must be connected to a providing output. Note that if you want to use the parametric study in a nested loop and define the settings via inputs the parametric study must run in passiv mode (i. e. an input to *Inputs(to forward)* or *Inputs(evaluation results received from loop)* must be added).

The “from” value is the start point of the iteration and will be the first value to be send out. After that, the step size will be added to the last sent value and be the next one. This will be done, until the next to be sent out value is bigger than the “to” value.

There is an option called "Fit step size to bounds". This option takes the given step size, but then sets it to the nearest step size so that the upper bound is never overstepped but the last value will be exactly the upper bound.

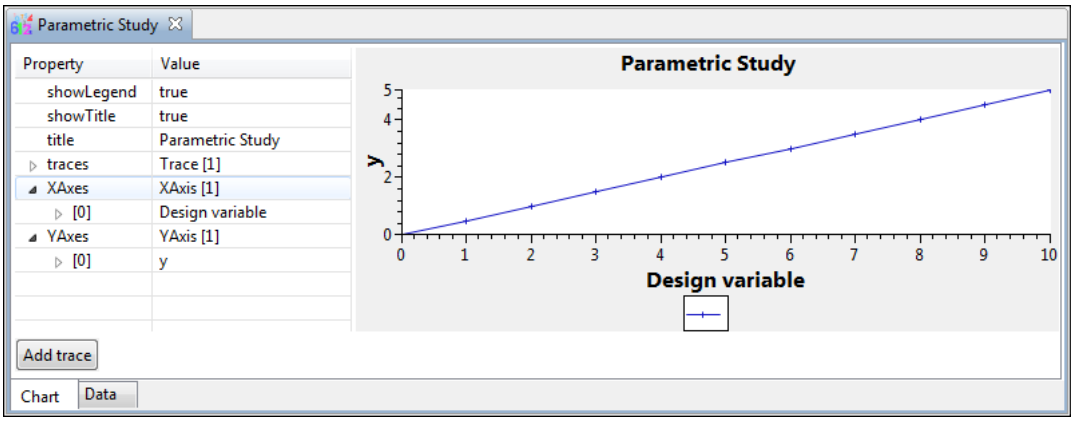
Note that the step size must always be positive. If the "to" value is smaller than the "from" value, the step size is still positive, but will decrease the steps internally.

For all three options "from", "to" and "step size", it is possible to declare them to be defined from an input. For that, the "use input" checkbox must be checked. Then an input for the selected option will be created, which will receive the value for the option during the workflow execution.



Having these values defined, there are two possible modes for running the component.

If you define a new dynamic endpoint in the parameters tab and connect it to other components, the study will send out the values subsequently. Meaning, a new value is sent out as soon as the study receives a response at the newly defined endpoint (except for the first value, which gets sent without the need for a response). This makes it possible to plot a graph when the workflow is started, because for every sent out value there is a corresponding incoming value.



The other mode is active, if no parameter was defined. In this case, all values in the study range are sent out in the first run of the component, so the next component will have all input at the same time. With this, a graph like in the first mode is not possible.

Output

In a running workflow, you are able to see the result from a workflow run by double clicking the component in the Parametric Study view. There you can plot a graph with the results or see them in a table in the data tab.

For help concerning nested and fault-tolerant loop settings, see the general section "Workflows" in the user guide.

13. SCP Input Loader Component

13.1. Synopsis

The SCP Input Loader is a helper component for building workflows that will be published for remote access. This is the point where the Remote Workflow Access feature sends the provided inputs into your workflow. You can change the data types or add/delete inputs in the properties view of the input loader.

14. SCP Output Collector Component

14.1. Synopsis

The SCP Output Collector Component Usage is a helper component for building workflows that will be published for remote access. This is the point where the Remote Workflow Access feature and collects the final outputs from your workflow. You can change the data types or add/delete outputs in the properties view of the output collector.

15. Script Component

15.1. Synopsis

The Script component allows the execution of a self-written script. Currently, two script languages are supported:

- Python: must be installed on the executing system
- Jython [http://www.jython.org/] : a Java implementation of Python.

15.2. Rationale

Based on the selection, the component uses either a natively installed Python version or the Java implementation Jython. This approach was selected on purpose because many users use their own specific modules and want to use Python for this, others are satisfied with the standard Python operations and need a faster implementation which is Jython.

Limitations:

Only single file scripts are allowed, because the user's script contents are converted into a temporary "wrapped" script, executed in the executor’s temporary directory. It is currently not possible to copy satellite files like modules or input data files to the directory where the script is residing.

The execution speed of the Python implementation is dominated by the initial start-up time of the Python interpreter (or virtual machine, just-in-time compiler). Each script execution first wraps the user script into a temporary script file, then starts the Python executable in a new process and after that, processes the output bindings.

Advantages of native Python:

- 100% binary compatibility
- Exotic setups are supported automatically, including third-party modules, binary libraries, `cpython` and so on. Users gain the benefit of using additionally installed Python modules like `<<numpy>>` , `<<scipy>>` or `<<mysqldb>>`
- Self-compiled Python interpreters with binary extensions can be used
- No problems with library indexing as e.g. in Jython

15.3. Usage

15.3.1. Python Executable

There are two options for using Python as script language, the (old) "Python" and the "Python (Python Agent)" option.

If the "Python" option was chosen, the path to the Python executable must be chosen at workflow start (in the second page of the workflow executiondialog). This must be done for every Script component of the workflow. If all components shall use the same Python interpreter, the “Apply to all” button helps.

"Python (Python Agent)" is a new experimental implementation for using Python as script language that aims to improve on the (old) "Python" option and will replace it in the future. If you want to use the Python Agent, the path to the Python executable must be configured in RCE configuration file. For further information please see chapter 2.2 of the RCE User Guide.

15.3.2. Script API

For interacting with RCE in the script execution, there is an API. All methods of this API are listed here .

How to use the script API:

Define your inputs and outputs in the Inputs/Outputs tab of the Properties view (appearing for the selected Script component on double click). Write your script in the Script tab (in the same Properties view). You can either do it in the text box or in a separate text editor by clicking on the button “Open in Editor”. For interacting with RCE from a script, there is a module called "RCE". To get an overview of all RCE API methods, look at the script API detailed description The most important methods there are reading inputs and writing outputs. For reading an input, call the method

```
RCE.read_input(String input_name)
```

or

```
RCE.read_input(String input_name, default value)
```

You can write outputs within your script with

```
RCE.write_output(String output_name, OutputDataType
    value)
```

Thereby, the type (OutputDataType) of the value must fit the data type of the output (as defined in the tab Inputs/Outputs). File and Directory are represented by the absolute file paths.

> **Note**
>
> The module RCE uses is already imported in the script during execution.

Examples:

If you like to double an incoming value (x is an input of type Integer and y an output of type Integer):

```
RCE.write_output("y", RCE.read_input("x")*2)
```

If you like to access an incoming file (f_in is an input of type File):

```
file = open(RCE.read_input("f_in"),"r")
```

If you like to send a file to an output (f_out is an output of type File):

```
absolute_file_path = /home/user_1/my_file.txt
    RCE.write_output("f_out", absolute_file_path)
```

If an output is not needed any more (e.g. you want to end an inner loop), you can close an output using the command:

```
RCE.close_output(String output_name)
```

Example:

```
RCE.close_output(“y”)
```

The following components will get the finished signal.

If a script fails because of some invalid parameters sent by a Parametric Study or Optimizer component, you can send a "not a value" signal to your output(s). This signal indicates that the script failed because of invalid parameters and did not fail at all. This signal is ignored by most of the components, only the Parametric Study and the Optimizer component handle this signal. For sending it, use

```
RCE.write_not_a_value_output(String outputname)
```

For the other API methods refer to the example workflow "Script_with_all_API_methods.wf" from the workflow examples project or to the script reference found below and in the user guide.

15.3.3. Script component states

The Script component is able to keep its state from one run to another. Use the API to write and read state variables. The values are stored in a Python dictionary. They must be compatible with the RCE data types. Script components of nested loops are reset if the nested loop has been terminated. Resetting a script component in a nested loop also resets its state map.

15.3.4. Input File Factory

The Input File Factory is an extension of the Script API that aims to write Python input parameter files during a workflow run. To call the Input File Factory the user must first create a file using the command `file = RCE.create_input_file()`. Afterwards the user can add variable declarations, comments or Python dictionaries by calling the previously created file (e.g. `file.add_variable(name,value)`). Finally,the stored data must be written to the file system by executing the command `write_to_file(filename)`, whereat the name of the file is the given *filename*.

Note

It is not allowed to enter a relative or absolute path for *filename*.

When the Input File Factory is called from an integrated tool, the file is written to the working directorie's *Input directory* or to the tool directory depending on the user configuration. When a script component calls the factory, the file is written to the temp directory. Use the *RCE_write_output* command to forward the file in a workflow.

Example Script:

Assume that the Script Component receives an input called "float" with value 1.0.

```
f = RCE.read_input("float")       // read input

input_file = RCE.create_input_file()      // create an empty input file

input_file.add_comment("This is an example input file")   // add comment
input_file.add_variable("float",f)      // add float variable
input_file.add_dictionary("exampleDict")     // create empty dictionary
input_file.add_value_to_dictionary("exampleDict","key1","value1") // add (key,value) pair to
 dictionary
input_file.add_value_to_dictionary("exampleDict","key2","value2")

file = input_file.write_to_file("example.py")       // write input file to data management
```

To foward the file in a workflow to an output called "file", use:

```
RCE.write_output("file", file)
```

The written file looks like this:

```
# This is an example input file
float = 1.0
exampleDict = {'key1': 'value1', 'key2': 'value2'}
```

15.4. Script API Reference

Method	Description
`def RCE.close_all_outputs ( )`	Closes all outputs that are known in RCE
`def RCE.close_output (name)`	Closes the RCE output with the given name
`def RCE.fail (reason)`	Fails the RCE component with the given reason
`def RCE.get_execution_count ( )`	Returns the current execution count of the RCE component
`def RCE.get_input_names_with_datum ( )`	Returns all input names that have got a data value from RCE
`def RCE.get_output_names ( )`	Returns the read names of all outputs from RCE
`def RCE.get_state_dict ( )`	Returns the current state dictionary
`def RCE.getallinputs ( )`	Gets a dictionary with all inputs from RCE
`def RCE.read_input (name)`	Gets the value for the given input name or an error, if the input is not there (e.g. not required and it got no value)
`def RCE.read_input (name,defaultvalue)`	Gets the value for the given input name or returns the default value if there is no input connected and the input not required
`def RCE.read_state_variable (name)`	Reads the given state variables value, if it exists, else None is returned
`def RCE.read_state_variable (name,defaultvalue)`	Reads the given state variables value, if it exists, else the default value is returned and stored in the dictionary
`def RCE.write_not_a_value_output (name)`	Sets the given output to "not a value" data type
`def RCE.write_output (name,value)`	Sets the given value to the output "name" which will be read from RCE
`def RCE.write_state_variable (name,value)`	Writes a variable *name* in the dictionary for the components state
`def RCE.create_input_file ()`	Creates and returns a file from the input file factory Syntax: file = RCE.create_input_file ()
`def add_variable (name,value)`	Adds the variable declaration of *name* (i.e. name = value) to the input file Syntax: file.add_variable(name, value)
`def add_comment(value)`	Adds a comment (i.e. # value) to the given file Syntax: file.add_comment(value)
`def add_dictionary (name)`	Defines an empty Python dictionary with the given *name* (i.e. name = {}) and adds it to the input file. Note: The data type of *name* has to be String. Syntax: file.add_dictionary(name)
`def add_value_to_dictionary (dic,key,value)`	Writes a *(key,value)* pair (i.e. dic[key] = value) to the dictionary *dic* into the input file. Note: An

Method	Description
	empty dictionary with the given name *dic* has to be defined beforehand. Syntax: file.add_value_to_dictionary(dic, key, value)
`def write_to_file (filename)`	Writes a previously created input *file* to the temp, working or tool dir, depending on the user configurations, and returns the path to the file. The name of the written file is the given *filename* . The component will fail with an error, if a file with the given *filename* already exists. Note: The data type of *filename* has to be String. An input *file* must first be created using the RCE.create_input_file () method. Syntax: filepath = file.write_to_file(filename)
`def write_to_file (filename,overwriteFile)`	Writes a previously created input *file* to the temp, working or tool dir, depending on the user configurations, and returns the path to the file. The name of the written file is the given *filename* . The boolean parameter *overwriteFile* is optional. If set to *True* , an existing file with the given *filename* will be overwritten. The default value is *False* . Note: The data type of *filename* has to be String. An input *file* must first be created using the RCE.create_input_file () method. Syntax: filepath= file.write_to_file(filename, True)

16. Switch Component

16.1. Synopsis

The Switch component forwards input values to one or more outputs. Whether or not an input is forwarded to a specific output depends on user-specified conditions.

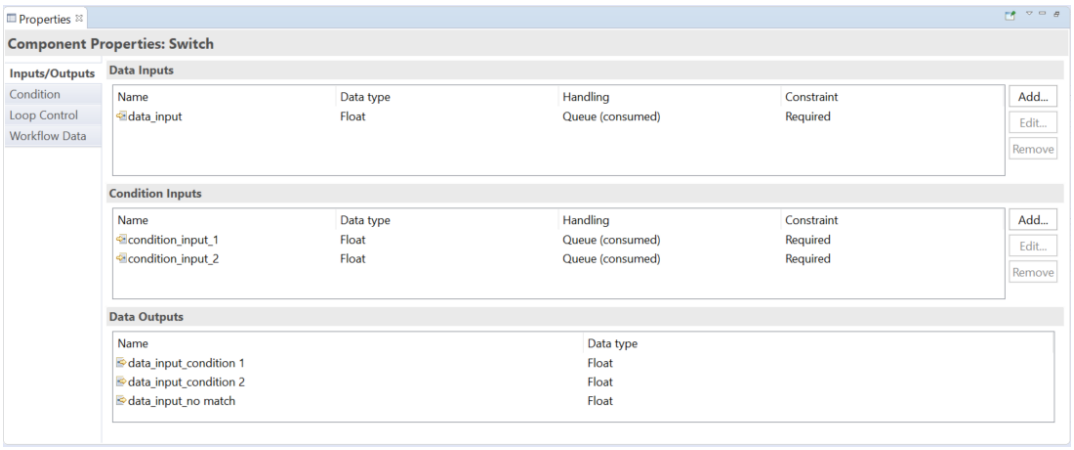
16.2. Rationale

The Switch component allows to direct the data flow within a workflow. It receives input values (so-called data input) and forwards them to at least one output (so-called data output). It depends on user-specified conditions to which output the values are forwarded. Each condition may evaluate the data input values as well as so-called condition inputs. These inputs can be defined but will not be forwarded.

16.3. Usage

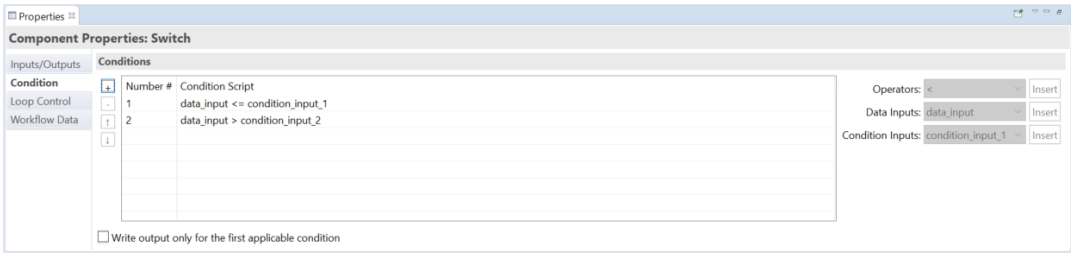
On the 'Inputs/Outputs' tab the user can define the data inputs to be forwarded. The default data type of a data input is float. Change the data type to the one you require to forward. For each data input several data outputs will be generated automatically. The quantity depends on the number of conditions which can be defined on the 'Condition' tab. For one condition, two outputs are created. One output is used to forward the input value in case the condition is true, the other in case the condition is false. For more than one condition there will be one output for each condition plus one output in case no condition evaluates to true. If several conditions evaluate to true, the default behavior is that all input values are written to all related outputs. Optionally the user can decide to write the values only to one related output.

The naming scheme for the automatically created data outputs is as follows: `<data input name>_condition <condition_number>` and `<data_input_name>_no match`. In addition to the data input, you can create condition inputs. They can only be used within the condition and will therefore not be forwarded. The permissible data types are integer, float or boolean.



Define your conditions on the 'Condition' tab. A condition may contain data inputs, condition inputs, numbers, relational operators and parentheses. You can insert inputs and valid operators by hand or you can select them from the drop down list on the right-hand side. Inputs can only be used within conditions, if their data type is either integer, float or boolean. Only inputs of permissible data types are shown in the drop down list. You can add or remove conditions using the plus and minus buttons on the left-hand side.

You can control to which output the values will be written in case several conditions hold true via the option 'Write values only for the first applicable condition'. If this option is set, the data inputs will be forwarded only to the outputs related to the first applicable condition. Otherwise the values will be written to all outputs related to conditions that evaluate to true. You can adjust the order of the conditions via the arrow buttons on the left-hand side.



There are three options available that define the behavior of the Switch component within loops. In the 'Loop Control' tab you can choose between:

- Never close outputs: Use this option if the switch component is used outside of a loop or if it is not supposed to control the loop.
- Close outputs on condition number: Select a condition number from the provided drop down menu. Use this option if the switch component is supposed to control a loop. All inputs of successive components in the workflow which are connected to any output ending with `_condition <condition number>` will be closed. Note that a component is finished if all of its inputs are closed.
- Close outputs if there is no match: Use this option if the switch component is supposed to control a loop. All inputs of successive components in the workflow which are connected to the outputs ending with `_no match` will be closed. Note that a component is finished if all of its inputs are closed.

17. TiGL Viewer Component

17.1. Synopsis

The TiGL Viewer component opens a TiGL Viewer within RCE during workflow execution to provide an integrated 3D viewer for CPACS geometries. In addition, it also allows opening standard CAD file formats like IGES, STEP and BREP.

Note
The TiGL Viewer component is currently only available on Windows operating systems.

17.2. Setup

The TiGL Viewer is not included in RCE anymore, but has to be downloaded and installed separately. Please visit https://dlr-sc.github.io/tigl/pages/download.html in order to obtain a copy. Once installed, add the following lines to your configuration file and restart RCE:

```
"thirdPartyIntegration": {
    "tiglViewer" : {
        "binaryPath" : "/path/to/tiglViewer.exe"
    }
}
```

For additional configuration options for the TiGL Viewer refer to the configuration reference.

17.3. Usage

The TiGL Viewer component has no properties at all. It has a preconfigured input channel and a preconfigured output channel “TiGL Viewer File”.

During workflow execution, the view “TiGL Viewer” will be opened during the first component run. In the following iterations the geometry shown in the view will be updated with the current input.

For more information about the TiGL Viewer software, please visit http://tigl.sourceforge.net/Doc/tiglviewer.html.

18. XML Loader Component

18.1. Synopsis

The XML Loader component loads an XML file from a project within the workspace into the workflow.

18.2. Usage

In the 'File' tab you can load an XML file into the component by clicking on the "Load... " button and navigating to the file of your choice. The content of the XML file will be stored within the workflow and is shown in the text box below so you can quickly verify that you chose the correct file:

> **Note**
> Changing the file itself does not change the loaded content. You have to load the file again to apply changes.

18.2.1. Writing values into an XML file

To map single values into the XML dataset you have to add one input channel per value to be mapped. If you require more complex or conditional mappings or transformations please refer to the XML Merger component. In the "Add Input" dialog click "XPath choosing... " and navigate to an XML file with the same structure as the one you referenced in the 'File' tab, preferably the same file. In the appearing "XPath Variables Dialog" window navigate along the tree to the node you desire to change and select it.

> **Note**
> Search for the XPathChooser help in the User Guide (3. Usage -> 2. Workflows -> 2.3 Workflow components) to garner further information about the usage of XPaths.

When the component is executed the value being received on this input channel will be written to the XPath location defined here. The resulting XML file with the inserted values is written to the output "XML". You can check the original and the outgoing XML files in the workflow data browser.

18.2.2. Reading values from an XML file

To read single values from an XML file add a dynamic output. In the "Add Output" dialog click "XPath choosing... " and navigate to an XML file with the same structure as the one you referenced in the 'File' tab, preferably the same file. In the appearing "XPath Variables Dialog" window navigate along the tree to the node you desire to read and select it. When the component is executed the content of the node that the XPath points to is written into the output channel.

19. XML Merger Component

19.1. Synopsis

The XML Merger component merges the XML content of two inputs on the basis of user-defined rules. These rules can be described in the XML or the XLST format and can either be sent to the component as an input file or can be configured in the component's properties view.

19.2. Rationale

The basic functionality of the XML Merger component is to merge two XML data sets into one as follows: There is a 'main'-XML data set (input channel "XML"). The complete XSLT mapping will be described regarding this XML dataset. All integrating parts of another XML data set (input channel "XML to integrate") will be described with a 'document'-reference in XSLT mapping.

As XSLT is a standard technology in the field of information technology it is referred to corresponding literature available. As a technical background, the approach used is based on Saxon processor. In the mapping file (see example) a XSLT-constant INTEGRATING_INPUT refers to the XML to integrate data channel.

Example: XSLT mapping:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" exclude-result-prefixes="xsi">

    <xsl:output method="xml" version="1.0" encoding="UTF-8" indent="yes"/>

    <!--Define Variable for integrated CPACS-->
    <xsl:variable name="cpacsIntegratedDoc" select="'INTEGRATING_INPUT'"/>

    <!--Copy complete Source to Result -->
    <xsl:template match="@* | node()">
        <xsl:copy>
            <xsl:apply-templates select="@* | node()"/>
        </xsl:copy>
    </xsl:template>
    <xsl:template match="/cpacs/aircraft/configuration/trajectories/global">
        <global>
            <ReferenceTrajectoryUID>
                <xsl:value-of select="document($cpacsIntegratedDoc)/cpacs/aircraft/configuration/
trajectories/global/ReferenceTrajectoryUID"/>
            </ReferenceTrajectoryUID>
            <xsl:copy-of select="document($cpacsIntegratedDoc)/cpacs/aircraft/configuration/
trajectories/global/ReferenceTrajectory"/>
            <xsl:copy-of select="/cpacs/aircraft/configuration/trajectories/global/*"/>
        </global>
    </xsl:template>
</xsl:stylesheet>
```

Instead of using XSLT, the mapping rules can also be described in XML.

Example: XML mapping:

```
<?xml version="1.0" encoding="UTF-8"?>

<map:mappings xmlns:map="http://www.rcenvironment.de/2015/mapping" xmlns:xsl="http://www.w3.org/1999/
XSL/Transform">

 <map:mapping>
     <map:source>/cpacs/vehicles/aircraft/model/reference/area</map:source>
     <map:target>/toolInput/data/var1</map:target>
 </map:mapping>
 <map:mapping mode="delete">
```

```
            <map:source>/toolOutput/data/result1</map:source>
            <map:target>/cpacs/vehicles/aircraft/model/reference/area</map:target>
        </map:mapping>

</map:mappings>
```

RCE automatically determines which format is used based on the filename endings, so the mapping file name must end with ".xslt" or ".xml", respectively.

19.3. Usage

On the “Mapping” tab you can choose whether you want to send the mapping file to the XML Merger component via an input (in this case, the component has an input "Mapping file", to which the file has to be sent during the workflow) or if you want to load a mapping file into the component. In the second case, the content of the mapping file will be stored within the workflow, which will lead to a larger workflow file. Editing the mapping content allows you to edit and store the corresponding content in the workflow (the originally loaded file itself will not be changed).

You can define the scheduling of the static input channels on the tab “Inputs/Outputs”. You are also able to add dynamic input or output channels here. Thus it is possible to map single values into the XML dataset via XPATH declaration. Moreover, you can select single values in order to write them in an output channel.

Note
Search for the XPathChooser help in the User Guide (3. Usage -> 2. Workflows -> 2.3 Workflow components) to garner further information about the usage of XPaths.

20. XML Values Component

20.1. Synopsis

The XML Values Component is capable of reading and writing values within an XML file. These values are declared via dynamic in- and outputs.

20.2. Usage

You can add dynamic inputs to map single values into the XML dataset. Using the XPath declaration, the exact position for the value in the XML file can be chosen. Non-existent elements within the XPath declaration will be generated. The output "XML" contains the changed XML file. Furthermore, you can add additional outputs to select single values from the XML dataset to write them to an output channel.

Note

Search for the XPathChooser help in the User Guide (3. Usage -> 2. Workflows -> 2.3 Workflow components) to gain further information about the usage of XPaths.