User Guide

Build 10.8.1.202610141259

Table of Contents

1. Preface 1
    1.1. Abstract 1
    1.2. Intended Audience 1
    1.3. License Information 1
    1.4. Compatible Operating Systems 1
        1.4.1. Support of 32 Bit Operating Systems 2
    1.5. Updates/Security Information 2
    1.6. Known Issues 2
        1.6.1. KDE on Red Hat Enterprise Linux 7 2
        1.6.2. KDE with Oxygen 2
        1.6.3. Jython scripts are executed sequentially 2
        1.6.4. 32-bit Java is not supported 3
2. Setup 4
    2.1. Installation on Windows 4
        2.1.1. Prerequisites 4
        2.1.2. Obtaining the Signing Key 4
        2.1.3. Installation 5
        2.1.4. Starting RCE as a GUI Client 5
        2.1.5. Starting a Non-GUI ("Headless") Instance 5
        2.1.6. Installation as a Service on a Windows Server 6
    2.2. Configuration and Profiles 6
        2.2.1. Choosing the Profile Directory 6
        2.2.2. Contents of the Profile Directory 8
        2.2.3. Configuration Parameters 8
        2.2.4. Configuration UI 15
            2.2.4.1. Remote Access: SSH account configuration 16
            2.2.4.2. Mail: SMTP server configuration 16
        2.2.5. Importing authorization data without GUI access 17
            2.2.5.1. Importing or deleting RCE authorization group keys 17
            2.2.5.2. Importing SSH Uplink passwords or keyfile passphrases 18
            2.2.5.3. Importing SSH Remote Access passwords or keyfile passphrases..... 18
3. Usage 19
    3.1. Command Line Parameters 19
    3.2. Graphical User Interface 21
    3.3. Workflows 23
        3.3.1. Rationale 24
        3.3.2. Getting Started 24
        3.3.3. Workflow Components 24
        3.3.4. Coupling Workflow Components 25
        3.3.5. Execution Scheduling of Workflow Components 26
        3.3.6. (Nested) Loops 27
        3.3.7. Fault-tolerant Loops 29
        3.3.8. Manual Tool Result Verification 29
    3.4. User-Defined Components 30
        3.4.1. Integrating External Tools as Components 30
            3.4.1.1. Directory Structure for Integrated Tools 31
            3.4.1.2. Copying of Integrated Tools 32
            3.4.1.3. Integration of CPACS Tools 33
            3.4.1.4. Known Issues 37
        3.4.2. Integrating Workflows as Components (Experimental) 37
            3.4.2.1. Integrating a Workflow via command console 38
            3.4.2.2. Workflow Integration Editor 40
            3.4.2.3. Technical Details of Executing an Integrated Workflow 41
            3.4.2.4. Limitations, Caveats, and FAQ 42
    3.5. Connecting RCE instances 43

3.5.1. RCE Network Connections 44
3.5.2. Uplink Connections 44
3.5.2.1. Configuring an RCE instance as an Uplink relay 45
3.5.2.2. Configuring an RCE instance as an Uplink client or gateway (in GUI mode) 45
3.5.2.3. Configuring an Uplink Gateway in non-GUI mode 45
3.5.2.4. Tool publishing 46
3.5.2.5. Possibly surprising behavior (or non-behavior) 46
3.5.2.6. Known issues/limitations of the current release 46
3.5.3. Example of a Cross-Organization Network 46
3.5.4. SSH Remote Access Connections 47
3.5.4.1. Configuring an RCE instance as an SSH server 48
3.5.4.2. Configuring an RCE instance as an SSH client 48
3.6. Tool publishing and authorization 48
3.6.1. Managing authorization groups 49
3.6.2. Publishing tools on the command console 49
3.7. RCE's Command Console 50
3.7.1. Commands 50
3.7.2. Parameters 57
3.7.2.1. Configuration Placeholder Value Files 57
3.8. The Event Log 58
3.8.1. Design Principles 58
3.8.2. Event Structure and File Representation 58
3.8.3. Events Types and Attributes 59
A. Script API Reference 72

List of Figures

2.1. Profile Selection UI 7
2.2. Configuration tool for SSH account and SMTP server configuration 16
3.1. Workbench with different views and the workflow editor opened 21
3.2. Connection Editor 22
3.3. Network View 22
3.4. Workflow Data Browser 23
3.5. Workflow Console 23
3.6. Run process of an user-integrated CPACS Tool 35
3.7. Workflow for determining the optimal input for the function f_c(x). 37
3.8. Workflow from the above figure prepared for integration as a component. 39
3.9. Workflow Integration Editor 41
3.10. Mapping Page 41
3.11. Example RCE network 47

List of Tables

2.1. "general" 8
2.2. "backgroundMonitoring" 9
2.3. "network" 9
2.4. "componentSettings" 11
2.5. "thirdPartyIntegration" 11
2.6. "sshServer" 11
2.7. Possible roles for SSH accounts 12
2.8. "uplink" 13
2.9. "sshRemoteAccess" 14
2.10. "smtpServer" 15
3.1. Command line arguments for RCE 19
3.2. Data Type Conversion Table 26
3.3. Inputs of Optimizer 27
3.4. Outputs of Optimizer 28
3.5. Inputs of Design of Experiments 28
3.6. Outputs of Design of Experiments 28
3.7. Inputs of Parametric Study 28
3.8. Outputs of Parametric Study 28
3.9. Inputs of Converger 28
3.10. Outputs of Converger 29
3.11. Connection types - feature matrix 43

List of Examples

2.1. Choosing the Profile Directory 7

1. Preface

This chapter gives an introduction to RCE.

1.1. Abstract

RCE (Remote Component Environment) is an open source software that helps engineers, scientists and others to create, manage and execute complex calculation and simulation workflows. A workflow in RCE consists of components with predefined inputs and outputs connected to each other. A component can be a simulation tool, a tool for data access, or a user-defined script. Connections define which data flows from one component to another. There are predefined components with common functionalities, like an optimizer or a cluster component. Additionally, users can integrate their own tools. RCE instances can be connected with each other. Components can be executed locally or on remote instances of RCE (if the component is configured to allow this). Using these building blocks, use cases for complex distributed applications can be solved with RCE.

1.2. Intended Audience

The intended audience of this document consists of engineers, scientists, and everybody else interested in developing automated workflows with RCE.

1.3. License Information

RCE is published under the Eclipse Public Licence (EPL) 1.0. It is based on Eclipse RCP 4.8.0 (Photon), which is also published under the Eclipse Public Licence (EPL) 1.0. RCE also makes use of various libraries which may not be covered by the EPL; for detailed information, see the file "THIRD_PARTY" in the root folder of an RCE installation. (To review this file without installing RCE, open the RCE release .zip file.)

For downloads and further information, please visit https://rcenvironment.de/.

1.4. Compatible Operating Systems

RCE releases are provided for Windows and Linux. It is regularly tested on

- Windows 10
- Windows Server 2019
- CentOS 8
- Debian 11
- Ubuntu 20.04 LTS

and should also run on Mint 10.04 and SUSE Linux Enterprise Server 15 SP2.

1.4.1. Support of 32 Bit Operating Systems

Starting with release 8.0.0, RCE is only shipped for 64 bit systems. If you still require 32 bit packages, you can continue to use previous RCE releases, but there will be no standard feature or bugfix updates for them.

1.5. Updates/Security Information

As third-party software (e.g. libraries) included in RCE is constantly updated, it is generally recommended to always upgrading RCE to the latest available release. However, not every update might be strictly necessary. To allow RCE users and administrators to make informed decisions, the page "Update/Security Information for RCE Releases" [https://rcenvironment.de/pages/updatessecurity.html] provides an overview of the status of all public RCE releases. In particular, this includes information about how important it is to update from a certain RCE version.

1.6. Known Issues

1.6.1. KDE on Red Hat Enterprise Linux 7

On Red Hat Enterprise Linux 7 with KDE 4, RCE (like any other Eclipse-based application) can cause a segmentation fault at startup. If you encounter such an issue, you can try choosing a different GTK2 theme:

1. Open the **System Settings** application (systemsettings).
2. Go to **Application Appearance**
3. Open **GTK** page
4. Switch the GTK2 theme to "Raleigh" or "Adwaita" and click on **Apply**

1.6.2. KDE with Oxygen

On Unix Systems using KDE as desktop environment and Oxygen as theme it can happen that RCE crashes when certain GUI elements are shown. It is a known issue in the theme Oxygen and happens on other Eclipse-based applications as well. If you encounter such an issue, please choose a different theme like "Raleigh" or "Adwaita".

1.6.3. Jython scripts are executed sequentially

The Script component can use Jython for the evaluation of scripts and the pre- and postprocessing of integrated tools always uses Jython. Due to a known bug in the Jython implementation it is not possible to execute several Jython instances in parallel. Therefore, the execution will be done sequentially. If several Script components should be executed in parallel, Python should be used instead.

1.6.4. 32-bit Java is not supported

Running RCE with a 32-bit Java Runtime Environment doesn't work. On some operating systems an error dialog will be displayed in this case, on some other systems nothing will happen at all. Therefore, always make sure a 64-bit Java Runtime Environment is used to run RCE.

2. Setup

This section describes the installation and configuration of RCE.

2.1. Installation on Windows

2.1.1. Prerequisites

To run RCE on a system, the only prerequisite is an installed Java Runtime Environment (JRE), version 11 or above. If you do not already have one on your machine, you can download it from e.g.

```
https://adoptium.net/temurin/releases
```

and install it. Starting with RCE 8.x we only publish 64 bit packages of RCE. Therefore, please make sure to install the 64 bit version of the JRE.

Note

Some pre-installed components of RCE have additional dependencies. Please refer to Section 2.3 (Workflow Components) for more details.

2.1.2. Obtaining the Signing Key

Any software can be tampered with by a malicious attacker. For RCE, the consequences of such tampering may be worse than with other software, since its intended behavior already includes executing arbitrary processes, opening outgoing network connections, and listening for incoming ones. One common safeguard against such tampering is software signing. If the developers sign a software artifact, e.g., a zip-archive or an executable file, the user can verify the signature. This verification confirms that the artifact downloaded onto their machine is identical to the artifact prepared by the software developers and has not been tampered with.

In order to sign a software artifact, the developers combine the artifact and a so-called signing key to form a signature file. The user can then verify the signature using the artifact, the signature file, and a part of the signing key that can only be used for verification, but not for signing. A technical introduction to the minutiae of software signing is out of the scope of this user guide and we refer to the literature for more information on this topic.

While verifying the downloaded artifact is optional, we strongly recommend doing so.

In order to verify the signature of a software, you require

- the artifact that you want to verify (in this case a .zip-file)
- a signature file provided by the signer (in this case provided by us)
- the verification part of the signing key.

You can find the former two items at https://rcenvironment.de under the menu item `Download`. The signature file is called `SHA256SUM.asc` and is available in the same directory as the artifacts. The latter item, i.e., the signing key, is not available via https://rcenvironment.de or linked to in this user guide on purpose: Recall that the purpose of software signing is to protect against compromised communication channels between developers and users. Thus, if the artifact, the signatures, and the

signing key were available at the same location, an attacker that takes control over that location could easily forge all three components.

One common way to distribute such keyfiles is via so-called public keyservers. We had previously published the signing key for RCE at the SKS-keyservers, which are no longer available. Nevertheless, we have published the keyfile via https://github.com in the repository called `rce-signing` owned by the organization `rcenvironment`. For now, this is the key supplier to use until we determine a new public key server. Please verify the integrity of the obtained keyfile by checking its fingerprint against the one published by us via a trusted channel. We omit giving direct links as well as the key's fingerprint here on purpose. This slightly decreases the chance of attackers directing you toward a forged key.

The precise steps required for signature verification differ from system to system. Commonly, key retrieval and verification is handled by gpg4win (available at https://www.gpg4win.org). Please refer to its documentation in order to verify your downloaded software artifact.

2.1.3. Installation

On Windows, we provide a single .zip-file to set up both client and server installations. This file is available at

```
https://download.rcenvironment.de/rce/10.x/products/standard/releases/latest/zip/
```

Installing RCE amounts to simply extracting the zip file to a location on your file system.

> **Note**
>
> Due to restrictions of the standard Windows file system (NTFS), you should choose a destination path that is as short as possible. Long paths are known to cause problems, as some RCE files may not be properly extracted from the zip file.

2.1.4. Starting RCE as a GUI Client

To use RCE with a graphical user interface (GUI), simply start the "rce" executable from Windows Explorer. Optionally, create a desktop icon from the right-click menu using the "send to > desktop" option.

Once your RCE instance has started, you can open the configuration file with the menu option "Configuration > Open Configuration File". Edit the file, save it, and then restart RCE using the "File > Restart" menu option to apply the changes. There are configuration templates and other information available via the "Configuration > Open Configuration Information" option. The available configuration settings are described in the configuration chapter.

2.1.5. Starting a Non-GUI ("Headless") Instance

RCE can also be run from the command line without a graphical user interface (which is called "headless" mode), which uses less system resources and is therefore recommended when the GUI is not needed.

To run a headless RCE instance, open a command prompt and run the command

```
rce --headless -console
```

While RCE is running, you can enter various console commands described in Section 3.1, “Command Line Parameters”; note that you need to prefix all RCE commands with "rce" here. To perform a clean shutdown, for example, type `rce stop` and press enter.

2.1.6. Installation as a Service on a Windows Server

Please refer to the section "RCE as a Windows Service" in the "RCE Administration and Security Guide" to install RCE as a service.

2.2. Configuration and Profiles

Each instance of RCE reads its configurations from the *profile directory*, or *profile* for short. In this document, if the distinction is irrelevant or clear from context, we use these terms interchangeably. On Windows systems, the profile is, by default, located in the *main RCE directory* `C:\Users\<username>\.rce\default`.

If this directory or any of its parent directories do not exist, RCE creates them at first startup and populates them with a default configuration. RCE creates the directory `.rce` as a hidden directory. Thus, you will have to enable the display of hidden directories in order to see that directory.

2.2.1. Choosing the Profile Directory

Since RCE uses the profile directory to load its configuration at startup, it is not possible to change the profile directory at runtime. Instead, when running RCE via the commandline, you can use the commandline switch `-p` to change the profile at startup. Currently, there is no way to select the profile via the graphical interface of RCE.

You can temporarily (i.e., for a single execution of RCE) change the profile directory by supplying its path as a parameter for the commandline switch `-p`. If you supply a relative path (i.e., one not starting with `C:\` or some other drive name), RCE resolves that path relative to its main directory, i.e., to `C:\Users\<username>\.rce`. If you supply an absolute path, RCE treats that path as the path to its profile directory.

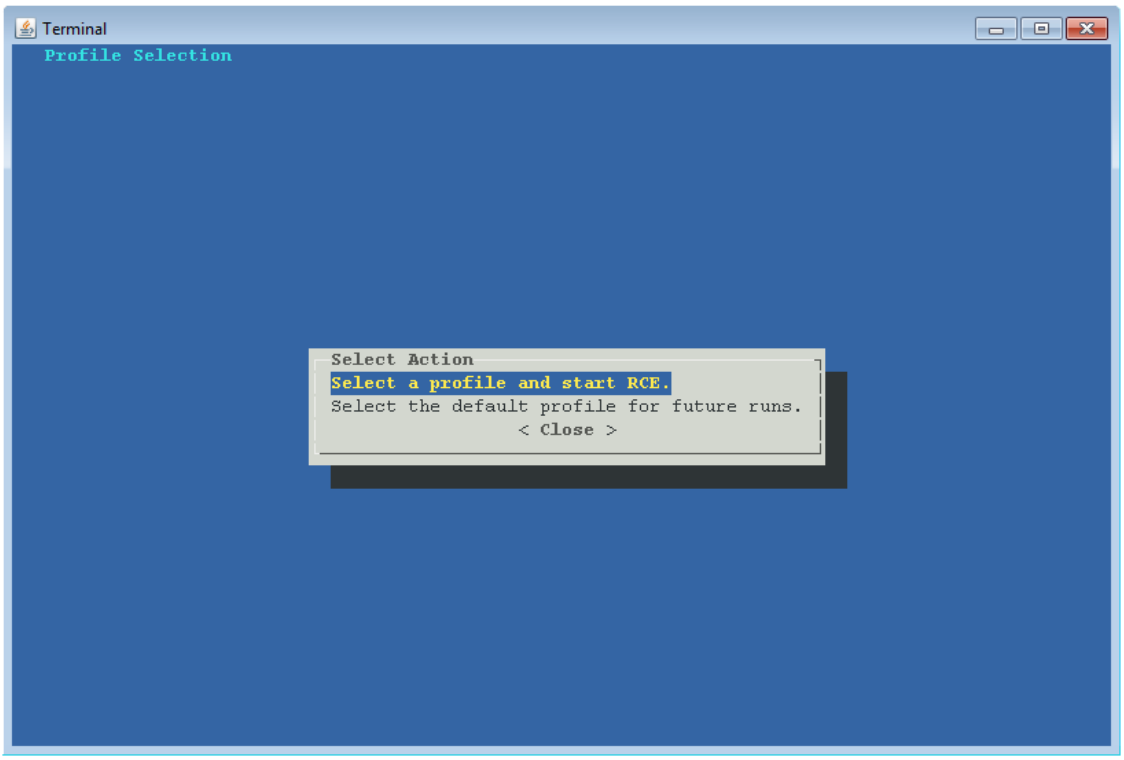
Note

You can change the path to which RCE resolves relative profile directories by editing the file rce.ini in the main folder of your RCE installation. Add the line `-Drce.profiles.parentDir="<directory>"` and replace `<directory>` with the absolute path to a directory which contains your profiles.

In either case, RCE creates the profile directory and its parents if they do not exist. The range of valid profile names is mainly restricted by your OS's range of valid directory names. We strongly recommend to restrict your profile names to the range of printable ASCII characters. Furthermore, the profile name `common` is reserved for internal use of RCE.

If you want to change the profile location permanently, you can do so by supplying the commandline switch `-p` without a parameter. In this case, RCE starts with a text-based profile selection UI which allows you to set a new location of the default profile. We show this profile selection UI in the following figure.

Figure 2.1. Profile Selection UI



Choosing the second option ("Select the default profile for future runs") will present a list of available profiles. On selection of one of these profiles, RCE will not be started using this profile, but instead the selected profile will be marked as the default profile for future runs. This selection is only stored for the current user and for the current installation location of RCE. Different users on the same machine can therefore configure different default profiles. Furthermore, different installations of RCE can have different default profiles configured.

> **Note**
>
> The Profile Selection UI will only display profiles if they have been started once with RCE 7.0 or newer.

No two instances of RCE may use the same profile simultaneously. Upon startup, RCE checks whether some other instance of RCE is already running which uses the given profile. In this case, RCE displays an error message stating that it "failed to lock profile directory" and shuts down. If you want to start multiple instances of RCE simultaneously on the same machine, you have to specify a different profile for each instance.

Example 2.1. Choosing the Profile Directory

Install RCE on some system that has not had RCE installed previously. For the sake of this example, we assume the current user to be called `rceuser`.

Start RCE without any parameters (i.e., via `rce.exe`). RCE will use the profile `C:\Users\rceuser\.rce`. Execute that command a second time in a second cmd while the first instance is still running. Startup of RCE will fail, since the profile is already used by the first instance of RCE.

Now execute the command `rce.exe -p secondinstance` in a second cmd. You will have two instances of RCE running, one using the profile `C:\Users\rceuser\.rce\default`, the other using the profile `C:\Users\rceuser\.rce\secondinstance`.

Finally, execute the command `rce.exe -p`. RCE will offer you a list of profiles that you have most recently used. Pick the profile `secondinstance` as the default profile. If you now close all running instances of RCE and restart RCE without any parameters (i.e., via `rce.exe`), RCE will start using the profile `secondinstance`.

2.2.2. Contents of the Profile Directory

As stated above, each profile contains the static configuration of RCE as well as its integrated tools. The former is defined via the JSON-file `configuration.json`. For an in-depth introduction to the JSON file format, please refer to https://www.json.org/ [http://www.json.org/]. The file `configuration.json` is located in the root of the profile directory, while the integrated tools are defined in the directory `integration` and its subdirectories.

To change the static configuration of RCE, change the contents of the file configuration.json and restart RCE. You can either change the contents of this file manually using your favorite text editor, or from the GUI of RCE via the toolbar or the menu entry *Configuration*.

Note

Recall that RCE only parses its configuration during startup. Thus, if you change the contents of the file `configuration.json`, you have to restart RCE to apply these changes.

2.2.3. Configuration Parameters

Configuration parameters are grouped within the configuration file. The configuration parameters are listed below. There is one list per JSON configuration group. Some example snippets are given as well. The complete example configurations can be found in the installation data directory in the sub-directory *examples/configuration* or by opening the configuration information in RCE.

Table 2.1. "general"

Configuration key	***Comment***	***Default value***
instanceName	The name of the instance that will be shown to all users in the RCE network. The following placeholders can be used within the instance name: • ${hostName} is resolved to the local system's host name. • ${systemUser} is resolved to the user account name. • ${profileName} is resolved to the last part of the current profile's file system path. • ${version} is resolved to the build id. • ${javaVersion} is resolved to the JRE version number. Example: "Default instance started by \"${systemUser}\" on ${hostName}".	"<unnamed instance>"
isWorkflowHost	If set to *true*, the local instance can be used as a *workflow host* by other RCE instances. I.e., the workflow controller can be set to this instance and the workflow data is stored there as well.	false
isRelay	If set to *true*, the local node will merge all connected nodes into a single network, and forward messages between them. This behaviour is transitive; if a relay node connects to another relay node, both networks will effectively merge into one.	false

Configuration key	*Comment*	*Default value*
	If set to *false* (the default value), the local node can connect to multiple networks at once without causing them to merge.	
tempDirectory	Can be used to override the default path where RCE stores temporary files. Useful if there is little space in the default temp file location. Must be an absolute path to an existing directory, and the path must not contain spaces (to prevent problems with tools accessing such directories). The placeholder *${systemUser}* can be used for path construction, e.g. "/tmp/custom-temp-directory/${systemUser}"	An "rce-temp" subdirectory within the user or system temp directory.
enableDeprecatedInputTab	If set to *true* the tab 'Inputs' is enabled again in the properties view of running workflows. It is disabled by default due to robustness and memory issues. It is recommended to use the 'Workflow Data Browser' to see inputs received and outputs sent.	false

Table 2.2. "backgroundMonitoring"

Configuration key	*Comment*	*Default value*
enabledIds	Comma-separated list of identifiers referring to certain kind of monitoring data that should be logged continuously in the background. Currently, only 'basic_system_data' is supported.	
intervalSeconds	Logging interval	10

Table 2.3. "network"

Configuration key	*Comment*	*Default value*
requestTimeoutMsec	The timeout (in milliseconds) for network requests that are made by the local node. If this time expires before a response is received, the request fails.	40000
forwardingTimeout Msec	The timeout (in milliseconds) for network requests that are forwarded by the local node on behalf of another node. If this time expires before a response is received, an error response is sent back to the node that made the request.	35000
connections	A map of all connections that the local instance tries to establish on startup. This allows the local instance to act as a client. For each connection a unique identifier (id) must be given.	{} (an empty map in JSON format)
connections/[id]/host	IP address of the host to connect to. Host names and IPv4 addresses are permitted.	-
connections/[id]/port	Port number of the remote RCE instance.	-
connections/[id]/ connectOnStartup	If set to *true*, the connection is immediately established on startup.	true
connections/[id]/ autoRetryInitialDelay	The initial delay, in seconds, to wait after a failed or broken connection before a reconnect attempt is made.	-

Configuration key	*Comment*	*Default value*
	This configuration must be present to enable the auto-reconnect feature.	
connections/[id]/ autoRetryDelayMultiplier	A decimal-point value >= 1 that the delay time is multiplied with after each consecutive connection failure. This provides an "exponential backoff" feature that reduces the frequency of connection attempts over time. This configuration must be present to enable the auto-reconnect feature.	-
connections/[id]/ autoRetryMaximumDelay	Defines an upper limit for the delay time, even when applying the multiplier would create a higher value. This can be used to maintain a minimum frequency for retrying the connection. This configuration must be present to enable the auto-reconnect feature.	-
serverPorts	A map of all server ports that the local instance registers for other instances to connect to. This allows the local instance to act as a server. For each server port a unique identifier (id) must be given.	{} (an empty map in JSON format)
serverPorts/[id]/ip	The IP address for other instances to connect to.	-
serverPorts/[id]/port	The IP port number (1024-65535) for other instances to connect to.	-
ipFilter	Allows to limit the incoming connections to a set of IP addresses.	-
ipFilter/enabled	If set to *true*, the ip filter active.	false
ipFilter/allowedIPs	List of IP addresses, which are allowed to connect to the instance.	[] (an empty list in JSON format)

Note

IMPORTANT: When setting up a network of RCE instances, keep in mind that the RCE network traffic is currently *not encrypted*. This means that it is *not secure* to expose RCE server ports to untrusted networks like the internet. When setting up RCE connections between different locations, make sure that they either connect across a secure network (e.g. your institution's internal network), or that the connection is secured by other means, like SSH tunneling or a VPN. Alternatively, you can set up an uplink connection in RCE instead of the standard RCE connections.

Network Server Sample:

```
"network" : {
  "serverPorts" : {
   "relayPort1" : {
    "ip" : "127.0.0.1",
    "port" : 21000
   }
  },
  "ipFilter" : {
   "enabled" : false,
   "allowedIPs" : [
    "127.0.0.1",
    "127.0.0.2"
   ]
  }
 }
```

Network Client Sample:

```
"network" : {
  "connections" : {
   "exampleConnection1" : {
    "host" : "127.0.0.1",
```

```
    "port" : 21000,
    "connectOnStartup": false,
    "autoRetryInitialDelay" : 5,
    "autoRetryMaximumDelay" : 300,
    "autoRetryDelayMultiplier" : 1.5
   }
  }
 }
```

Table 2.4. "componentSettings"

Configuration key	*Comment*	*Default value*
de.rcenvironment. cluster	Configuration of the cluster workflow component.	-
de.rcenvironment. cluster/ maxChannels	Maximum number of channels, which are allowed to be opened in parallel to the cluster server.	8

Table 2.5. "thirdPartyIntegration"

Configuration key	*Comment*	*Default value*
tiglViewer	Configuration of the external TiGL Viewer application integration. This needs to be configured to enable RCE´s TiGL Viewer view and thus, the TiGL Viewer workflow component. Note:TiGL Viewer must be downloaded and installed separately.	-
tiglViewer/binaryPath	The path to the TiGL Viewer executable file. Must be an absolute path.	-
tiglViewer/ startupTimeoutSeconds	The timeout in seconds, to wait for the external TiGL viewer application to start and determine its process id.	10
tiglViewer/embedWindow	If set to *false*, the external TiGL Viewer application Window will not be embeded into RCE´s TiGL Viewer view.	true
python	Configuration of a external Python installation. This needs to be configured to enable Python script language for the Script Component. Note: Python must be downloaded and installed separately.	-
python/binaryPath	The path to a local python installation. This needs to be configured to enable Python script language for the Script Component	-

Third Party integration Python path example:

```
"thirdPartyIntegration": {
   "python":{
   "binaryPath": "/path/to/python/executable"
     }
   }
```

Table 2.6. "sshServer"

Configuration key	*Comment*	*Default value*
enabled	If set to *true* the local instance acts as an SSH server.	false
ip *(deprecated alias: "host")*	The host's ip address to bind to. If you want to make the SSH server accessible from remote, you should set	127.0.0.1

Configuration key	*Comment*	*Default value*
	this to the IP of the machine's external network interface. Alternatively, you can set this to "0.0.0.0" to listen on all available IPv4 addresses, if this is appropriate in your network setup.	
port	The IP port number (1024-65535) for SSH or Uplink clients to connect to.	-
idleTimeoutSeconds	The time to keep an idle SSH connection alive, in seconds. For typical SSH usage, the default value is usually sufficient. Higher values are, for example, needed when invoking long-running tools using the SSH Remote Access feature.	10
accounts	A map of accounts. For each account a unique identifier (account name) must be given.	{} (an empty map in JSON format)
[account name]/passwordHash	The hashed password for the account, if password authentication is used. If the SSH account is configured using the configuration UI, the hash is automatically computed and stored here.	-
[account name]/password *(deprecated)*	The password for the account. SSH passwords can also be configured as plain text, which is however not recommended. To prevent misuse of the configured login data, any configuration file with SSH accounts *must* be secured against unauthorized reading (e.g. by setting restrictive filesystem permissions). A more secure alternative is to just store the password hash.	-
[account name]/publicKey	The public key for the account, if keyfile authentication is used. Only RSA keys in the OpenSSH format are supported. The public key has to be entered here in the OpenSSH format (a string starting with "ssh-rsa", like it is used for example in authorized_keys files). Only applicable on RCE version 7.1 or newer.	-
[account name]/role	The role of the account. See next table for a list of the possible roles.	-
[account name]/enabled	If set to *true*, the account is enabled.	true

SSH Server Sample:

```
"sshServer" : {
  "enabled" : true,
  "ip" : "127.0.0.1",
  "port" : 31005,
  "accounts" : {
   "ra_demo" : {
    // hashed form of the "ra_demo" test password; DO NOT reuse this for live accounts!
    "passwordHash" : "$2a$10$qxCBuEvq0xWoOlox2dVbCu8zCYsyxQMBe5SAnS2HId0uaEp59aAu2",
    "role" : "remote_access_user",
    "enabled" : true
   }
  }
 }
```

Table 2.7. Possible roles for SSH accounts

Role name	*Allowed commands*
uplink_client (Standard role for using Uplink connections)	Cannot open a command shell or run any commands

Role name	***Allowed commands***
remote_access_user (Standard role for using SSH remote access tools and workflows)	ra|sysmon (can use Uplink connections)
remote access (backwards compatibility alias for remote_access_user)	ra|sysmon (can use Uplink connections)
remote_access_admin	ra|ra-admin|sysmon|components
workflow_observer	components|net info|sysmon|wf list|wf details
workflow_admin	components|net info|sysmon|wf
local_admin	cn|components|mail|net|restart|shutdown|stop|stats|tasks|auth
instance_management_admin	im|net info|auth
instance_management_delegate_user	cn|components|net|restart|shutdown|stop|stats|tasks|wf|ra-admin|auth
developer	<all>

Table 2.8. "uplink"

Configuration key	***Comment***	***Default value***
uplinkConnections	A map of Uplink connections.This allows the local instance to act as an Uplink client. For each connection a unique identifier (id) must be given.	{} (an empty map in JSON format)
uplinkConnections/[id]/displayName	The name for the connection that will be shown in the network view.	-
uplinkConnections/[id]/host	The remote RCE instance (Uplink relay) to connect to. Host names and IPv4 addresses are permitted.	-
uplinkConnections/[id]/port	Port number of the remote RCE instance.	-
uplinkConnections/[id]/loginName	The login name for authentication.	-
uplinkConnections/[id]/keyfileLocation	Path to the private key file, if keyfile authentication is used. Only RSA keys in the OpenSSH format are supported.	-
uplinkConnections/[id]/noPassphrase	This option should only be set if a private key that requires no passphrase is used for authentication. If set to *true*, RCE does not ask for a passphrase before connecting.	false
uplinkConnections/[id]/clientID	If other RCE instances use the same account to connect to the relay, you have to set a unique client ID here (max. 8 characters)	default
uplinkConnections/[id]/isGateway	If set to *true*, this instance will act as an Uplink gateway (see chapter Section 3.5.2, “Uplink Connections” for further information)	false
uplinkConnections/[id]/connectOnStartup	If set to *true*, the connection is immediately established on startup. (Only possible when the password is stored.)	false
uplinkConnections/[id]/autoRetry	If set to *true*, RCE will try to automatically reconnect the connection (every 5 seconds) if it can not be established	false

Configuration key	*Comment*	*Default value*
	or is lost of a network error. (Only possible when the password is stored in the secure store.)	

Uplink Connection Sample:

```
"uplink" : {
  "uplinkConnections" : {
   "exampleUplinkConnectionID" : {
    "displayName" : "example",
    "clientID": "client1",
    "host" : "127.0.0.1",
    "port" : 31005,
    "connectOnStartup": false,
    "autoRetry" : false,
    "isGateway" : false,
    "loginName" : "ra_demo"
    //The passphrase is not stored here, it has to be entered when connecting.
   }
  }
 }
```

Table 2.9. "sshRemoteAccess"

Configuration key	*Comment*	*Default value*
sshConnections	A map of SSH connections.This allows the local instance to act as a SSH remote access client. For each connection a unique identifier (id) must be given.	{} (an empty map in JSON format)
sshConnections/[id]/displayName	The name for the connection that will be shown in the network view.	-
sshConnections/[id]/host	The remote RCE instance to connect to. Host names and IPv4 addresses are permitted.	-
sshConnections/[id]/port	Port number of the remote RCE instance.	-
sshConnections/[id]/loginName	The login name for authentication.	-
sshConnections/[id]/keyfileLocation	Path to the private key file, if keyfile authentication is used. Only RSA keys in the OpenSSH format are supported.	-
sshConnections/[id]/noPassphrase	This option should only be set if a private key that requires no passphrase is used for authentication. If set to *true*, RCE does not ask for a passphrase before connecting.	false
sshConnections/[id]/connectOnStartup	If set to *true*, the connection is immediately established on startup. (Only possible when the password is stored in the secure store.)	false
sshConnections/[id]/autoRetry	If set to *true*, RCE will try to automatically reconnect the connection (every 10 seconds) if it can not be established or is lost of a network error. (Only possible when the password is stored in the secure store.)	false

Remote Access Connection Sample

```
"sshRemoteAccess" : {
    "sshConnections" : {
     "exampleSSHConnection" : {
      "displayName" : "example",
      "host" : "127.0.0.1",
      "port" : 31005,
      "connectOnStartup": false,
```

```
      "autoRetry" : false,
      "loginName" : "ra_demo"
      //The passphrase is not stored here, it has to be entered when connecting.
     }
    }
   }
```

Table 2.10. "smtpServer"

Configuration key	*Comment*	*Default value*
host	The IP address or hostname of the SMTP server, which should be used for mail delivery.	-
port	Port number of the SMTP server.	-
encryption	Can either be "explicit" or "implicit". Select "implicit" if you want to connect to the SMTP server using SSL/TLS. Select "explicit" if you want to connect to the SMTP server using STARTTLS. Unencrypted connections are not permitted.	-
username	The login name for authentication.	-
password	The obfuscated password for authentication. Plaintext password cannot be used here. To create the obfuscated password from the plaintext password, you need to use the Configuration UI described in Section 2.2.4, “Configuration UI”	-
sender	Email address, which should be displayed as the sender in the sent email.	-

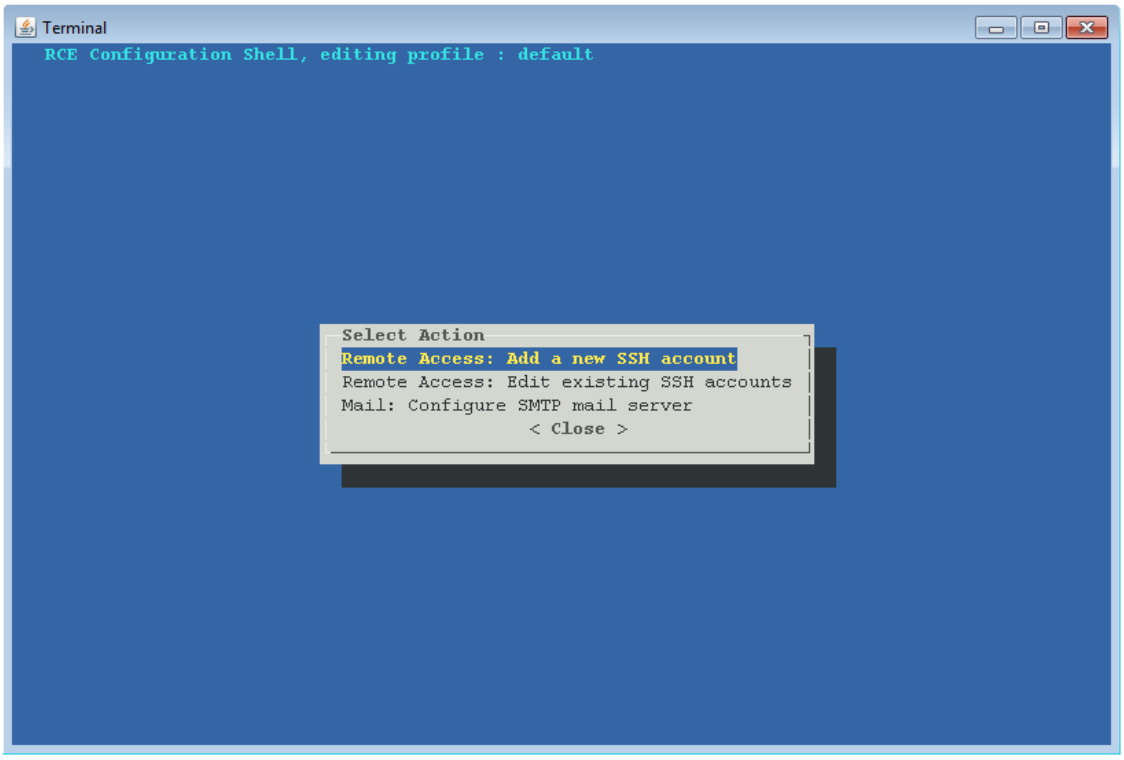
Note

The used SMTP server needs to be configured using the Configuration UI described in Section 2.2.4.2, “Mail: SMTP server configuration”, since the password needs to be obfuscated.

2.2.4. Configuration UI

If you want to configure SSH accounts with passphrases or you want to configure e-mail support for the instance, you need to use the Configuration UI. You can access the interactive tool by executing RCE from the command line with the option "rce --configure" or by using the "Launch Configuration UI" script in the "extras" folder of your RCE installation directory.

Figure 2.2. Configuration tool for SSH account and SMTP server configuration



2.2.4.1. Remote Access: SSH account configuration

If the RCE instance shall act as a SSH server, you can configure SSH accounts using the Configuration UI, which encrypts the SSH passwords before storing them in the configuration file.

> **Note**
>
> All SSH accounts configured with this tool initially have the role "remote_access_user", which allows to execute commands needed for remote access on tools and workflows. If you want to change the role of an SSH account, you can do this by editing the configuration file manually (see Table 2.7, “Possible roles for SSH accounts”).

2.2.4.2. Mail: SMTP server configuration

If you use the tool output verification (cf. Section 3.3.8, “Manual Tool Result Verification”) and want RCE to send the verification key via email, you need to configure an SMTP server. RCE does not send e-mails directly to the recipient, but instead sends the e-mails to an SMTP server, which delivers them to the recipient. You need to use the Configuration UI to configure such an SMTP server, since the password used for authentication needs to be obfuscated before it is stored in the configuration file. The SMTP server parameters that need to be configured are described in more detail in Table 2.10, “"smtpServer"”

Note

Due to a known bug on Windows system with a German keyboard layout, the Configuration UI inserts the characters "q@" into a text field if you want to insert the @ sign. You can manually remove the additional character "q".

2.2.5. Importing authorization data without GUI access

There are currently two categories of authorization data that should not be simply written into configuration files for security reasons: SSH login passwords and keyfile passphrases, and RCE authorization group keys (the "export/import" strings). To support scenarios where interactive entry is not possible, for example daemon/service installations, a file-based import mechanism is provided as well.

The general usage is the same for all kinds of import data:

- Locate the folder of the profile that you want to import into.
- If it does not exist yet, create a folder "`import`" within that profile directory.
- Within this "`import`" folder, create the sub-folder mentioned in the specific description below; for example, "`auth-group-keys`".
- To perform an import, create a file inside this specific sub-folder and edit it, or copy a file that you already prepared into that folder. These files are referred to as "import files". The filenames and contents to use for them are described in the specific sections below.
- Once you have created or copied all import files that you want to processs, (re)start RCE. Currently, all import file processing is done on startup. (Note: Future versions of RCE may be expanded to also detect and process new import files without a restart.)
- If a file has been successfully imported, it is *deleted* to minimize the time that it is present in the filesystem (for security), and to prevent it from being processed again on every RCE start. Make sure that this file is not the only reference to the authorization data that you have!

2.2.5.1. Importing or deleting RCE authorization group keys

This section focusses on importing or deleting *already defined* authorization groups via their group keys. Creating groups is explained in Section 3.6, “Tool publishing and authorization”.

- Group key import files must be placed in `<profile>/import/auth-group-keys/`.
- For group keys, the import files can have any name. For each key you wish to import a single file is required.
- The import file's content must be the group import string; it should look similar to "`MyGroupName:23b0ad9043a39496:1:1K6D5C9BKYu[...]sSMLlj0Tg`".
- Deleting groups is also supported. To delete a group, write "`delete`" into the file, followed by the full id (name + random part) of the group you want to delete. For convenience, you can also use the full import string as used above. For example, if you wanted to delete the group mentioned above, either of these contents would work:
  - "`delete MyGroupName:23b0ad9043a39496`"
  - "`delete MyGroupName:23b0ad9043a39496:1:1K6D5C9BKYu[...]sSMLlj0Tg`"
- After successful import or deletion of a group key, the file is deleted from the profile folder.

2.2.5.2. Importing SSH Uplink passwords or keyfile passphrases

- Uplink password/passphrase import files must be placed in `<profile>/import/uplink-pws/`.
- The names of the import files are relevant: These must be the "connection id" used in the Uplink connection configuration. This id is the string right in front of the the part outside of the connection's configuration block (e.g. `"myConnection" : { ...<connection settings>... }`). For convenience on Windows, a ".txt" extension can be added to this filename; this will be cut away by the importer.
- The content of the files is the password or keyfile passphrase.
- When the profile is launched, the password or keyfile passphrase is imported into RCE's secure storage and the import files are deleted to mitigate potential security risks.
- As this is never actually needed, deleting passwords is not directly supported. If you have imported a password/passphrase you would rather remove from RCE's secure storage, simply import a dummy password for the same connection id. This will overwrite and erase the previous data.

2.2.5.3. Importing SSH Remote Access passwords or keyfile passphrases

- Uplink password/passphrase import files must be placed in `<profile>/import/ra-pws/`.
- The names of the import files are relevant: These must be the "connection id" used in the Remote Access connection configuration. This id is the string right in front of the the part outside of the connection's configuration block (e.g. `"myConnection" : { ...<connection settings>... }`). For convenience on Windows, a ".txt" extension can be added to this filename; this will be cut away by the importer.
- The content of the files is the password or keyfile passphrase.
- When the profile is launched, the password or keyfile passphrase is imported into RCE's secure storage and the import files are deleted to mitigate potential security risks.
- As this is never actually needed, deleting passwords is not directly supported. If you have imported a password/passphrase you would rather remove from RCE's secure storage, simply import a dummy password for the same connection id. This will overwrite and erase the previous data.

3. Usage

This chapter describes the main usage concepts.

3.1. Command Line Parameters

General syntax

```
>rce.exe --[RCE arguments] -[RCP arguments] -[vmargs -[VM arguments]]
```

> **Note**
> To launch RCE as a console application, you can also use `rcec.exe`. The Windows operating system then automatically allocates a console for the application that shows the console output. This executable does not automatically add any arguments, so you still need to specify `-console`, `-consoleLog`, etc.

Table 3.1. Command line arguments for RCE

Argument	Type	Description
profile "<profile id or path>"	RCE	Sets a custom profile folder to use. If only an id (any valid directory name) is given, the profile directory "<user home>/.rce/*id*" is used. Alternatively, a full filesystem path can be specified.
profile	RCE	If the profile argument is specified without a profile id or path, RCE launches the Profile Selection UI, which allows to select a profile folder for the startup as described in ???.
batch "<command string>"	RCE	Behaves like the "exec" command, but also implies the "--headless" option and always shuts down RCE after execution.
headless	RCE	Starts RCE in a headless modus without GUI. It will remain in the OSGi console and waits for user input.
allow-privileged	RCE	Allows running RCE in a privileged process, i.e. as administrator (on Windows) or root (on Linux). Usually this is forbidden and the prohibition is enforced at application startup. However in certain circumstances it might be required to run RCE with such privileges and for these cases this flag can be used to disable the check at startup. However this is only to be used with absolute caution and in accordance with the system administrator!
exec "<command string>"	RCE	Executes one or more shell commands defined by <command string>. For the list of available commands, refer to the command shell documentation. This argument is usually used together with --headless to run RCE in batch mode. Multiple commands can be chained within <command string> by separating them with " ; " (note the spaces); each command is completed before the next is started. You can use the "`stop`" command at the end of the command sequence to shut down RCE after the other commands have been executed. However, any error during execution of these commands will cancel the

Argument	Type	Description
		sequence, and prevent the "`stop`" command from being executed. To ensure shut down at the end of the command sequence, use the `--batch` option instead of "`--exec`". As an example, `rce --headless --exec "wf run example.wf ; stop"` will execute the "example.wf" workflow in headless mode and then shut down RCE. However, if the workflow fails to start, RCE will keep running, as the "`stop`" command is never executed. To attempt execution of the workflow file, but then always shut down regardless of the outcome, use `rce --batch "wf run example.wf"` instead.
configure	RCE	Starts the RCE Configuration UI (Section 2.2.4, “Configuration UI”) which can be used to configure SSH accounts with passphrases or to configure e-mail support for the RCE instance.
data @noDefault	RCP	Set the default workspace location to empty
consoleLog	RCP	Logs everything for log files on the console as well.
console	RCP	Runs RCE with an additional OSGi console window, which allows you to execute RCE shell commands. See the Command Shell documentation for more information.
Deprecated: console <port>	RCP	Specify the port that will be used to listen for telnet connections. (*NOTE:* this access is insecure; configure SSH access instead)
clean	RCP	Cleans before startup
vmargs	VM	Standard JVM arguments. This argument needs to be listed before any arguments intended to be passed to the JVM (i.e. arguments prefixed with -D), otherwise they will not be passed correctly. Also note that the JVM will receive these arguments appended to the ones specified in the rce.ini file (the file itself will not be modified). Example: "rce.exe -vmargs -D<some-arg>=<some-value>"
Dde.rcenvironment.rce.configuration.dir=<insert-config-path>	VM	Sets the configuration directory
Drce.network.overrideNodeId =<some-id>	VM	Sets the local node id, overriding any stored value. This is mostly used for automated testing. Example: "-Drce.network.overrideNodeId= a96db8fa762d59f2d2782f3e5e9662d4"
Dcommunication.uploadBlockSize=<block size in bytes>	VM	Sets the block size to use when uploading data to a remote node. This is useful for very slow connections (less than about 10 kb/s) to avoid timeouts. The default value is 262144 (256 kb). Example:

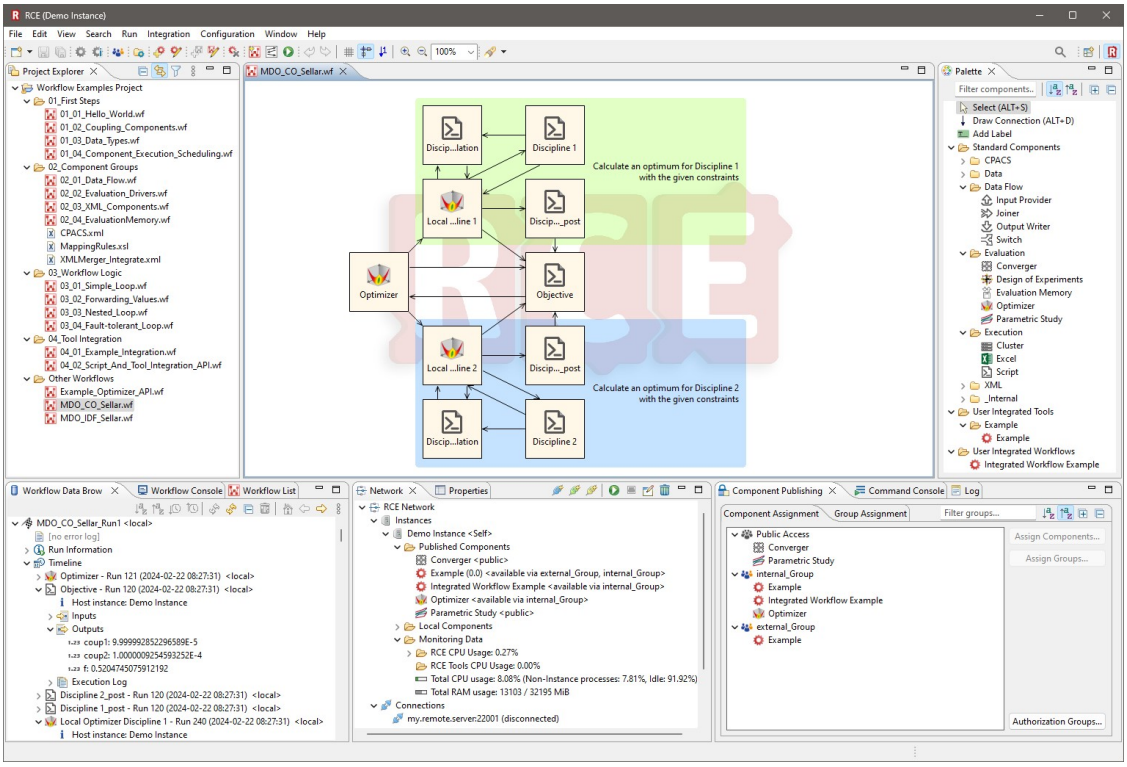
Argument	Type	Description
		"-Dcommunication.uploadBlockSize=131072" - sets the upload block size to 128kb (half the normal size)

3.2. Graphical User Interface

This section introduces the Graphical User Interface (GUI).

The GUI of RCE is composed of different views and editors (besides standard GUI elements such as the menu bar, status bar, etc.). Views can be (re-)arranged by the user. They can even be closed and opened again. Some views are closed by default, but can be opened as desired. To open a view go to: Window → Show view.

Figure 3.1. Workbench with different views and the workflow editor opened



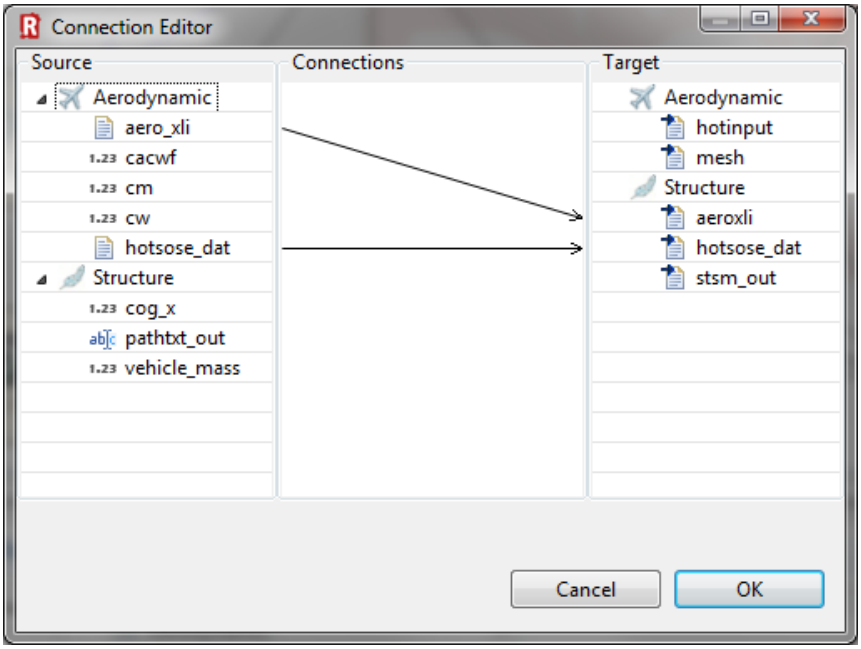
Left hand side:

- *Project Explorer:* View to manage projects. All relevant data including workflow files needs to be organized in projects.
- *Workflow List:* Lists all active workflows and allows to manage them (stop, pause, resume, dispose).

Right hand side and center:

- *Workflow Editor:* Core view of RCE used to build and configure workflows.
- *Palette:* Lists all available workflow components. If RCE runs in a distributed environment this includes local as well as remote workflow components. At the top, it also provides actions for connecting workflow components. We show the connection editor in the following Figure. Additionally, connections of the workflow are shown in the *Properties* view at the bottom, if the background of the workflow editor is selected.

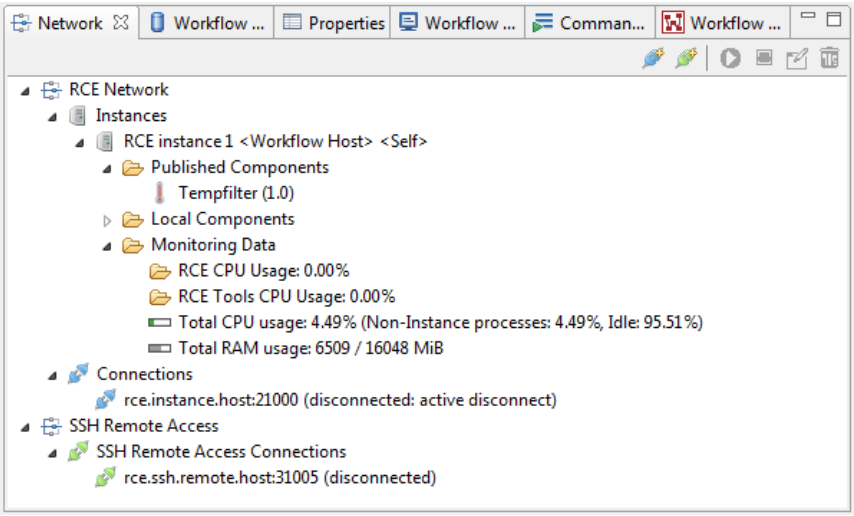
Figure 3.2. Connection Editor



Bottom:

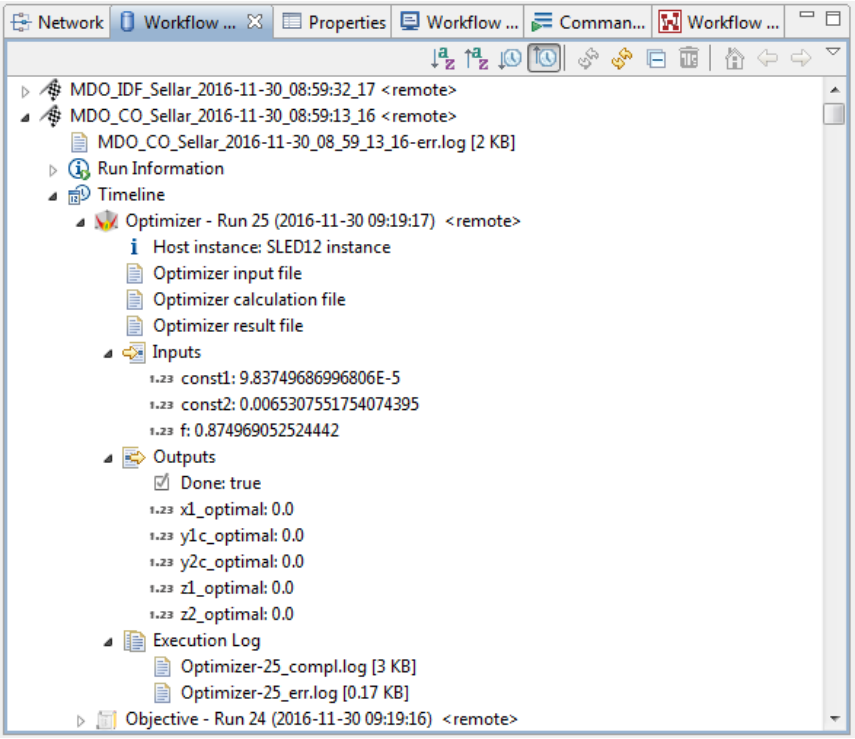
- *Log:* Shows all log output of RCE, e.g. error messages during workflow execution.
- *Network View:* Shows all RCE instances of the distributed RCE network and their published workflow components. It also shows the outgoing connections of the own RCE instance and allows to manage them (start, stop, etc.). Furthermore, you are able to see monitoring data like CPU or RAM usage for each instance.

Figure 3.3. Network View



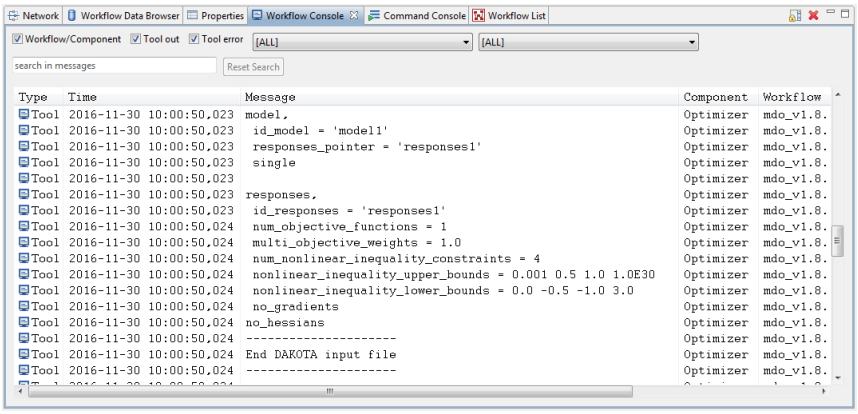
- *Workflow Data Browser:* Shows workflow related result data.

Figure 3.4. Workflow Data Browser



- *Properties:* Allows configuration of workflow components (e.g. Inputs/Outputs) if they are selected in the workflow editor. View adapts to selected workflow component.
- *Workflow Console:* Shows all native console line output of integrated tools during workflow execution. Provides full text search.

Figure 3.5. Workflow Console



3.3. Workflows

This section describes the basics of workflows in RCE.

3.3.1. Rationale

RCE is designed to execute automated, distributed workflows. Workflows consist of so called workflow components which can be coupled with each other. Loops are supported, even multi-nested ones.

3.3.2. Getting Started

To get started with workflows in RCE it is recommended to both read the following sections about workflows and walk through the example workflows provided in RCE. The sections here refer to the workflow examples where it is useful and vice versa.

Workflows in RCE are encapsulated in so called projects. To create the workflow examples project go to: File → New → Workflow Examples Project. A dialog appears. Leave the default project name or give it a name of your choice and confirm by clicking `Finish`. In the *Project Explorer* on the left-hand side, the newly created project is shown. The example workflows are grouped in sub folders. It is recommended to walk through the workflows following the prefix starting with `01_01_Hello_World.wf`.

3.3.3. Workflow Components

Workflow components are either tools that are integrated by users or are provided by RCE supplying multi-purpose functionality. The following list shows workflow components provided by RCE grouped by purpose (workflow components that are deprecated (i.e., they are removed soon) or that are not recommended to use anymore are left out):

- *Data:* Database
- *Data Flow:* Input Provider, Output Writer, Joiner, Switch
- *Evaluation:* Optimizer, Design of Experiments, Parametric Study, Converger, Evaluation Memory
- *Execution:* Script, Cluster, Excel
- *XML:* XML Loader, XML Merger, XML Values
- *CPACS:* TiGL Viewer, VAMPzero Initializer

> **Note**
>
> The Optimizer component uses the Dakota toolkit [https://dakota.sandia.gov/] in order to perform the actual optimization. This toolkit is included in the RCE distribution, i.e., it is installed together with RCE. On some systems, however, notably Ubuntu 18.04, this toolkit cannot be executed, as the required library *libgfortran3* is not installed by default. If the toolkit cannot be executed, the Optimizer component will issue the error `Could not start optimizer. Maybe binaries are missing or not compatible with system.; cause: Optimizer exited with a non zero exit code. Optimizer exit code = 127 (E#1543567120128)` or similar in the workflow console and the data management.
>
> Please refer to the documentation or the administrator of your system in order to satisfy the missing dependency of the Dakota toolkit.

The example workflows in subfolder `02_Component Groups` introduce some of the workflow components provided. Additionally, there is a documentation for each workflow component available in RCE. To access it, you can either rightclick on a component in a workflow and select `Open Help` or press `F1`. A help view opens on the right-hand side. Moreover there is an entry `Help Contents` in the `Help` menu where you can navigate to the component help you require.

The XML and CPACS components are able to read or extract data from an XML file via dynamic in- or outputs. The XPathChooser is a feature that provides help selecting the item, which shall be read or extracted. Add an in- or output and press the `XPath choosing...` button to open a window where

you can select the XML file which contains the item that shall be selected. After choosing the file, the XPathChooser opens containing a tree, symbolizing the XML file. By selecting an element, the text below is updated and displays the current path. The last two columns are used to choose attributes. The attribute name can be selected via the column `Attributes`. In the column `Values` the proper value can be selected. Use a double-click on an element to expand or fold the tree. The chosen XPath will be written in the text field of the window in which the XPathChooser has been opened originally. Using this text field, new paths can be created. Add a slash and the name of the node that shall be created to the existing path. The new path will be added during the workflow run.

New XPaths can only be generated within the inputs tab. Using the outputs tab will cause an error.

3.3.4. Coupling Workflow Components

A workflow component can send data to other workflow components. Therefore, a so called connection needs to be created between the sending workflow component and the receiving one. For that purpose, workflow components can have so called inputs and outputs. A connection is always created between an output and an input. You can think of a connection as a directed data channel. Data is sent as atomic packages which are not related to each other (there is no data streaming between workflow components). Supported data types are:

Primitive data types:

- *Short Text:*A short text (up to 140 characters)
- *Integer:*Integer number
- *Float:*Floating point number
- *Boolean:*Boolean value (true or false)

Referenced data types (The actual data is stored in RCE's data management and only a reference is transfered):

- *File:*File
- *Directory:*Directory

Other data types

- *Small Table:* The RCE syntax for Small Tables is [[a,b,...],[c,d,...],...], whereat the table values a,b,c,d are restricted to values of type Short Text, Integer, Float, Boolean (primitive data types) as well as File and Directory. Be aware, that in case of File and Directory simply the path to the Files or Directories will be stored in the Small Table. Each column holds the same number of entries. The total number of possible cells is up to 100.000.
- *Vector:*one-dimensional "Small Table" (one column) restricted to values of type Float i.e. [x,y,z,...]
- *Matrix:*Small Table restricted to values of type Float

Not all of the workflow components support all of the data types listed. A connection can be created between an output and an input if:

- The data type of the output is the same as or convertible to the data type of the input.
- The input is not already connected to another output.

Note that data from an output can be sent to multiple inputs, but an input can just receive data from a single output.

The following table shows which data types are convertible to which other types:

Table 3.2. Data Type Conversion Table

To / From	Boolean	Integer	Float	Vector	Matrix	Small Table	Short Text	File	Directory
Boolean		x	x	x	x	x			
Integer			x	x	x	x			
Float				x	x	x			
Vector					x	x			
Matrix						x			
Small Table									
Short Text						x			
File									
Directory									

3.3.5. Execution Scheduling of Workflow Components

The execution of workflows is data-driven. As soon as all of the desired input data is available, a workflow component will be executed. Which input data is desired is defined by the component developer (for RCE's default workflow components), the tool integrator, and/or the workflow creator. The workflow component developer and tool integrator decide which options are allowed for a particular workflow component. The workflow creator can choose between those options at workflow design time. The following options exist:

Input handling:

- *Constant:* The value won't be consumed during execution and will be reused in the next iteration (if there is any loop in the workflow). The workflow will fail if there is more than one value received, except for nested loops: All inputs of type *Constant* are resetted within nested loops, after the nested loop has been finished.
- *Single (Consumed):* The input value will be consumed during execution and won't be reused in the next iteration (if there is any loop in the workflow). Queuing of input values is not allowed. If another value is received before the current one was consumed, the workflow will fail. This can guard against workflow design errors. E.g., an optimizer must not receive more than one value at one single input within one iteration.
- *Queue (Consumed):* The input value will be consumed during execution and won't be reused in the next iteration (if there is any loop in the workflow). Queuing of input values is allowed.

Execution constraint:

- *Required:* The input value is required for execution. Thus, the input must be connected to an output.
- *Required if connected:* The input value is not required for execution (e.g., if a default value will be used as fall back within the component). Thus, the input doesn't need to be connected to an output. But if it is connected to an output, it will be handled as an input of type *Required*.
- *Not required:* The input value is not required for execution. Thus, the input doesn't need to be connected to an output. If it is connected to an output, the input value will be passed to the component if there is a value available at the time of execution. Values at inputs of type *Not required* cannot trigger component execution except if it is the only input defined for a component. Note: With this option, non-deterministic workflows can be easily created. Use this option carefully. If in doubt, leave it out.

Note: With RCE 6.0.0 the scheduling options changed. Below is the migration path:

- *Initial* was migrated to *Constant* and *Required*.
- *Required* was migrated to *Single (Consumed)* and *Required*.
- *Optional* was migrated to *Single (Consumed)* and *Required if connected*.

If you encounter any problems with workflows created before RCE 6.0.0, it is likely, that it affects the migration to *Single (Consumed)* instead of to *Queue (Consumed)*. We decided to migrate conservatively to not hide any existing workflow design errors. So, if queuing of input values is allowed for an input, just change the input handling option to *Queue (Consumed)* after the workflow was updated. Another issue can affect the migration of *Optional*. If it affects an input of the script component, check the option, which let the script component execute on each new value at any of its inputs. Also check *Not required* as an alternative execution constraint option.

3.3.6. (Nested) Loops

Workflow components can be coupled to loops. A loop must always contain a so-called driver workflow component. Driver workflow components (group "Evaluation") are: Optimizer, Design of Experiments, Parametric Study, Converger (see the example workflow "02_02_Evaluation_Drivers"). The responsibilities of a driver workflow component in a loop are:

- Send values to the loop and receive the result values.
- Finish the loop based on some certain criteria.

If a loop contains another loop, we speak of the latter as a nested loop. A nested loop can contain again another loop and so on. To create workflows with nested loops (see example workflows in "03_Workflow_Logic"), some certain concepts behind nested loops must be understood:

- Loop level: If a loop contains another loop, that loop is considered as a nested loop with a lower loop level. From the perspective of the nested loop, the other loop is considered as a loop with an upper loop level.
- If a driver workflow component is part of a nested loop, you need to check the checkbox in the "Nested Loop" configuration tab
- Data exchange between loops of different loop levels is only allowed via a driver workflow component. Thereby, only particular inputs and outputs of driver workflow components are allowed to be connected to inputs and outputs of the next upper loop level and particular ones to inputs and outputs of the same loop level. For example, if a 'same loop level' output is connected to a loop with an upper loop level, the workflow won't succeed or might even get stuck. Below you find tables of inputs and outputs for each driver workflow component and whether they must be connected to the same loop level or to the next upper loop level.

> **Note**
> In the inputs and outputs tables of driver workflow components (in 'Inputs/Outputs' properties tab), the loop level requirement is present in a particular column for each input and output.

Table 3.3. Inputs of Optimizer

Input	Loop Level
* - lower bounds - start value	To next upper loop level
* - upper bounds - start value	To next upper loop level

Input	Loop Level
* - start value	To next upper loop level
* (Objective functions)	To same loop level
* (Constraints)	To same loop level
d*.d* (Gradients)	To same loop level

Table 3.4. Outputs of Optimizer

Output	Loop Level
*_optimal	To next upper loop level
Done	To next upper loop level
* (Design variables)	To same loop level
Gradient request	To same loop level
Iteration	To same loop level

Table 3.5. Inputs of Design of Experiments

Input	Loop Level
*_start	To next upper loop level
*	To same loop level

Table 3.6. Outputs of Design of Experiments

Output	Loop Level
Done	To same loop level
*	To same loop level

Table 3.7. Inputs of Parametric Study

Input	Loop Level
*_start	To next upper loop level
*	To same loop level

Table 3.8. Outputs of Parametric Study

Output	Loop Level
Done	To same loop level
*	To same loop level

Table 3.9. Inputs of Converger

Input	Loop Level
*_start	To next upper loop level
*	To same loop level

Table 3.10. Outputs of Converger

Output	Loop Level
Converged	To next upper loop level
Converged absolute	To next upper loop level
Converged relative	To next upper loop level
*_converged	To next upper loop level
Done	To same loop level
*	To same loop level

3.3.7. Fault-tolerant Loops

Workflow components of a loop can fail. There are two kind of failures:

- A workflow component fails gracefully, i.e. it couldn't compute any results for the inputs received but works normally. In this case, it sends a value of type "not-a-value" with the specified cause to its outputs which finally are received by the driver workflow components as results.
- A workflow component fails, i.e. it crashes for an unexpected reason. In this case, the workflow engine sends values of type "not-a-value" with the specified cause as results to the driver workflow component.

In the "Fault Tolerance" configuration tab of workflow driver components, it can be configured how to handle failures in loops, for both kind of failures separately.

3.3.8. Manual Tool Result Verification

After the execution of an integrated tool, the results are sent via outputs to the next workflow component (e.g. to the next integrated tool). By default, this is done in an automated manner without any user interaction. If the data should be verified by a person responsible for the tool before they are sent further, manual verification of tool results must be enabled in the tool integration wizard in the 'Verification' tab of the 'Inputs and Outputs' page.

In case manual verification of tool results is enabled, the results are hold after each tool execution and the corresponding workflow component remains in state "Waiting for approval". Then, there are two options:

- Approve tool results: The tool results are sent via the outputs to the next workflow component and the workflow continues normally.
- Reject tool results: The tool results are not sent via the outputs to the next workflow component and the workflow is cancelled.

To apply one of the options, a so called verfication key is required. The verification key is generated by RCE after each tool execution and is written to a file on the file system of the machine which executed the tool. (The location is specified in the 'Verification' tab of the 'Inputs and Outputs' page in the tool integration wizard.) Optionally, the verification key can also be sent via e-mail if e-mail support is configured for the RCE instance where the tool is integrated. (E-mail support can only be configured using the Configuration UI as described in Section 2.2.4.2, “Mail: SMTP server configuration”) E-

mail delivery can be enabled and the recipients can be defined in the 'Verification' tab of the 'Inputs and Outputs' page in the tool integration wizard.

Once the verification key is known (either from the file or an e-mail), perform follwing steps to approve or reject the tools results:

- Start an RCE instance with a graphical user interface. (Your tool must be available, i.e. it must appear in the palette of the workflow editor.)
- In the menu bar at the top, go to Run -> Verify tool results...
- A dialog appears that guides you through the verification process.

3.4. User-Defined Components

RCE comes with a number of components that already allow you to create rather large and complex workflows. It is technically possible to construct a workflow using only the components that come with RCE, since you can call external tools via the Script-component. Maintaining such a workflow will, however, prove quite cumbersome. Moreover, every instance of the Script-component in an RCE workflow must be configured individually, since there exists no possibility to share configuration between component instances. Finally, while components can be shared among a network of RCE instances (cf. Section 3.6, “Tool publishing and authorization”) this sharing does not extend to configuration values such as the script of a Script-component. To simplify the construction, sharing, and maintenance of workflows containing calls to external tools, RCE allows for the integration of such external tools as user-defined components.

3.4.1. Integrating External Tools as Components

If you want to integrate an external tool, the tool must

- be callable via command line,
- have a non-interactive mode which is called via command line, and
- have its input provided through environment variables, command line arguments, or files

If these requirements are fulfilled, a tool can be integrated into RCE. An *integration file* describes the "interface" of the tool to RCE. This interface consists of, among others, its inputs, outputs, as well as how the tool is executed. The integration file can be found in the profile directory (cf. Section 2.2, “Configuration and Profiles”) in the subdirectory `integration/common/<tool name>`, where `<tool name>` is the name under which your tool will be available as a component in RCE. You can find an example of such an integration file by importing the Workflow Examples Project (via *File -> New -> Workflow Examples Project*) and opting to integrate an example tool during the import.

If you use RCE with a graphical user interface you can integrate a tool via a wizard which guides you through the settings. This wizard can be found in the menu *Tool Integration -> Integrate Tool...*. Required fields are marked with an asterisk (*). When the wizard is finished and if everything is correct, the integrated tool will automatically show up in the Workflow Editor palette.

> **Note**
>
> The wizard has a dynamic help, which is shown by clicking on the question mark on the bottom left or by pressing F1. It will guide you through the pages of the wizard.

One major part of the tool integration consists of the definition of a *pre-*, an *execution-*, and a *post script*. Pre- and post script define how incoming data from RCE will be passed to the tool, and how

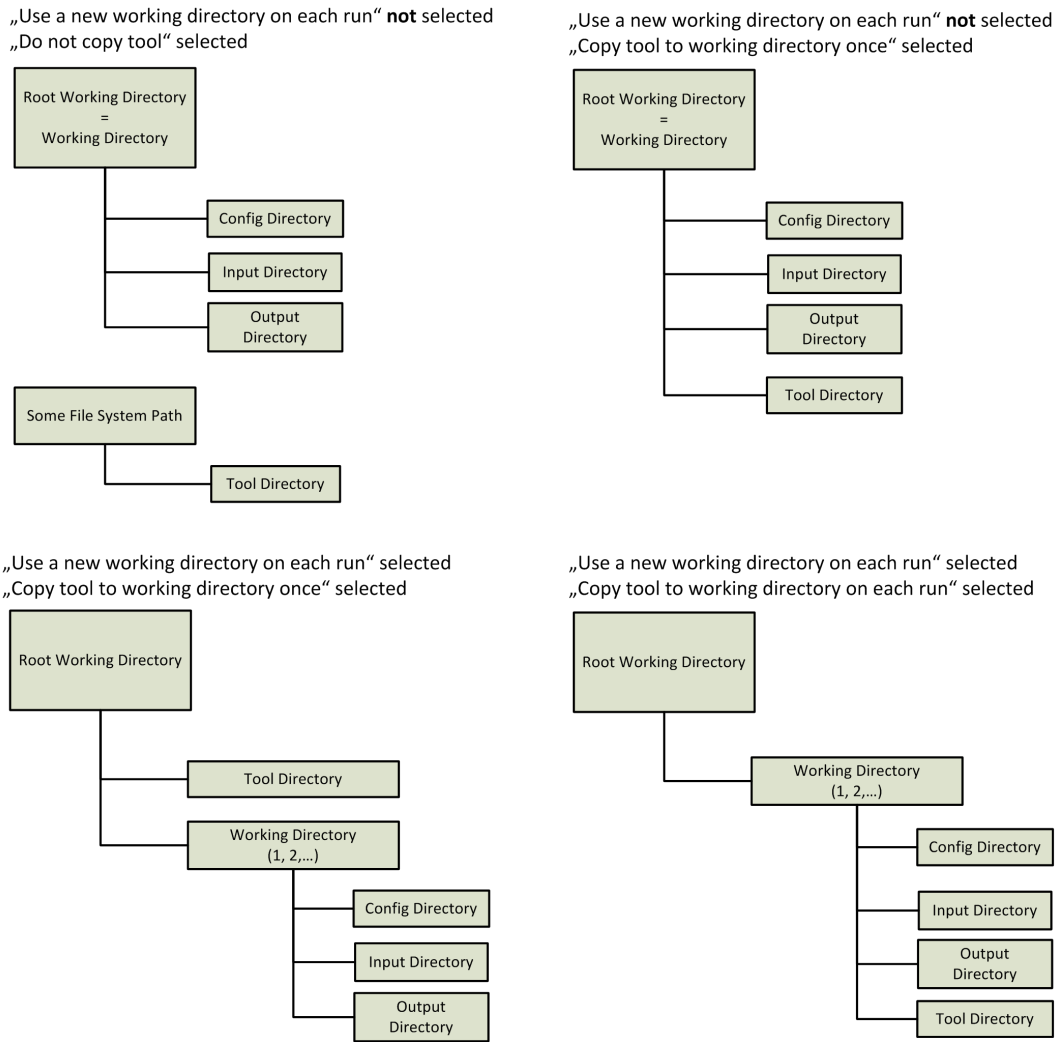
outgoing data from the tool will be passed back to RCE, respectively. These scripts are written in Python and executed using Jython, a Java implementation of Python 2. Please make sure that your pre- and post scripts have the desired behavior under this version of Python. (cf. Section 3.4.1.4, “Known Issues”) . The execution script determines how the tool is called and is given in either cmd or sh, depending on whether the tool is executed on Windows or Linux.

During integration, you can specify tool properties. These can be used to, e.g., switch between different execution modes of a tool, such as fast or precise computation. While the use of properties allows the eventual user of the component a great amount of flexibility, they also easily lead to inadvertent security issues. Consider, e.g., a tool that copies some data to a configurable directory and removes that directory after its computation as part of cleanup. Malicious users may set the configuration directory to `C:\Windows` and cause the tool to remove vital system directories on termination. To prevent users from creating such security issues by accident, RCE does not allow the use of property values containing `\"`, ASCII-characters in the range `0x00-0x1f`, `\\`, `/`, `\*`, `\?`, or `%`.

If you would like to allow your users to supply a configuration file or a configuration directory, please add this file as an explicit input to the component. Other options include, e.g., fixing a "whitelist" of safe configuration options and allowing the user a choice of these configuration options via properties.

3.4.1.1. Directory Structure for Integrated Tools

When executing an integrated tool, a certain directory structure is created in the chosen working directory. This structure depends on the options you have chosen in the integration wizard. The two options that matter are "Use a new working directory each run" and "Tool copying behavior".



Root Working Directory: This is the directory you choose in the "Tool Integration Wizard" as "Working Directory" on the "Launch Settings" page.

Config Directory: In this directory, the configuration file that may be created by the tool integration will be created by default. The configuration files can be created from the properties that are defined for the tool on the "Tool Properties" page.

Input Directory: All inputs of type "File" and "Directory" will be copied here. They will have a subdirectory that has the same name as the name of the input (e.g. the input "x" of type "File" will be put into "Input Directory/x/filename").

Output Directory: All outputs of type "File" and "Directory" can be written into this directory. After that, you can use the placeholder for this directory to assign these outputs to RCE outputs in the post execution script. To write, e.g., the output directory into an output "x" of type "Directory" the following line in the post execution script would be required: `${out:x} = "${dir:output}"`

Tool Directory: This is the directory where the actual tool is located. If the tool should not be copied, it will be exactly the same directory that you choose, otherwise it will be the same as the chosen directory but copied to the working directory.

Working Directory: A working directory is always the location, where all the other directories will be created. If the option "Use a new working directory on each run" is disabled, this will always be the same as the "Root Working Directory". Otherwise, a new directory is created each run (the name will be the run number) and is the working directory for the run.

3.4.1.2. Copying of Integrated Tools

When a component is created in the integration wizard, a configuration file is created.

All configuration files from the tool integration are stored in the directory `<profile folder>/integration/tools/`

In this directory, there is a separation between different kinds of integration realized through one subdirectory for each. The `common` folder always exists.

In these subdirectories, the integrated tools are stored, again separated through into a subdirectory for each. The name of the directory is the name of integration of the tool.

If an integrated tool is copied to another RCE instance or another machine, the directory of the tool must be copied, containing a `configuration.json` and some optional files. It must be put in the equivalent integration type directory of the target RCE instance. After that, RCE automatically reads the new folder and if everything is valid, the tool will be integrated right away.

> **Note**
>
> If you want to delete a tool folder that contains some documentation, this can cause an error. If you have this problem, first empty the documentation folder and delete the empty folder the documentation folder at first (it must be empty), afterwards you can delete the tool folder.

Tool Execution Return Codes

The tools are executed by using a command line call on the operating system via the execution script. When the tool finished executing (with or without error), its exit code is handed back to the execution script and can be analyzed in this script. If in the script nothing else is done, the exit code is handed back to RCE. When there is an exit code that is not "0", RCE assumes that the tool crashed and thus lets the component crash without executing the post script. Using the option "Exit codes other than 0 is not an error" prevents the component from crashing immediately. With this option enabled, the post script wil be executed in any way and the exit code from the tool execution can be read by using the placeholder from *Additional Properties*. In this case, the post script can run any post processing and

either not fail the component, so the workflow runs as normal, or let the component crash after some debugging information was written using the Script API `RCE.fail("reason")`.

3.4.1.3. Integration of CPACS Tools

Additional concepts of CPACS Tool Integration

Extending the common Tool Integration concept, the CPACS Tool Integration has some additional features.

- Parameter Input Mapping (optional): Substitutes single values in the incoming CPACS content, based on an XPath configured at workflow design time as a dynamic input of the component
- Input Mapping: Generates the tool input XML file as a subset of the incoming CPACS file XML structure, specified by a mapping file
- Tool Specific Input Mapping (optional): Adds tool specific data to the tool input file, based on a mapping file and a data XML file
- Output Mapping: Merges the content of the tool output XML file into the origin incoming CPACS file, based on a mapping file
- Parameter Output Mapping (optional): Generates output values as single values of the CPACS result file, based on an XPath configured at workflow design time as a dynamic output of the component
- Execution option to only run on changed input: If enabled, the integrated tool will only run on changed input. Therefore the content of the generated tool input file is compared to the last runs content. Additionally the data of the static input channels are compared to the previous ones.

All the features listed above can be configured in the tool integration wizard on the dedicated *CPACS Tool Properties* page.

The mappings can be specified by XML or XSLT as shown in the following examples. RCE differentiates between these methods in accordance to the corresponding file extension (.xml or .xsl).

For XML mapping, the following mapping modes are supported (see the mapping mode definitions in the mapping examples below):

- append: Elements in the target path that have no equivalent in the source path are retained and are not deleted. Otherwise the elements in the target path are replaced by the corresponding elements in the source path. Two elements in the source and target path are considered to be the same if they have the same element name, the same number of attributes and the same attributes with the same values.
- delete: Before copying, all elements that are described by the target path are deleted in the target XML file. This is also the standard behavior if no mapping mode is explicitly set in a mapping rule.
- delete-only: All elements that are described by the target path are deleted in the target XML file.

If a target element described by the target path is not available in the XML file, it is created including all of its parent elements.

Example for an input or tool specific 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 mode="append">
        <map:source>/path/to/your/element</map:source>
        <map:target>/toolInput/data/var1</map:target>
    </map:mapping>
```

```
    <map:mapping mode="delete">
        <map:source>/path/to/your/element</map:source>
        <map:target>/toolInput/data/var2</map:target>
    </map:mapping>

    <map:mapping mode="delete-only">
        <map:target>/toolInput/data/var3</map:target>
    </map:mapping>

    <map:mapping>
        <map:source>/path/to/your/element</map:source>
        <map:target>/toolInput/data/var4</map:target>
    </map:mapping>

    <xsl:for-each select="$sourceFile/result/cases/case">
        <map:mapping mode="delete">
            <map:source>/path/to/your/case[<xsl:value-of select="position()" />]/element</map:source>
            <map:target>/toolInput/data/condition[<xsl:value-of select="position()" />]/var</
map:target>
        </map:mapping>
    </xsl:for-each>

</map:mappings>
```

Input or tool specific 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" xsi:noNamespaceSchemaLocation="cpacs_schema.xsd">
 <xsl:output method="xml" media-type="text/xml" />
 <xsl:template match="/">
  <toolInput>
   <data>
    <var1>
     <xsl:value-of select="/path/to/your/element" />
    </var1>
   </data>
  </toolInput>
 </xsl:template>
</xsl:stylesheet>
```

Example of an output 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>/toolOutput/data/result1</map:source>
        <map:target>/path/to/your/result/element</map:target>
    </map:mapping>

</map:mappings>
```

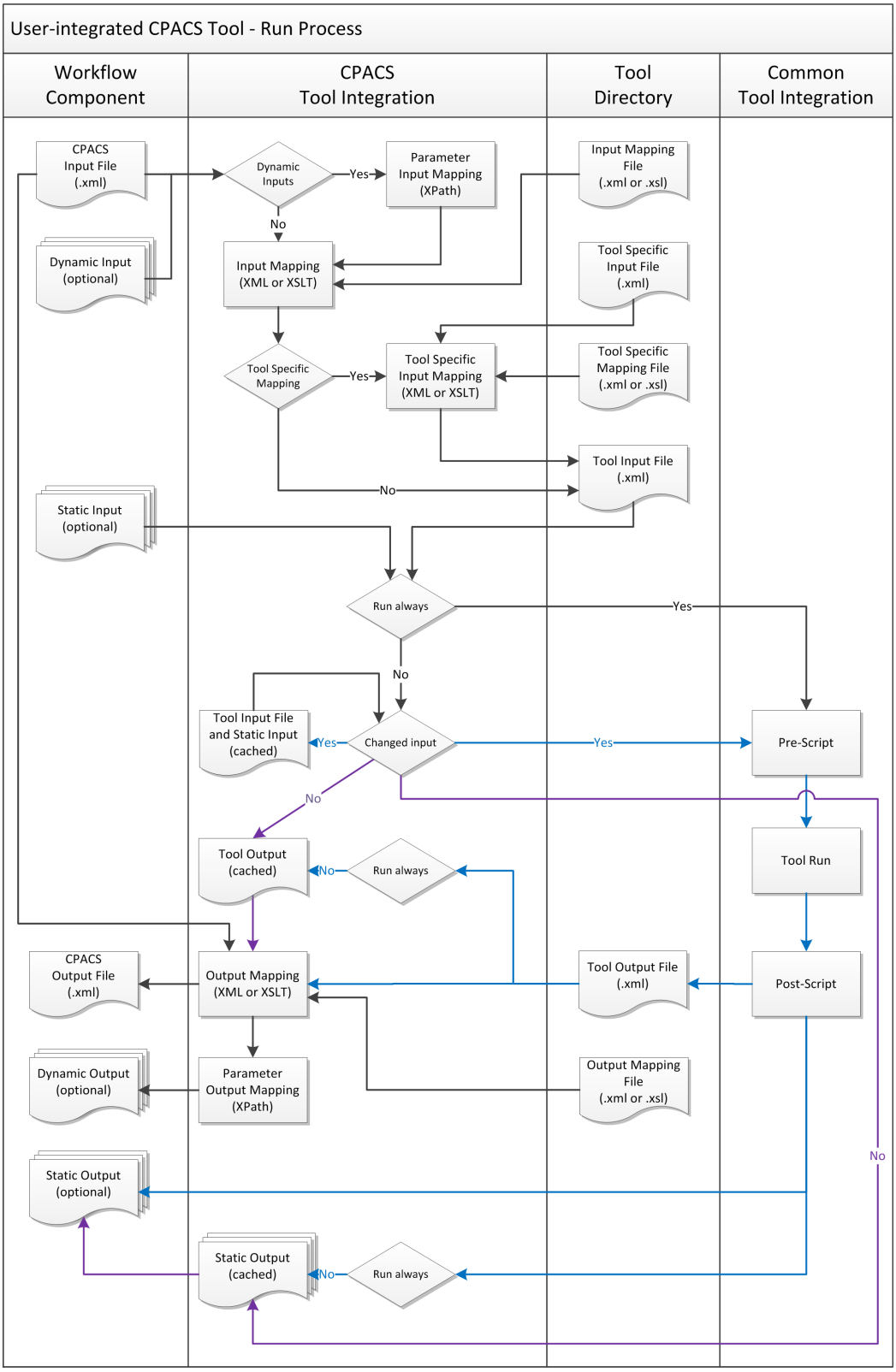
And output XSLT mapping:

```
<?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 toolOutput.xml-->
  <xsl:variable name="toolOutputFile" select="'./ToolOutput/toolOutput.xml'"/>
  <!--Copy complete source file to result file -->
  <xsl:template match="@* | node()">
    <xsl:copy>
      <xsl:apply-templates select="@* | node()"/>
    </xsl:copy>
  </xsl:template>
  <!--Modify a value of an existing node-->
    <xsl:template match="/path/to/your/result">
    <element>
    <xsl:value-of select="document($toolOutputFile)/toolOutput/data/result1"/>
    </element>
    </xsl:template>
</xsl:stylesheet>
```

Please ensure to use the proper namespace for map (xmlns:map="http://www.rcenvironment.de/2015/mapping") in XML mapping files and the proper namespace for xsl (xmlns:xsl="http://www.w3.org/1999/XSL/Transform") in both types of mapping files.

The figure below illustrates how the additional features are used in the run process of an user-integrated CPACS tool.

Figure 3.6. Run process of an user-integrated CPACS Tool



Integrate a CPACS Tool into a Client Instance

1. Start RCE as Client
2. Open the *Tool Integration Wizard* by clicking the *Integrate Tool...* in the *File* menu.

Note

You will always find further help by clicking the *?* on the bottom left corner on each page of the wizard or by pressing *F1*.

3. Choose the option *Create a new tool configuration from a template*.

Note

The CPACS templates delivered with RCE are designed to match the conventions of the old CPACS tool wrapper (respectively ModelCenter tool wrapper). Most of the properties are preconfigured and do not need to be changed.

4. Select one of the *CPACS* templates.
   Click *Next*.

5. Fill in the *Tool Description* page.
   Click *Next*.

6. On the *Inputs and Outputs* page you will find preconfigured static in- and outputs, that will match the old tool wrapper conventions. If your tool needs additional in- or outputs, feel free to configure.
   Click *Next*.

7. Skip the page *Tool Properties* by clicking *Next* since it is not relevant for tools that match the conventions of the old CPACS tool wrapper.

8. Add a launch setting for the tool by clicking the *Add* button on the *Launch Settings* page. Configure the path of the CPACS tool and fill in a version, click *OK*. If you would like to allow users of your tool to choose that the temp directory won't be deleted at all after workflow execution, check the property *Never delete working directory(ies)*. Not to delete the working directory can be very useful for users for debugging purposes, at least if they have access to the server's file system. But this option can result in disc space issues as the amount required grows continuously with each workflow execution. It is recommended to check that option during integrating the tool and uncheck it before publishing the tool.
   Click *Next*.

9. The *CPACS Tool Properties* are preconfigured to match the folder structure defined for the old CPACS tool wrapper. In most cases you do not have to change this configuration. If you are using XSLT mapping, please select the corresponding mapping files. If your tool does not work with static tool specific input, please deselect this property.
   Click *Next*.

10. In the *Execution command(s)* tab on the *Execution* page, you need to define your execution command itself as well as optional pre and post commands. Commands will be processed sequentially line by line. An example for a typical Windows command including pre and post commands will look like the following:

```
rem pre-command
pre.bat

rem tool-execution
YourTool.exe ToolInput/toolInput.xml ToolOutput/toolOutput.xml

rem post-command
post.bat
```

11. Click *Save and activate* and your tool will appear immediately in the palette and is be ready to use.

12. If not already done, do not forget to publish your tool (cf. Section 3.6, "Tool publishing and authorization") after testing it locally. To check if your tool is successfully published to the RCE network open the tab *Network View* at the bottom and checkout *Published Components* after expanding the entry of your RCE instance.

Integrate a CPACS Tool into a Server Instance in Headless Mode

The way to integrate a CPACS tool on a server running RCE in headless mode is as follows: Perform the steps to integrate a CPACS tool on a client instance and make sure that the path of the CPACS tool configured on the *Launch Settings* page (step 8) matches the absolute tool path on your server system. Afterwards, you will find the configuration files inside your rce profile folder at the following location:

```
/integration/tools/cpacs/[YourToolName]
```

Copy the folder `[YourToolName]` to the same location inside the profile folder running with your headless server instance. Use the "auth" commands (cf. Section 3.6, “Tool publishing and authorization”) to publish your tool. If the server instance is already running, your tool will be available immediately after publishing.

3.4.1.4. Known Issues

As noted above, pre- and postscripts are executed using a Java implementation of Python 2. One particular caveat is Python 2's handling of unicode strings, which requires prefixing strings containing non-ASCII-characters with `u`. Please refer to, e.g., this tutorial [https://python.readthedocs.io/en/v2.7.2/howto/unicode.html] on handling unicode in Python 2 for further information.

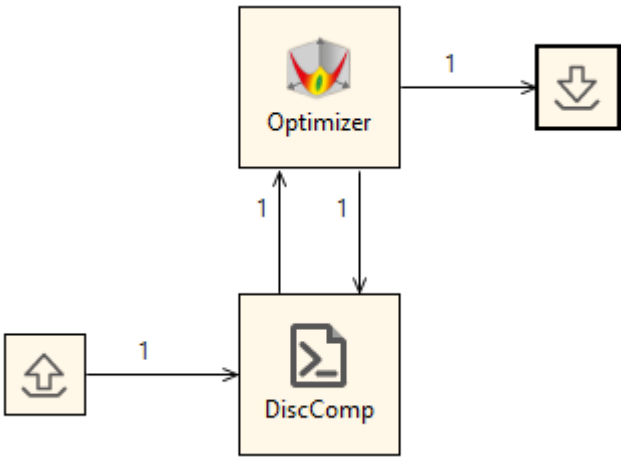
Under Windows, the execution script is eventually handed to the Java method `Runtime#eval`. This method in turn relies on internal Windows APIs which have known issues with handling unicode. Please make sure that your execution script works as expected when executed via RCE with unicode strings.

3.4.2. Integrating Workflows as Components (Experimental)

In this section we describe how to integrate a workflow containing multiple components as a component itself. This feature is currently experimental and *not recommended for productive use*.

Consider a disciplinary tool that computes the value of some function f_c(x) for some parameter c and some input value x and assume that the user has already integrated this tool as the component `DiscComp`. Furthermore assume that in multiple workflows the user would like to fix some value for c and find a minimum of f_c. She implements this use case via the structure shown in the following figure.

Figure 3.7. Workflow for determining the optimal input for the function f_c(x).



In that workflow, the user opted to provide the parameter c via an input provider, while she used an optimizer to determine the optimal value of x. That optimal value is then written via an output writer. The user now wants to use this workflow as part of other, more complex workflows.

One approach would be to simply copy the part of the workflow that implements the actual computation (i.e., the components `Optimizer` and `DiscComp`) and paste it whenever it needs this functionality in other workflows. This approach, however, is neither scalable nor maintainable: While this example requires only copying of two componentes, one can easily imagine situations in which the functionality to be copied is implemented via dozens of components, which leads to severe cluttering of the workflows in which the functionality is used. Furthermore, if the user changes the original workflow, e.g., if she uses another algorithm for the optimization, she would have to re-copy the changed parts to all workflows that use the original parts.

Instead of manually copying and pasting, the user may instead opt to integrate the workflow shown in the above figure as a tool to be used in other workflows. This allows her to hide the details of the implementation (i.e., the use of an optimizer and of `DiscComp`) from users of her component and to easily update that implementation.

In the following, we first show how to integrate an existing workflow as a component before detailing the technical backgrounds of executing a workflow as a component. Finally, we discuss caveats and common questions about this feature. In all these sections, we will refer to an "inner" workflow and an "outer" workflow. These refer to the workflow that is integrated as a component and to the workflow in which that component is used later on, respectively.

3.4.2.1. Integrating a Workflow via command console

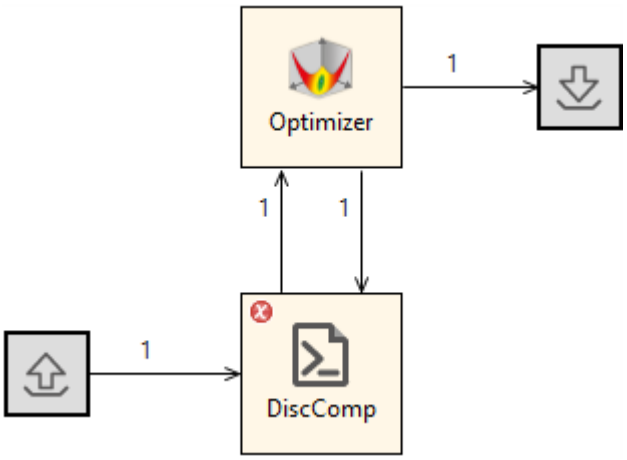
Before integrating the workflow shown above, we assume that you have already constructed a workflow that implements the behavior that you want to provide to other users as a component. Moreover, we assume that this workflow contains some input providers that feed initial data into the workflow and some output writers that persist the results of the computation implemented by the workflow. In the figure above, these input providers and output writers are situated to the left of the component `DiscComp` and to the right of the optimizer, respectively. Finally, the workflow to be integrated must not contain any placeholders (cf. ???). Otherwise user input would be required at execution time in order to assign values, which would prevent automated execution of the integrated workflow.

> **Note**
>
> You can easily determine whether your workflow contains placeholders by opening the workflow execution wizard (either via the green arrow in the upper bar in the GUI or via the shortcut Ctrl + Shift + X). If there exist any placeholders that are to be assigned values before the start of the execution, the wizard will show a second page that displays all such placeholders. If no such page exists, the workflow does not contain placeholders and is ready for integration as a component.

Integrating a workflow consists of nothing more than determining endpoints of components in the inner workflow that will be exposed to the outer workflow by the resulting component. In this case, we opt to expose the input `c` of `DiscComp` as well as the output `x_output` of `Optimizer`. In general, inputs will be exposed as inputs on the component in the outer workflow, while outputs will be exposed as outputs. It is not possible to expose an input of a component in the inner workflow as an output to the outer workflow, or vice versa.

In order to integrate the above workflow as a component, we first remove the input providers and output collectors that handle the inputs and outputs that are to be passed into the inner workflow by the outer workflow. In the example above, we simply deactivate the two components (e.g. via the keyboard shortcut Ctrl + D) and obtain the workflow shown in the following figure.

Figure 3.8. Workflow from the above figure prepared for integration as a component.



While previously, all endpoints of all components were connected, now there exist two unconnected endpoints: The input c of `DiscComp` as well as the output `x_optimal` of `Optimizer`. The workflow is now ready for integration as a component.

Integration of workflows is performed via the command console and, in particular, via the command `wf integrate`. This command has the following general form:

```
wf integrate [-v] <component name> <absolute path to .wf file> [<exposed endpoint definition>...]
```

The optional argument `-v` enables verbose mode. If this parameter is set, the command outputs detailed information about the endpoints that are exposed to the calling workflows. This does not change the behavior of the command.

The parameter `component name` determines the name of the component that is integrated, i.e., the name that will appear in the pallet and in the workflow editor. Since in our example the purpose of the new component in our example is to determine some optimal parameter x, we opt to call the component `FindOptimalX`.

The parameter `absolute path to .wf file` is self-explanatory and denotes the path on your local file system where the workflow file describing the workflow to be integrated is located. In our example we assume that the workflow file is located at `C:\Users\user\workflow.wf`.

> **Note**
>
> Recall that you can obtain the absolute path to any workflow file in the project explorer via a right click on the workflow and selecting *Copy Full Path*.
>
> Furthermore, recall that parameters in the command console are separated by spaces unless the parameter is surrounded by quotation marks . Hence, if the path to your workflow contains spaces, enclose it in quotation marks. Finally, recall that backslashes must be escaped, i.e., the path `C:\My Folder` would have to be entered as `"C:\\My Folder"`.

Each succeeding parameter is interpreted as the definition of an exposed endpoint. Each such definition is of the following form:

```
--expose <component name>:<internal endpoint name>:<exposed endpoint name>
```

Here, `component name` refers to the name of the component in the inner workflow whose endpoint is to be exposed. The parameter `internal endpoint name` denotes the name of the endpoint of the component that is to be exposed, while the parameter `exposed endpoint name` determines the name of the endpoint on the resulting component. Make sure that each `exposed endpoint name` is unique within the context of the resulting component, as the behavior of a component with multiple inputs or outputs of the same name is undefined.

Note

Instead of the names of the component and the endpoints that are displayed in the workflow editor, you may instead use the internal identifiers of these nodes and endpoints, respectively. These are not currently shown in the GUI of RCE but can, e.g., be determined by inspecting the workflow file via some text editor. While this should not be necessary when integrating workflows manually, it may prove useful when automating the creation and integration of workflows.

Recall that you do not need to specify whether the endpoint is exposed as an input or as an output, but that the underlying endpoint determines the configuration of the endpoint on the resulting component: Inputs are only ever exposed as inputs, whereas outputs are only ever exposed as outputs. This principle extends to the configuration of inputs: If the endpoint on the component in the inner workflow is, e.g., configured be required for component execution and to only expect a constant value, then the endpoint on the resulting component is configured analogously.

Furthermore recall that we want to expose the input `c` of `DiscComp` as well as the output `x_output` of `Optimizer`. We want the former input to retain its original name, while we want to expose the latter input as `optimalX`. In order to integrate the example workflow prepared above as a component, we thus issue the following command:

```
wf integrate FindOptimalX "C:\Users\user\workflow.wf"--expose DiscComp:c:c --expose
 Optimizer:x_optimal:optimalX
```

When enabling verbose mode via the switch `-v`, RCE writes the following output

```
Input Adapter : c --[Float,Constant,Required]-> c @ 03b5b758-3b44-4a53-b832-be9991321285
Output Adapter: x_optimal @ 402cac5e-2206-48cc-a62f-803bd320a15a --[Float]-> x_opt
```

where `03b5b758-3b44-4a53-b832-be9991321285` and `402cac5e-2206-48cc-a62f-803bd320a15a` denote the IDs of the component `DiscComp` and of `Optimizer`, respectively.

Once the execution of the command has finished, a new component named `FindOptimalX` with a single input named `c` and a single output named `optimalX` will be available for use in all other workflows.

3.4.2.2. Workflow Integration Editor

With RCE 10.4 we provide the Workflow Integration Editor to support the workflow integration via the graphical user interface of RCE. There are two options to open the editor:

- Select the workflow you want to integrate as a component from the Project Explorer and go to either the main menu *Intgration > Integrate Workflow File* or use the quick acess via the icon bar.
- Select a (sub-)workflow in the Workflow Editor and choose from the context menu *Integrate selected Components as Component*

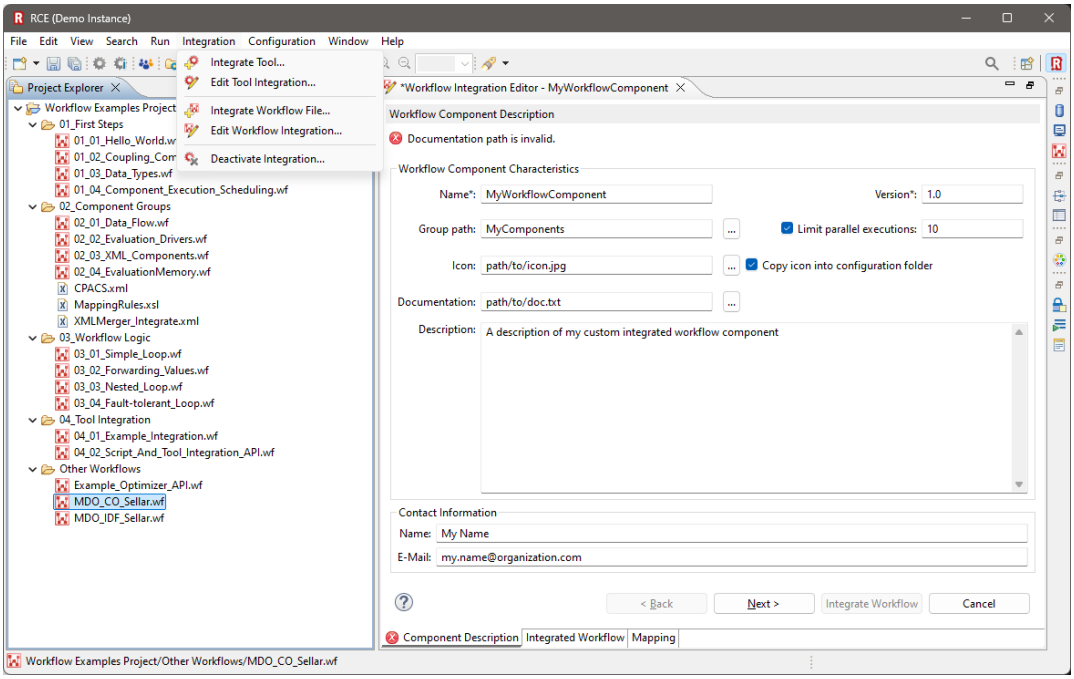
Required fields are marked with an asterisk (*). Press *Integrate Workflow* to integrate the workflow as a component. The component will automatically show up in the Workflow Editor palette.

Note

The workflow integration editor has a dynamic help, which is shown by clicking on the question mark on the bottom left or by pressing F1. It will guide you through the tabs of the workflow integration editor.

To edit an already integrated workflow component use the *Intgration > Edit Workflow Integration* from the main menu or use the context menu in either the Workflow Editor or the Palette after selecting the desired workflow component.

Figure 3.9. Workflow Integration Editor

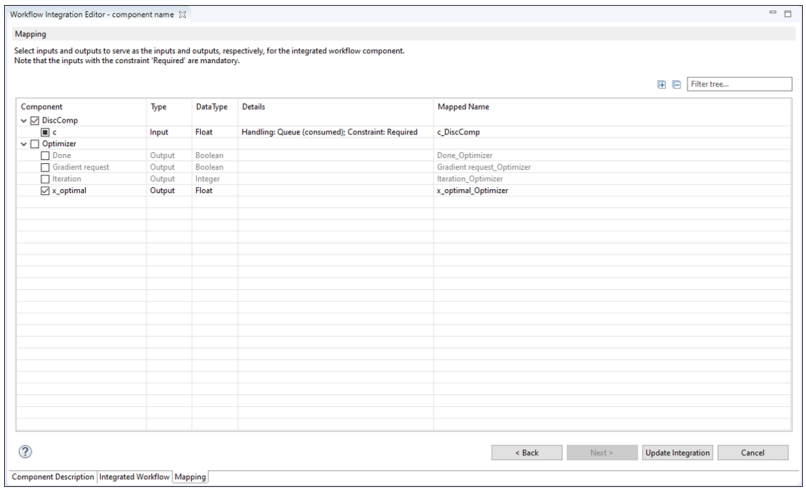


One major part of the workflow integration consists of the input/output mapping. On the Mapping tab all unconnected endpoints of your selected (sub-)workflow are displayed. Here you can select endpoints to serve as inputs and outputs for the integrated workflow component.

> **Note**
>
> Inputs with the constraint *Required* are mandatory.

You can assign custom names for your endpoints by editing the Mapped Name cell in the table.

Figure 3.10. Mapping Page



3.4.2.3. Technical Details of Executing an Integrated Workflow

In this section some technical details are given for interested users how exactely an integrated workflow used as a workflow component in another workflow is executed. If you are interested in more conceptual details or encounter problems while running an integrated workflow, this section may be helpful.

Recall that each workflow that RCE executes is controlled by some particular instance, i.e., by the workflow controller. Since executing an integrated workflow executes the underlying workflow, RCE requires a workflow controller for doing so. That workflow controller may or may not be the same as the one executing the outer workflow. Currently, the instance publishing the component serves as the workflow controller for the execution of the inner workflow. We call it the "inner workflow controller" to distinguish it from the workflow controller that runs the outer workflow. This "inner workflow controller" runs a copy of the workflow in the exact state in which it was integrated, i.e., changes made to the workflow after integration will have no effect on the behavior of the integrated component.

Futhermore, since the publishing instance serves as "inner workflow controller" for the execution of the integrated workflow, the execution of the integrated workflow will show up in the Workflow Data Browser of the publishing instance under the name `<workflow component name> running as component '<component name>' of workflow '<outer workflow>'`. The `<workflow component name>` denotes the name as which the publishing instance published the integrated workflow, `<component name>` denotes the name under which the workflow component is used in the outer workflow, and `<outer workflow>` denotes the name under which the outer workflow is stored in the Workflow Data Browser of its workflow controller.

Note

Nesting workflows, i.e., integrating workflows as components that already contain workflows integrated as components, can easily lead to unreadable names of workflow executions that are stored in the Workflow Data Browser. This may significantly inhibit manual inspection of the resulting data. Keep this in mind when designing workflows.

Technically, before starting the integrated workflow, the instance controlling the inner workflow injects two additional components into the workflow, one so-called input adapter and one so-called output adapter. These components are not accessible by the user when constructing workflows. They are only used to transport data from the inputs of the workflow component to the exposed inputs as well as data from the exposed outputs to the outputs of the workflow component, respectively.

Upon execution of the integrated component in the outer workflow, the instance publishing the component first injects the input and the output adapter as described above. It subsequently executes the workflow and collects the results via the output adapter.

3.4.2.4. Limitations, Caveats, and FAQ

Since the integration of workflows as components is currently under development and only released as a beta feature, there are some caveats and known issues that you should be aware of. We have alluded to these limitations and caveats throughout this section, but briefly list them here again for the sake of readability.

- Workflow files are "frozen" at integration time. Changes to an integrated workflow file after integration do not change the behavior of the component. If you want to apply changes to the workflow file to the component, you will have to re-integrate the workflow.

- Currently, no placeholder files (cf. ???) are supported, i.e., the integrated workflow must contain no placeholders. Moreover, the workflow is not checked for containing placeholders at integration time, but instead the execution of the the component will fail at execution time.

- Via the command console, the user cannot specify a version of the integrated component. If there is demand, we will add the command line switch `--version` in order to allow the user to have multiple versions of the same workflow integrated simultaneously. Also, the user can currently not specify an individual icon to be used for the integrated component. This may also be added in future versions.

- If some adapted output is written to multiple times during a single run of the integrated workflow, only the final values written to that output are forwarded to the outer workflow.

- Due to this new implementation, there is doubled functionality between the command `wf integrate` and the command `ra-admin wf-publish`. After the full release of the integration of workflows as component, the latter command will be deprecated and its output replaced by a message asking the user to use `wf integrate` instead.
- If the underlying workflow is paused during execution, this pause state is not reflected in the calling workflow. Instead the component is shown as running. Similarly, if the integrated workflow includes some result verification and the results are rejected, the component simply fails instead of indicating the rejection of results.
- Component names passed to the command `wf integrate` are not checked to satisfy the rules on component names. This will be fixed before release and integration of a component with an invalid name will be refused with an informative error message.

Furthermore, there are some common questions that may occur in the context of integrating a workflow as a component. We collect and answer these questions here again for the sake of readability.

Where is the integration folder of my new component?	The integration of a workflow as a component is stored in a profile in the folder `integration\tools\workflow`.
Can I move the folder containing the integration of a workflow to other instances, similarly to the integration of common tools?	Yes, this is possible, since the integration folder contains a copy of the workflow file which was produced at integration time. Also, you can publish integrated workflows to other instances just as you can publish common tools.
What happens if an integrated workflow uses some remote components that are not available?	In that case the component is still available as long as the instance publishing it is available. The availability of the components contained in the integrated workflow is only checked at execution time. If a component is unavailable at that time, the execution of the component fails.

3.5. Connecting RCE instances

Since RCE 10, RCE provides three possibilities to connect your RCE instance to other RCE instances and to use the user-integrated tools and components published on those instances: The RCE network connections, SSH Uplink connections and SSH Remote Access connections. RCE connections are meant to be used only in a trusted network (e.g. your institution's internal network). The RCE network traffic is currently *not encrypted* . This means that it is *not secure* to expose RCE server ports to untrusted networks like the internet. In the case that it is not possible or not secure to use RCE connections, SSH connections provide a more secure alternative.

As the new Uplink connections do not yet support all features of the former SSH connections (the publishing of workflows is not possible by Uplink connections), we decided to keep both types of connections in the current release. Thus, in the network view there are now 3 types of connections: the standard RCE connections (meant to be used in secure internal networks), the old "SSH Remote Access Connections" and the new "Uplink Connections".

The following table compares the three connection types:

Table 3.11. Connection types - feature matrix

Connection type	RCE connections ("internal network")	SSH Remote Access connections	SSH Uplink connections
Publishing built-in tools (e.g. Joiner, Parametric Study, ...)	yes	no	no

Connection type	RCE connections ("internal network")	SSH Remote Access connections	SSH Uplink connections
Publishing user-integrated tools	yes	yes	yes
Publishing workflows as tools	no *	yes	no *
Symmetric/bidirectional tool publishing	yes	no	yes
Accessing remote workflow status and data management	yes	no	no
Remote system monitoring (CPU/RAM)	yes	yes	no **
Login authorization (via password or SSH keyfile)	no	yes	yes
Suitable for insecure networks (e.g. internet)	no (!)	partially ***	yes (via relay)

* = planned for RCE 11; ** = may be added in a future release; *** = connections are encrypted, but require an open incoming network port for publishing tools - if possible, use Uplink connections instead

3.5.1. RCE Network Connections

RCE connections are meant to be used only in a trusted network (e.g. your institution's internal network). To build up a network of RCE instances, at least one of the instances has to be configured as a server (see the "Configuration" section or the sample configuration file "Relay server" for details).

On the client side, RCE network connections can be added in the "network" view by clicking "Add network connection" and entering the hostname and port of an RCE server instance. The connections are shown in the "RCE Network"->"Connections" subtree. They can also be edited, connected and disconnected in the network view. However, the changes made here are not saved in the configuration yet, i.e. they will be lost when RCE is closed or restarted. To permanently add connections, you can edit the configuration file (see the "Configuration" section for details).

In the "RCE Network" -> "Instances" subtree all RCE instances in the network are listed. When expanding the entry for an instance, you can see monitoring data like CPU or RAM usage for this instance, and the published components and tools of this instance (if it has any).

The published components and tools of the other instances in your network are also shown in the palette of the Workflow Editor. From there, you can use them in your workflows just like your local components and tools. When you start a workflow, in the "Execute Workflow" wizard there is an overview which component will be run on which RCE instance. If a component is available on several instances, you can choose here on which instance it should be run. In the same wizard, you can also choose another instance as the "Controller Target Instance", which means that the workflow execution will be controlled by this instance (see the section "Configuration Parameters" for more information). This can be useful when you start a long-running workflow where all components are run on remote instances and you do not want to keep your local computer connected all the time.

3.5.2. Uplink Connections

Uplink connections allow to use the "SSH relay" functionality. This means that it is possible to setup a single server as the "relay" for a project (and only this server needs to be reachable on an SSH port). All other RCE instances can connect to this server as clients via SSH Uplink Connections and

publish their tools so that they can be used by other clients. (In contrast, with the former version of SSH connections every partner who wanted to publish tools needed to configure an SSH server).

3.5.2.1. Configuring an RCE instance as an Uplink relay

The RCE instance that should be used as the relay has to be configured as an SSH server and provide at least one account with the role "uplink_client" or "remote_access_user"(see Section 2.2, “Configuration and Profiles” or the sample configuration file "Uplink relay" for details). The recommended role is "uplink_client", which allows only access to Uplink connections and no access to an interactive SSH shell.

Note

In RCE 10, only RSA keys are supported when configuring an SSH account using a key file. A private key has to be in PEM format starting with `-----BEGIN RSA PRIVATE KEY-----` or in the proprietary OpenSSH format, which starts with `-----BEGIN OPENSSH PRIVATE KEY-----`.

We recommend a key length of at least 3000 bits. When using Windows, we recommend `puttygen` for key generation, which comes bundled together with the popular SSH client `putty`. When using Linux, we recommend `ssh-keygen`, which is part of the SSH tools by OpenSSH. Please refer to the documentation of your system for instructions on installing the required tools.

3.5.2.2. Configuring an RCE instance as an Uplink client or gateway (in GUI mode)

On the client side, Uplink connections can be added in the "network" view by clicking "Add Uplink Connection". In the following dialog, enter the hostname and port of an Uplink relay as well as the user name and the authentication data of an SSH account configured on this instance. Depending on the SSH account, you have to authenticate using a passphrase or by an RSA private key file. If your private key is protected by a passphrase, select the authentication type "Keyfile with passphrase protection", else select "Keyfile without passphrase protection". If several clients are using the same account on a relay, enter a different "client ID" on each of them.

If the instance should serve as a gateway (i.e. forward tools between the (external) Uplink network and a local network), set the "isGateway" parameter to "true".

The connections are shown in the "Uplink"->"Uplink Connections" subtree. They can also be edited, connected and disconnected in the "network" view. It is possible to store passphrases using the Eclipse Secure Storage Mechanism. However, the changes made here are not saved in the configuration yet, i.e. they will be lost when RCE is closed or restarted. To permanently add Uplink connections, you can edit the configuration file (see Section 2.2, “Configuration and Profiles” for details). Sample configuration files are avaible as "Uplink Client" and "Uplink Gateway".

3.5.2.3. Configuring an Uplink Gateway in non-GUI mode

Configuring a gateway in non-GUI mode involves four steps:

- Configure an SSH Uplink connection to the SSH relay server in the profile's `configuration.json` file. In this connection, make sure to set the "`isGateway`" parameter to "`true`" (without quotes).
- Configure a normal RCE server port for the internal network. This is the network port that clients in the local (internal) network can connect to with standard ("internal network") connections.
- Using the file-based import feature (see section Section 2.2.5, “Importing authorization data without GUI access”), import the SSH password or the SSH keyfile passphrase for logging into the Uplink relay. (Please note that currently, the gateway must be (re)started after creating these import files to apply the changes.)

- To allow the gateway to forward tools that are not public, but only published for specific authorization groups, the gateway must be a member of at least one matching group. Use the file-based import feature (see section Section 2.2.5, “Importing authorization data without GUI access”) to import any required group keys. (Please note that currently, the gateway must be (re)started after creating these import files to apply the changes.)

3.5.2.4. Tool publishing

In order to make tools available for other clients, you have to publish them (for example using the "Component Publishing" view; see user guide for more information about pubishing/authorization groups). To make a tool available via an SSH relay, it has to be either in the "Public Access" group or in an authorization group which name starts with "**external_**". Tools in other authorization groups will only be shared in your local RCE network.

Note

Note: Tools that are available to a client via an Uplink connection are also available to RCE instances connected to that client in its local RCE network (if they possess the corresponding authorization group key and the "isGateway" option is set for the Uplink connection). Accordingly, tools published by those instances in the "Public Access" group or in an authorization group which name starts with "external_" will also be made available via the Uplink relay. Please not that this only works if the gateway itself also possesses the authorization group key.

3.5.2.5. Possibly surprising behavior (or non-behavior)

Nodes connected via Uplink connections do not show up in the network view as nodes (same as Remote Access).

Imported tools show up in the Network view under the Node running the Uplink connection (also the same as Remote Access), and they are not yet marked or distinguishable from normal components.

Tools located on the RCE instance serving as relay are not published automatically. If you want to publish them, you have to add a connection to the relay from the same instance.

3.5.2.6. Known issues/limitations of the current release

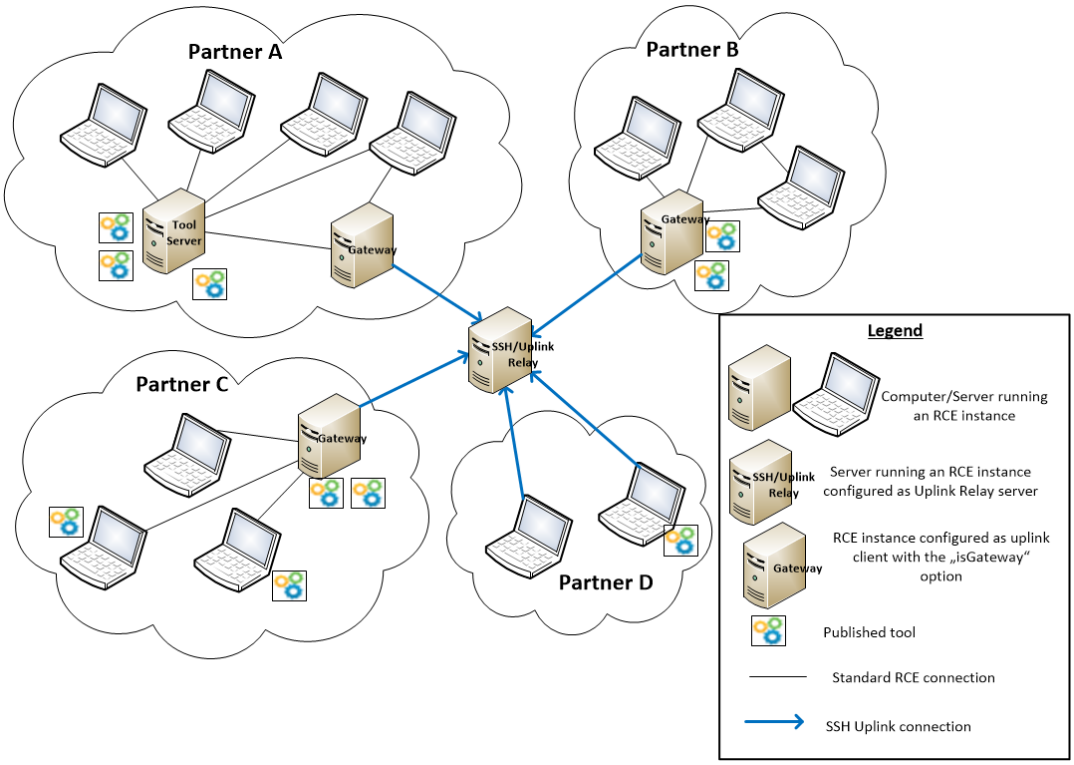
Uplink connections are an *experimental* feature in RCE 10.x and have some known limitations:

- Connections are always encrypted as part of the SSH connection, but there is no additional "internal" encryption of tool input/output data yet (which is planned for future versions to protect users against untrustworthy relay servers).
- The behavior on errors, disconnects, and server shutdowns is not fully implemented yet; this will be stabilized in RCE 11.
- Custom tool icons are not yet transferred over Uplink connections.

3.5.3. Example of a Cross-Organization Network

The following figure gives an example of how a cross-organization network using Uplink connections could be structured:

Figure 3.11. Example RCE network



The four project partners in the example all have an internal network of RCE instances which are connected by standard RCE connections. Uplink connections to a relay server are used to connect between the different partners. The relay server is located outside of the organizations networks, and only the relay server has to be reachable via SSH over the internet. Typically, for each organization one RCE instance (called SSH gateway) established an SSH connections to this relay server. All other instances in the institution’s internal network can be connected to it by standard RCE connections and still publish tools to the other partners/ access tools published by other partners.

Each institution in the example has a different internal setup, all of which are possible:

- Partner A has a dedicated RCE server where the published tools are located, which is connected to the SSH gateway by an RCE connection. All other RCE users in the internal network are connected to this server.
- Partner B has put all the tools directly on the SSH gateway instance.
- In Partner C's network, some tools are located on the SSH gateway, but some tools are also published by users directly on their own machines. As long as they are connected to the SSH gateway also those tools can be published to the other partners.
- Partner D has no tool server, instead the users’ computers connect directly to the relay server.

3.5.4. SSH Remote Access Connections

Note

Since RCE 10, the recommended connection type for secure connections are the SSH Uplink connections (cf. previous chapters). However, as the new Uplink connections do not yet support all features of the SSH Remote Access Connections (e.g. access to monitoring data is not possible by Uplink connections), the current release provides both types of connections. This chapter describe the usage of the SSH Remote Access Connections.

SSH connections provide a more secure alternative to the standard RCE connections and can be used to access tools remotely. The published tools are shown in the palette of the client's Workflow Editor

(this may take a few seconds after connecting, as the tool list is fetched from the remote hosts every few seconds). From there, you can use them in your workflows just like your local components and tools. Differently from tools accessed by RCE network connections, in this case the component is shown to be executed on your local instance in the Workflow Execution wizard.

Also workflows that were published on the remote instance (for information about the publishing see section "Remote Tool and Workflow Access") are shown as components in the palette of the client's Workflow Editor in the group "SSH Remote Access Workflows" (if the client runs RCE 7.1 or newer). These remote workflows can be added to workflows and executed like local components/tools.

3.5.4.1. Configuring an RCE instance as an SSH server

The RCE instance that publishes the tool, or another instance connected to it by RCE network connections, has to be configured as an RCE remote access server (see the "Configuration" section or the sample configuration file "Remote access server" for details).

Note

In RCE 10, only RSA keys are supported when configuring an SSH account using a key file. A private key has to be in PEM format starting with `-----BEGIN RSA PRIVATE KEY-----` or in the proprietary OpenSSH format, which starts with `-----BEGIN OPENSSH PRIVATE KEY-----`.

We recommend a key length of at least 3000 bits. When using Windows, we recommend `puttygen` for key generation, which comes bundled together with the popular SSH client `putty`. When using Linux, we recommend `ssh-keygen`, which is part of the SSH tools by OpenSSH. Please refer to the documentation of your system for instructions on installing the required tools.

3.5.4.2. Configuring an RCE instance as an SSH client

On the client side, SSH connections can be added in the "network" view by clicking "Add SSH Remote Access Connection". In the following dialog, enter the hostname and port of an RCE instance that provides an SSH server as well as the user name and the authentication data of an SSH account configured on this instance. Depending on the SSH account, you have to authenticate using a passphrase or by an RSA private key file. If your private key is protected by a passphrase, select the authentication type "Keyfile with passphrase protection", else select "Keyfile without passphrase protection".

The connections are shown in the "SSH Remote Access"->"SSH Remote Access Connections" subtree. They can also be edited, connected and disconnected in the "network" view. It is possible to store passphrases using the Eclipse Secure Storage Mechanism. However, the changes made here are not saved in the configuration yet, i.e. they will be lost when RCE is closed or restarted. To permanently add SSH connections, you can edit the configuration file (see the "Configuration" section for details).

3.6. Tool publishing and authorization

RCE components and integrated tools can be published to make them usable by other connected ("remote") RCE instances. The publishing options for each component/tool can be defined in the "Component Publishing" view. In this view, each component can be assigned to one of three basic publication levels:

- Local (the default option): Components with the "local" setting can only be used on the local instance; they are not visible to other instances.
- Custom: This setting allows to make the component/tool available only to specific groups of users. To use this setting, one or more authorization groups have to be created first, which is explained in the next section. Each component/tool can then be assigned to one or multiple groups. Users on remote instances can see and use components if they are members of at least one of these groups.

- Public: Components with the "public" setting can be used by all connected RCE instances. This is equivalent to the tool and component publishing in earlier versions of RCE. Tools in the "public" group are also available over Uplink Connections and Remote Access connections.

> **Note**
>
> If the "Component Publishing" view is not visible, you can open it from the "Window > Show View" menu. If it is not listed there, choose "Other" and select them from the "RCE" category.

3.6.1. Managing authorization groups

Authorization groups can be created and managed in the "Authorization Groups" dialog, which can be opened from the "Component Publishing" view. To create a new group, click the "Create Group"-button and enter a name for the group. To provide access to this group to other users, select the group in the list and click "Export Group Key". Copy the provided key from the dialog that appears, and pass it on to the users that you would like to invite to this group.

> **Note**
>
> IMPORTANT: This exported group key is similar to a password. When passing it to other users, make sure to use a communication medium that unauthorized users cannot easily intercept. For example, passing the key via an encrypted chat system provided by your employer, or a Team Site that is only accessible to project members, is usually secure enough. On the other hand, sharing it by email outside of your organization is usually unsafe, and we recommend using more secure alternatives.

When the other user receives this key, they can import it into their RCE instance by using the "Import Group Key" button in their "Authorization Groups" dialog. After importing a key on an RCE instance, all tools published for that group on connected RCE instances are visible and can be used like a "public" component.

> **Note**
>
> To provide access to tools over Uplink Connections, the tools either have to be "public" or in an authorization group which name starts with "external_". Tools in other authorization groups are only accessible from the internal RCE network.

3.6.2. Publishing tools on the command console

Creating custom tool groups and publishing tools is also possible using the "auth" commands on the command line. A short reference:

- `auth create <name>` - creates an authorization group
- `auth list` - lists available access groups
- `auth delete <name/id>` - deletes an authorization group; if the name is ambiguous (e.g. there are two groups named "groupName"), you need to add the randomly generated id behind it, separated with a colon (e.g. `groupName:2716ab2d25`)
- `auth export <name/id>` - exports a group key in a form that can be imported by another instance via GUI or command line
- `auth import <exported key>` - imports a group key exported via GUI (as described above) or via the `auth export` command. The group name is embedded in the exported key, and is set automatically.
- `components set-auth <component id> <permissions>` - sets the permissions for a component. Possible values for "permissions" are either "`local`", "`public`", or a comma-separated list of authorization groups/ids.

- `components list-auth` - shows a list of all defined authorization settings. These settings are independent of whether a matching component exists, which means that settings are kept when a component is removed and later added again.

The component ids used in this commands can be derived as follows:

- `rce/<component name>` for standard RCE components, e.g. "`rce/Parametric Study`"
- `common/<tool name>` for integrated tools of type "common" e.g. "`common/ExampleTool`"
- `cpacs/<tool name>` for integrated tools of type "CPACS" e.g. "`cpacs/CPACSExampleTool`"

3.7. RCE's Command Console

RCE features a command console.
This command console is accessible via the command console view in RCE's desktop mode or via SSH. Commands consist of one or two tokens e.g. "wf" for one token and "wf run" for two tokens. A list of all commands is shown in section Section 3.7.1, "Commands". In addition, some commands can be provided with parameters. There are three types of parameters. These types and their uses are explained in section Section 3.7.2, "Parameters".

3.7.1. Commands

Commands are structured in groups. A command group is defined by the first token of a command. This means, that commands like "wf run" and "wf verify" are in the same command group. Some command groups like "mail" consist only of a single command. For each group there is a table explaining all the related commands.

The 'auth' command

Command	**Description**
auth	Alias for "auth list".
auth create <group id>	Creates a new authorization group. *<group id>*: an identifier consisting of 2-32 letters, numbers, underscores ("_") and/or brackets
auth delete <group id>	Deletes a local authorization group. *<group id>*: an identifier consisting of 2-32 letters, numbers, underscores ("_") and/or brackets
auth export <group id>	Exports a group as an invitation string that can be imported by another node, allowing that other node to join this group. *<group id>*: an identifier consisting of 2-32 letters, numbers, underscores ("_") and/or brackets
auth import <invitation string>	Imports a group from an invitation string that was previously exported on another node.

Command	Description
	<invitation string>: imports a group from an invitation string that was previously exported on another node
auth list	Lists the authorization groups that the local node belongs too.

The 'cn' command

Command	Description
cn	Alias for "cn list".
cn add <target> <description>	Add a new network connection (Example: cn add activemq-tcp:rceserver.example.com:20001 "Our RCE Server"). *<target>*: target of the connection *<description>*: description of the connection
cn list	Lists all network connections, including ids and connection states.
cn start <id>	Starts/connects a READY or DISCONNECTED connection (use "cn list" to get the id). *<id>*: id of the connection
cn stop <id>	Stops/disconnects an ESTABLISHED connection (use "cn list" to get the id). *<id>*: id of the connection

The 'components' command

Command	Description
components [--local|-l] [--remote|-r] [--as-table|-t]	Alias for "components list". *[--local|-l]*: only list components provided by the local node *[--remote|-r]*: only list components provided by the remote node(s) *[--as-table|-t]*: format the output as a table that is especially suited for automated parsing
components list [--local|-l] [--remote|-r] [--as-table|-t]	Show available components; by default, components on the local node as well as those published by a reachable remote node are listed. *[--local|-l]*: only list components provided by the local node *[--remote|-r]*: only list components provided by the remote node(s) *[--as-table|-t]*: format the output as a table that is especially suited for automated parsing
components list-auth	Shows a list of all defined authorization settings. Note that these settings are independent of whether a matching component exists, which means that settings are kept when a component is removed and later added again.

Command	Description
components set-auth <component id> <groups…>	Assigns a list of authorization groups to a component id; note that authorization settings always apply to all components with using this id, regardless of the component's version. *<component id>*: A component's id as listed by the "components list" command, e.g. "rce/Parametric Study", "common/MyIntegratedTool", or "cpacs/MyCpacsTool". *<groups…>*: list of authorization groups
components show <component id>	Show component definition. *<component id>*: A component's id as listed by the "components list" command, e.g. "rce/Parametric Study", "common/MyIntegratedTool", or "cpacs/MyCpacsTool".

The 'explain' command

Command	Description
explain	Show tokens.

The 'help' command

Command	Description
help <command group> [--details|-d] [--dev] [--asciidoc]	List available commands. *<command group>*: (optional) the command group of which the commands should be shown *[--details|-d]*: show details of the commands *[--dev]*: show dev commands *[--asciidoc]*: output in asciidoc format

The 'keytool' command

Command	Description
keytool ssh-pw	Generates a password for an SSH or Uplink connection, and the corresponding server entry.
keytool uplink-pw	Generates a password for an SSH or Uplink connection, and the corresponding server entry.

The 'mail' command

Command	Description
mail <recipient> <subject> <body>	Send an email. *<recipient>*: recipient of the e-mail *<subject>*: subject of the e-mail *<body>*: body of the e-mail

The 'net' command

Command	Description
net	Alias for 'net info'.
net filter	Show IP filter status.

Command	Description
net info	Show a list of reachable RCE nodes.
net reload-filter	Reloads the IP filter configuration.

The 'ra-admin' command

Command	Description
ra-admin list-wfs	Lists the ids of all published workflows.
ra-admin publish-wf <workflow file> <workflow id> [-g <group name>] [-p <JSON placeholder file>] [--keep-data|-k] [--temporary|-t]	Publishes a workflow file for remote execution via "ra run-wf" using <id>. *<workflow file>*: the file of the workflow *<workflow id>*: id of the workflow *[-g <group name>]*: set group name *[-p <JSON placeholder file>]*: JSON placeholder file *[--keep-data|-k]*: workflow data will not be deleted *[--temporary|-t]*: automatically unpublish when RCE shuts down
ra-admin unpublish-wf <workflow id>	Unpublishes (removes) the workflow file with id <id> from remote execution. *<workflow id>*: id of the workflow

The 'restart' command

Command	Description
restart	Restart RCE.

The 'saveto' command

Command	Description
saveto <output file> <command>	Save the command output to a file. *<output file>*: file to which the output will be written *<command>*: command whos output will be saved, does not have to be surounded by quotation marks

The 'shutdown' command

Command	Description
shutdown	Shut down RCE.

The 'ssh' command

Command	Description
ssh	Short form of "ssh list".
ssh add <display name> <host> <port> <username> <key file location>	Add a new ssh connection. *<display name>*: display name for the ssh connection *<host>*: host for the ssh connection *<port>*: port for the ssh connection

Command	Description
	<username>: username for the ssh connection *<key file location>*: location of the key file
ssh list	Lists all ssh connections, including ids and connection states.
ssh start <id>	Starts/connects an ssh connection (use " ssh list" to get the id). *<id>*: id for the ssh connection
ssh stop <id>	Stops/disconnects an ssh connection (use " ssh list" to get the id). *<id>*: id for the ssh connection

The 'stop' command

Command	Description
stop	Shut down RCE (alias of "shutdown").

The 'sysmon' command

Command	Description
sysmon [--local|-l] [--remote|-r]	Basic system-monitoring information. *[--local|-l]*: prints system monitoring data for the local instance *[--remote|-r]*: fetches system monitoring data from all reachable nodes in the network, and prints it in a human-readable format
sysmon api <operation: default|avgcpu+ram> <time span> <time limit>	Fetches system monitoring data from all reachable nodes in the network,and prints it in a parser-friendly format. *<operation: default|avgcpu+ram>*: operation to perform; avgcpu+ram: fetches the average CPU load over the given time span and the current free RAM *<time span>*: the maximum time span (in seconds) to aggregate load data over *<time limit>*: the maximum time (in milliseconds) to wait for each node’s load data response
sysmon remote	Fetches system monitoring data from all reachable nodes in the network, and prints it in a human-readable format.

The 'uplink' command

Command	Description
uplink	Short form for "uplink list".
uplink add <display name> <host> <port> <username> <key file location> <client id> <is gateway>	Add a new uplink connection. *<display name>*: display name *<host>*: host ip adress *<port>*: port for the ssh connection *<username>*: username

Command	Description
	<key file location>: location of the keyfile *<client id>*: id of the client *<is gateway>*: controls the isGateway property of the uplink
uplink list	Lists all uplink connections, including ids and connection states.
uplink start <id>	Starts/connects an uplink connection (use " uplink list" to get the id). *<id>*: id of the uplink
uplink stop <id>	Stops/disconnects an uplink connection (use " uplink list" to get the id). *<id>*: id of the uplink

The 'version' command

Command	Description
version [--detailed|-d]	Print version information. *[--detailed|-d]*: Show detailed information about the version

The 'wf' command

Command	Description
wf	Alias for "wf list".
wf cancel <id>	Cancel a running or paused workflow. *<id>*: id of the workflow
wf delete <id>	Delete and dispose a finished, cancelled or failed workflow. *<id>*: id of the workflow
wf details <id>	Show details of a workflow. *<id>*: id of the workflow
wf dispose <id>	Disposes a finished, cancelled or failed workflow. *<id>*: id of the workflow
wf list	Show workflow list.
wf open <id>	Open a runtime viewer of a workflow. Requires GUI. *<id>*: id of the workflow
wf pause <id>	Pause a running workflow. *<id>*: id of the workflow
wf resume <id>	Resume a paused workflow. *<id>*: id of the workflow
wf run <workflow file> [--delete <onfinished|never|always>] [--dispose <onfinished|never|	Starts a workflow from the given file and waits for its completion.

Command	Description
always>] [-p <JSON placholder file>] [--compact-output|-c]	*<workflow file>*: path to the workflow file *[--delete <onfinished|never|always>]*: deletion behaviour *[--dispose <onfinished|never|always>]*: dispose behaviour *[-p <JSON placholder file>]*: JSON placeholder file *[--compact-output|-c]*: This is a command flag
wf start <workflow file> [--delete <onfinished|never|always>] [--dispose <onfinished|never|always>] [-p <JSON placholder file>] [--compact-output|-c]	Starts a workflow from the given file and returns its workflow id if validation passed. *<workflow file>*: path to the workflow file *[--delete <onfinished|never|always>]*: deletion behaviour *[--dispose <onfinished|never|always>]*: dispose behaviour *[-p <JSON placholder file>]*: JSON placeholder file *[--compact-output|-c]*: This is a command flag
wf verify <workflows…> [--delete <onfinished|never|always>] [--dispose <onfinished|never|always>] [--pr <parallel runs>] [--sr <sequential runs>] [-p <JSON placholder file>] [--basedir <base directory>] [--includedirs <include directory…>]	Batch test the specified workflow files. *<workflows…>*: list of workflow files *[--delete <onfinished|never|always>]*: deletion behaviour *[--dispose <onfinished|never|always>]*: dispose behaviour *[--pr <parallel runs>]*: number of parallel runs *[--sr <sequential runs>]*: number of sequential runs *[-p <JSON placholder file>]*: JSON placeholder file *[--basedir <base directory>]*: optional base directory *[--includedirs <include directory…>]*: optional include directorys

The 'wf-integrate' command

Command	Description
wf-integrate <toolname> <workflow file> [--expose <ComponentName:OutputName:ExposedName, …>] [--expose-inputs <ComponentName:OutputName:ExposedName, …>] [--expose-outputs <ComponentName:OutputName:ExposedName, …>] [--verbose|-v]	Integrate a workflow file as a component. *<toolname>*: name for the tool *<workflow file>*: workflow to be integrated *[--expose <ComponentName:OutputName:ExposedName, …>]*: elements of the workflow to expose (see User Guide) *[--expose-inputs <ComponentName:OutputName:ExposedName, …>]*: elements of the workflow to expose (see User Guide) *[--expose-outputs <ComponentName:OutputName:ExposedName, …>]*: elements of the workflow to expose (see User Guide) *[--verbose|-v]*: enable verbose output

3.7.2. Parameters

RCE commands can use three different types of parameters. These are positional parameters, named parameters and flags. Positional and named parameters take at least one value. However, some of them also take a list parameters. List parameters are denoted by three dots after their name in the tables above, while parameters that take only a single value do not have it, e.g. **<parameter…>**. Multiple values for list parameters are entered using comma separation, e.g. **value0, value1, value2**.

Positional Parameters

Positional parameters must be entered immediately after a command. The order of the parameters must be followed for the command to process the parameters correctly. After all positional parameters have been entered, other types of parameters can be entered.

Named Parameters

Named parameters are optional parameters. The command input of these parameters always start with a double dash, e.g. **--name**. The value must be specified after the parameter name.

Command Flags

Command flags are used to modify the operation of a command. They are always optional. Flags start with a dash or a double dash. A single dash is used for flags consisting of only a single character, e.g. **-a**. Double dash is used for flags composed of more than one letter, e.g. **--all**. This long form is more descriptive and is therefore easier to understand. Command flags can have both long and short form, e.g. **-a** and **--all**, which do not differ in execution.

3.7.2.1. Configuration Placeholder Value Files

Some workflow components use placeholders for configuration values. The values for the placeholders are defined at workflow start. When executing workflows from the command line (e.g. in headless or batch mode), the placeholder's values must be defined in a file, which will be passed to the command with the -p option. Placeholder value files have following format:

```
{
 <component id>/<component version> : {
  <configuration placeholder id> : <configuration value>
 },
 <component id>/<component version>/<component instance name> : {
  <configuration placeholder id> : <configuration value>Usage
 }
}
```

> **Note**
>
> Every id and every value must be in enclosed in double quotes ("…").

The component id is the id string of a component (e.g. de.rcenvironment.script), the component version is the version of the component that is used in the workflow (e.g. 3.4). There are two ways of defining values for configuration placeholders: per component type and per component instance. When defined per component type, the id and version must be specified (e.g. "de.rcenvironment.script/3.4"). When defined per component instance the component id, component version, and the name of the component in the workflow must be specified (e.g. "de.rcenvironment.inputprovider/MyFile"). In both cases, the configuration placeholder id, which is the name of the configuration placeholder, and the actual configuration value must be specified. Component instance values override component type values.

> **Note**
>
> It is possible to mix component type and component instance values.

Below is an example placeholder value file, which defines one placeholder value (component type) for the input provider component and a placeholder value (component instance) for a specified input provider component of the workflow:

```
{
 "de.rcenvironment.inputprovider/3.2": {
  "inputFile": "C:/input/globalInputFile.txt"
 },
 "de.rcenvironment.inputprovider/3.2/Provider 1" : {
  "inputFile": "C:/input/Provider1.txt"
 }
}
```

The following table lists components and their configuration placeholders.

Component	Component ID and Version	Configuration Placeholders
Cluster	de.rcenvironment.cluster/3.1	authuser - user name authphrase - password (base64 encoded)
Input Provider	de.rcenvironment.inputprovider/3.2	<output name> - value of output
Output Writer	de.rcenvironment.outputwriter/2.1	targetRootFolder - path to target root folder
Script	de.rcenvironment.script/3.5	pythonExecutionPath - path to the Python executable (only required if Python is set as script language)

3.8. The Event Log

The RCE *Event Log* provides a high-level summary of events that are relevant to users and administrators of RCE instances.

3.8.1. Design Principles

Unlike a low-level technical log file, the Event Log is meant to aggregate events as much as possible. Where a technical log may list several steps related to an action or event, the Event Log is meant to focus on semantic results and final events, and condense them into a minimal number of entries.

For example, a technical log would typically represent the related events "user requested pause", followed by "pausing", and then "paused" as separate entries, often with other events mixed in-between.

The Event Log, in contrast, would represent this example as a single "paused" event. The details surrounding this event are collected and represented as attributes. For example, the "paused" event could have a "reason" attribute with a value of "initiated by user", and an optional "initiated by" attribute providing the user’s identity information. This design allows users to quickly understand what is happening in the system without associating multiple log events.

Of course, not all events can be merged this way. For example, the start of a network connection should be logged once it is established, as it is unknown when it will terminate. In these such cases, providing one or more unique association id(s) as event attribute(s) is recommended. If the event log is consumed using specialized clients, events can be automatically associated using these ids. Even if the event log is consumed without such tools, these association ids can still be easily found by using text-based search or filtering.

3.8.2. Event Structure and File Representation

In the current implementation, all event entries are appended to two files in the profile directory.

The `events-readable.log` file is meant for human inspection, and provides more explicit titles for event types and attributes. This file can be read in RCE by opening it via "File > Open", or with any external text file reader just like a `.txt` file.

The `events-compact.log` file, on the other hand, is meant for automated parsing. It uses technical ids for the event types and their attributes, and represents each event as a single line of UTF-8 JSON.

Event timestamps are represented differently in those files. In `events-readable.log`, times are formatted in a human-readable way, in the form of `<YYYY>-<MM>-<DD>T<hh>:<mm>:<ss>.<mmm>`. This format is based on ISO 8601, with milliseconds accuracy, using "unqualified local time", i.e., no time zone information but using local time. The latter choice was made to improve human readability when inspecting raw event data, at the cost of potential "jumps in time" on changes in the local UTC offset, e.g., "winter" time vs. "summer" time/DST. In `events-compact.log`, on the other hand, timestamps are encoded in the "_ts" JSON field as milliseconds from 1970-01-01T00:00:00Z.

Unlike other RCE log files, these files are intentionally not cleared on the restart of an instance, in order to provide a long-term protocol of system activity. For example, if an RCE instance is running as a system daemon, even repeated restarts of the host system will not erase older event data.

Note that both file formats are *preliminary*, and the scope of event types and attributes is not fully implemented yet. Notably, most workflow and component/tool execution events are not being logged yet.

3.8.3. Events Types and Attributes

This section lists the specified event types and their possible attributes. The human-readable titles of the types and attributes (listed first) are what appears in the `events-readable.log` file. Their technical ids (shown in a smaller font and parantheses) are used in the `events-compact.log` file, with the intention of more long-term stability.

Event Type (Title / Id / Description)	**Attribute (Title / Id)**	**Attribute Description**
Application Starting *(application.starting)* The application is starting and has passed basic initialization, including profile selection.	User home directory *(user_home)*	The "home" directory of the user running the application.
	User system id *(user_name)*	The technical/system name of the user running the application.
	Profile location *(profile_location)*	The file system location of the profile that the application is using.
	Working directory *(work_dir)*	The selected "working directory" when starting the application.
	RCE version id *(rce_version)*	The full RCE version string.
	OS name and version *(os_name)*	The name of the underlying operating system, as reported by the JRE.
	JVM version *(jvm_version)*	The Java version, as reported by the JRE.
Application Shutdown Requested *(application.shutdown.requested)* A user or technical process has requested this instance	Shutdown trigger *(method)*	The kind of event that initiated/requested the shutdown. Currently not strictly specified, for informational use only. Current examples are "console

Event Type (Title / Id / Description)	Attribute (Title / Id)	Attribute Description
to shut down, e.g. via GUI, console command, or system shell command. If available, information about the cause should be logged as "Shutdown trigger"/"method".		command", "CLI/network signal", and null (unspecified) *(optional)*
	Part of a restart *(is_restart)*	"yes" if the shutdown request is part of a restart request, i.e., a new application session will start after shutdown. Possible values: *yes*, *no*
Application Terminating *(application.terminating)* A marker event logged at the latest possible time during a regular shutdown of the application.	- *(no attributes)*	
Server Port Opened *(serverport.opened)* A network server port (typically TCP) was opened by the application.	Port type *(type)*	The technical type and/or protocol of the server port. Possible values: *localnet*, *ssh/uplink*
	IP port number *(port)*	The IP port number. Possible values: *Integer (1-65535)*
	IP bind address *(bind_ip)*	The IP address the port is bound to, which affects from which network interfaces it can be accessed.
Server Port Closed *(serverport.closed)* A network server port (typically TCP) was closed by the application.	Port type *(type)*	The technical type and/or protocol of the server port. Possible values: *localnet*, *ssh/uplink*
	IP port number *(port)*	The IP port number. Possible values: *Integer (1-65535)*
	IP bind address *(bind_ip)*	The IP address the port was bound to.
Incoming Connection Accepted *(connection.incoming.accepted)* An incoming connection has been successfully established. This should be logged as late as reasonably possible to avoid "accepted" events where the connection is immediately closed again due to a validation or version mismatch error. Mutually exclusive with the "connection.incoming.refused" event, but may be reached after one or more "connection.incoming.auth.failed"	Connection type *(type)*	The technical type and/or protocol of the connection. Usually equal to the server port's type. Possible values: *localnet*, *ssh/uplink*
	Connection id *(connection_id)*	An association id for this connection. The only assumptions that should be made about its content is that it is a string of "reasonable" length, not empty, suitable for log output, and unique within the application's session.

Event Type (Title / Id / Description)	Attribute (Title / Id)	Attribute Description
events if the server allows multiple login attempts.	Client software id *(client_version)*	The software version of the connecting client, if available (e.g., "rce/…").
	Remote node id *(remote_node_id)*	The id of the remote RCE node, if available from the protocol type's connection process. This id can by any of the four supported id types, but will typically be an "instance session id". *(optional)*
	Login/user name *(login_name)*	The user/login/account identifier successfully used for authentication/authorization, if applicable. *(optional)*
	Authentication method *(auth_method)*	The kind of authentication/ authorization (e.g. passphrase or private key) that was successfully used, if applicable. *(optional)*
	Failure count *(auth_failure_count)*	The number of failed authentication/authorization attempts for this connection, if applicable. *(optional)*
	Remote IP address *(remote_ip)*	The remote IP address of the incoming connection.
	Remote IP port *(remote_port)*	The remote IP port number of the incoming connection.
	Server IP port *(server_port)*	The local server IP port number that the incoming connection is connected to.
Incoming Connection Failed To Authenticate *(connection.incoming.auth.failed)* As part of establishing an incoming connection, an authentication/authorization attempt was made but failed. Whether this failure is fatal for the overall connection attempt or not is implementation-specific. If this is ultimately followed up by a successful authentication/authorization attempt (in case another attempt was allowed in the first place), this event should be succeeded by a "connection.incoming.accepted"	Connection type *(type)*	The technical type and/or protocol of the connection. Usually equal to the server port's type. Possible values: *localnet*, *ssh/ uplink*
	Connection id *(connection_id)*	An association id for this connection. The only assumptions that should be made about its content is that it is a string of "reasonable" length, not empty, suitable for log output, and unique within the application's session.
	Client software id *(client_version)*	The software version of the connecting client, if available (e.g., "rce/…").

Event Type (Title / Id / Description)	Attribute (Title / Id)	Attribute Description
event. Otherwise, this should always be succeeded by a "connection.incoming.refused" event.	Login/user name *(login_name)*	The user/login/account identifier used for the failed authentication/authorization attempt.
	Authentication method *(auth_method)*	The kind of authentication/ authorization (e.g. passphrase or private key) that was attempted.
	Failure reason *(auth_failure_reason)*	The reason why authentication/ authorization failed (e.g., wrong password or unknown user).
	Failure count *(auth_failure_count)*	The number of failed authentication/authorization attempts for this connection.
	Remote IP address *(remote_ip)*	The remote IP address of the incoming connection.
	Remote IP port *(remote_port)*	The remote IP port number of the incoming connection.
	Server IP port *(server_port)*	The local server IP port number that the incoming connection is connected to.
Incoming Connection Refused *(connection.incoming.refused)* An incoming connection has failed to complete its login process or has been refused for some other reason. Mutually exclusive with the "connection.incoming.accepted" and "connection.incoming.closed" events. Both incorrect authentication attempts as well as authentication timeouts (e.g. when an SSH client makes no authentication attempt at all) are both represented by this event. These sub-types can be distinguished by the "last_auth_failure_reason" attribute.	Connection type *(type)*	The technical type and/or protocol of the connection. Usually equal to the server port's type. Possible values: *localnet, ssh/ uplink*
	Connection id *(connection_id)*	An association id for this connection. The only assumptions that should be made about its content is that it is a string of "reasonable" length, not empty, suitable for log output, and unique within the application's session.
	Client software id *(client_version)*	The software version of the connecting client, if available (e.g., "rce/…").
	Reason *(close_reason)*	The human-readable reason why this connection was refused.
	Last login/user name *(last_login_name)*	The user/login/account identifier used for the last failed authentication/authorization attempt, if applicable. *(optional)*
	Last auth. failure *(last_auth_failure_reason)*	The last reason why authentication/authorization failed (e.g., wrong password or unknown user), if applicable. *(optional)*

Event Type (Title / Id / Description)	Attribute (Title / Id)	Attribute Description
	Last auth. method *(last_auth_method)*	The last kind of authentication/ authorization (e.g. passphrase or private key) that was attempted, if applicable. *(optional)*
	Auth. failure count *(auth_failure_count)*	The number of failed authentication/authorization attempts for this connection, if applicable. *(optional)*
	Remote IP address *(remote_ip)*	The remote IP address of the incoming connection.
	Remote IP port *(remote_port)*	The remote IP port number of the incoming connection.
	Server IP port *(server_port)*	The local server IP port number that the incoming connection is connected to.
	Duration *(duration)*	The duration (in msec) that this connection was open/ established for. The precise start and end times for this calculation are implementation- and type-dependent.
Incoming Connection Closed *(connection.incoming.closed)* An incoming connection has been closed, either by the client, the server, or a network event. This event is always preceded by a "connection.incoming.accepted" event. Mutually exclusive with the "connection.incoming.refused" event.	Connection type *(type)*	The technical type and/or protocol of the connection. Usually equal to the server port's type. Possible values: *localnet, ssh/ uplink*
	Connection id *(connection_id)*	An association id for this connection. The only assumptions that should be made about its content is that it is a string of "reasonable" length, not empty, suitable for log output, and unique within the application's session.
	Client software id *(client_version)*	The software version of the connecting client, if available (e.g., "rce/…").
	Remote node id *(remote_node_id)*	The id of the remote RCE node, if available from the protocol type's connection process. This id can by any of the four supported id types, but will typically be an "instance session id". *(optional)*
	Reason *(close_reason)*	The human-readable reason why this connection was closed.

Event Type (Title / Id / Description)	Attribute (Title / Id)	Attribute Description
	Remote IP address *(remote_ip)*	The remote IP address of the incoming connection.
	Remote IP port *(remote_port)*	The remote IP port number of the incoming connection.
	Server IP port *(server_port)*	The local server IP port number that the incoming connection is connected to.
	Duration *(duration)*	The duration (in msec) that this connection was open/ established for. The precise start and end times for this calculation are implementation- and type-dependent.
Incoming Uplink Connection Accepted *(uplink.incoming.accepted)* After the login credentials for an incoming SSH connection were accepted, this subsequent event indicates successful completion of the Uplink protocol handshake, too. This includes protocol compatibility validation and the successful assignment of an unused Uplink namespace.	Uplink session id *(session_id)*	The Uplink-specific session id.
	SSH connection id *(connection_id)*	The SSH connection id for correlation with "connection.incoming.*" events.
	Client software id *(client_version)*	The software version of the Uplink client; includes information about the client software used (e.g., "rce/…").
	Protocol version *(protocol_version)*	The Uplink protocol version being used for the connection/ session, based on the initial client-server handshake.
	Effective login name *(login_name)*	The final/effective login name used. Due to namespace mapping constraints, this may be different from the original login name; see "original_login_name".
	Original login name *(original_login_name)*	The login name requested by the client, before any modifications by the server. Omitted if equal to the effective login name (see "login_name"). *(optional)*
	Effective client id *(client_id)*	The final/effective client id used. Due to namespace mapping constraints, this may be different from the original client id; see "original_client_id".
	Original client id *(original_client_id)*	The client id requested by the client, before any modifications by the server. Omitted if equal to the effective client id (see "client_id"). *(optional)*

Event Type (Title / Id / Description)	Attribute (Title / Id)	Attribute Description
	Assigned namespace *(namespace)*	The namespace assigned to this client for Uplink destination address mapping. Typically related to the session id.
Incoming Uplink Connection Refused *(uplink.incoming.refused)* After the login credentials for an incoming SSH connection were accepted, this subsequent event indicates failure of the Uplink protocol handshake. This may be due to a version incompatibility or the desired namespace already being used. The next event logged after this should be "connection.incoming.closed", as the SSH connection was already "accepted". Mutually exclusive with "uplink.incoming.accepted" and "uplink.incoming.closed".	Uplink session id *(session_id)*	The Uplink-specific session id.
	SSH connection id *(connection_id)*	The SSH connection id for correlation with "connection.incoming.*" events.
	Client software id *(client_version)*	The software version of the Uplink client; includes information about the client software used (e.g., "rce/…").
	Protocol version *(protocol_version)*	The protocol version requested by the client. May or may not be a version that this server supports.
	Original login name *(login_name)*	The login name used by the client, without any mapping modification that would have been made on success.
	Original client id *(client_id)*	The client id sent by the client, without any mapping modification that would have been made on success.
	Reason *(reason)*	The human-readable reason for refusing the Uplink session.
Incoming Uplink Connection Closed *(uplink.incoming.closed)* After a previous "uplink.incoming.accepted" event, this indicates the end of the application-level Uplink session. This event should be logged for any kind of connection termination, from graceful disconnect to low-level connection errors. Mutually exclusive with "uplink.incoming.refused".	Uplink session id *(session_id)*	The Uplink-specific session id.
	SSH connection id *(connection_id)*	The SSH connection id for correlation with "connection.incoming.*" events.
	Final connection state *(final_state)*	The final (technical) state of the Uplink connection; indicates the reason for terminating the session.
Network Node Discovered/ Named *(network.node.named)* Either a new node in the logical RCE network has been discovered, or an existing node has changed its title. Relevant for determining which technical node was behind a given	Persistent instance id *(instance_id)*	The persistent part (the "instance node id") of the logical RCE-specific network address.
	Instance session id *(session_id)*	The per-session suffix of the of the logical RCE-specific network address. Used to distinguish runs of the same instance, i.e., the session id

Event Type (Title / Id / Description)	Attribute (Title / Id)	Attribute Description
display name at a certain time. Besides general logging, this is also a basic security trail for potential node "impersonation" attempts.		always changes on an instance's restart.
	Logical sub-node id *(logical_sub_node_id)*	The logical "sub-node" selector within the logical RCE-specific network address. If absent, then this event represents a change of the "root" name of an instance/node. If present, then a logical "sub-node" has changed its specific name, while the "root" name was unchanged. *(optional)*
	Announced name *(name)*	The title/name of the RCE instance, as defined by its user or administrator.
	Is local/own node *(is_local_node)*	\"yes\" if the observed node is the local node. This is a convenience property to simplify event filtering. Possible values: *yes*, *no* *(optional, derived)*
Workflow Request Initiated (Request Sent) *(workflow.request.initiated)* A workflow run was initiated from the local node. The designated workflow controller may be either the local or a remote node.	Workflow run id *(workflow_run_id)*	The id string assigned to this workflow run. Can be considered globally unique for all practical purposes.
	Workflow controller node *(workflow_controller_node)*	The logical node id of the node designated to run the workflow controller.
	Controller is local *(workflow_controller_is_local_node)*	A convenience attribute designating whether the workflow controller node is the local node. Possible values: *yes*, *no* *(derived)*
	Workflow file path *(local_workflow_file)*	The path of the local workflow file that was submitted for execution, if applicable; OTHERWISE absent. *(optional)*
	Workflow metadata (WIP) *(workflow_metadata)*	TODO specify contents; structured data
	success *(success)*	Whether the workflow was successfully initiated, i.e., whether the request was made and accepted by the designated execution node. Possible values: *yes*, *no*
Workflow Execution Requested (Request Received) *(workflow.execution.requested)*	Workflow run id *(workflow_run_id)*	The id string assigned to this workflow run. Can be considered globally unique for all practical purposes.

Event Type (Title / Id / Description)	**Attribute (Title / Id)**	**Attribute Description**
A workflow run was requested, and an attempt was made to initialize its workflow controller on the local node. If controller initialization fails, this event MUST still be logged to make the request visible, as the ".initiated" event may have been logged be on a remote node.	Start timestamp *(start_ts)*	On success, the canonical start time of the workflow, which will usually be close, but not necessarily equal to the timestamp of this event; OTHERWISE absent. (TODO specify format) *(optional)*
	Initiator node *(initiator_node)*	The logical session node id of the node that initiated this workflow run. Note that in RCE 10.x and earlier, this value is NOT strongly verified, and should be considered informational only.
	Initiator is local *(initiator_is_local_node)*	A convenience attribute designating whether the node that initiated this workflow run is the local node. Note that in RCE 10.x and earlier, this value is NOT strongly verified, and should be considered informational only. Possible values: *yes*, *no* *(derived)*
	Workflow metadata (WIP) *(workflow_metadata)*	TODO specify contents; structured data
	success *(success)*	Whether the workflow was successfully initialized. Possible values: *yes*, *no*
Workflow Execution Completed *(workflow.execution.completed)* The end of a local workflow controller's execution. (TODO clarify whether this may be logged on a failed ".requested" event or not.).	Workflow run id *(workflow_run_id)*	The id string assigned to this workflow run. Can be considered globally unique for all practical purposes.
	End timestamp *(end_ts)*	The canonical end time of the workflow, which will usually be close, but not necessarily equal to the timestamp of this event. (TODO specify format)
	Duration *(duration)*	A convenience attribute specifying the duration of this workflow run. (TODO specify format) *(derived)*
	Final workflow state *(final_state)*	The final state of the workflow, as defined by the workflow engine. Possible values: *FINISHED*, *CANCELLED*, *FAILED*, *RESULTS_REJECTED*
Workflow Request Completed *(workflow.request.completed)*	Workflow run id *(workflow_run_id)*	The id string assigned to this workflow run. Can be

Event Type (Title / Id / Description)	Attribute (Title / Id)	Attribute Description
The final event of a workflow run that was initiated by the local node. (TODO clarify whether this may be logged on a failed ".initiate" event or not.).		considered globally unique for all practical purposes.
	Final workflow state *(final_state)*	The final state of the workflow, as defined by the workflow engine. Possible values: *FINISHED*, *CANCELLED*, *FAILED*, *RESULTS_REJECTED*
Component/Tool Execution Initiated (Request Sent) *(component.request.initiated)* A workflow component run was initiated from the local node, which is always the workflow controller. The node controlling the component's execution may be the local or a remote one.	Workflow run id *(workflow_run_id)*	The id string assigned to this workflow run. Can be considered globally unique for all practical purposes.
	Component run id *(component_run_id)*	The id string assigned to this component run. Unique within the scope of the associated workflow.
	Executing node *(execution_controller_node)*	The logical session node id of the node designated to be this component run's execution controller.
	Executing node is local *(execution_controller_is_local_node)*	A convenience attribute designating whether this component's run execution controller node is the local node. Possible values: *yes*, *no* *(derived)*
Component/Tool Execution Requested *(component.execution.requested)* A workflow component run was requested, and an attempt was made to initialize its controller on the local node. If controller initialization fails, this event MUST still be logged to make the request visible, as the ".initiated" event may have been logged be on a remote workflow controller node.	Workflow run id *(workflow_run_id)*	The id string assigned to this workflow run. Can be considered globally unique for all practical purposes.
	Component run id *(component_run_id)*	The id string assigned to this component run. Unique within the scope of the associated workflow.
	Workflow controller node *(workflow_controller_node)*	The logical session node id of the node running the workflow controller. Note that in RCE 10.x and earlier, this value is NOT strongly verified, and should be considered informational only.
	Workflow contr. is local *(workflow_controller_is_local_node)*	A convenience attribute designating whether the workflow controller node is the local node. Note that in RCE 10.x and earlier, this value is NOT strongly verified, and should be considered informational only.
	Start timestamp *(start_ts)*	On success, the canonical start time of the component run,

Event Type (Title / Id / Description)	Attribute (Title / Id)	Attribute Description
		which will usually be close, but not necessarily equal to the timestamp of this event; OTHERWISE absent. (TODO specify format) *(optional)*
Component/Tool Execution Completed *(component.execution.completed)* The end of a local workflow controller's execution. (TODO clarify whether this may be logged on a failed ".requested" event or not.).	Workflow run id *(workflow_run_id)*	The id string assigned to this workflow run. Can be considered globally unique for all practical purposes.
	Component run id *(component_run_id)*	The id string assigned to this component run. Unique within the scope of the associated workflow.
	End timestamp *(end_ts)*	The canonical end time of the component run, which will usually be close, but not necessarily equal to the timestamp of this event. (TODO specify format)
	Duration *(duration)*	A convenience attribute specifying the wall-clock duration of this component run. (TODO specify format) *(derived)*
	Final component state *(final_state)*	The final state of the component, as defined by the workflow engine. Possible values: *FINISHED*, *CANCELLED*, *FAILED*, *RESULTS_REJECTED*
Component/Tool Execution Request Completed *(component.request.completed)* The final event of a component run that was initiated by a workflow controller running on the local node. (TODO clarify whether this may be logged on a failed ".initiate" event or not.).	Workflow run id *(workflow_run_id)*	The id string assigned to this workflow run. Can be considered globally unique for all practical purposes.
	Component run id *(component_run_id)*	The id string assigned to this component run. Unique within the scope of the associated workflow.
	End timestamp *(end_ts)*	The canonical end time of the component run, which will usually be close, but not necessarily equal to the timestamp of this event. (TODO specify format)
	Duration *(duration)*	A convenience attribute specifying the wall-clock duration of this component run. (TODO specify format) *(derived)*

Event Type (Title / Id / Description)	Attribute (Title / Id)	Attribute Description
	Final component state *(final_state)*	The final state of the component, as defined by the workflow engine. Possible values: *FINISHED*, *CANCELLED*, *FAILED*, *RESULTS_REJECTED*
Login/Account Data Initialized *(accounts.initialized)* A set of accounts has been initialized, typically at application or subsystem startup.	Account type *(type)*	The type of accounts that were initialized, e.g. "ssh". Possible values: *ssh*
	New number of accounts *(number_of_accounts)*	Indicates the total number of registered accounts after initialization.
	Account data origin *(origin)*	Indicates the data source from which the initial account data was read. Absent if not applicable for the current account type. *(optional)*
Login/Account Data Updated *(accounts.updated)* A set of accounts has been updated at application runtime, after being already initialized.	Account type *(type)*	The type of accounts that were updated, e.g. "ssh". Possible values: *ssh*
	New number of accounts *(number_of_accounts)*	Indicates the total number of registered accounts after the update (NOT the number of changes!).
	Account data origin *(origin)*	Indicates the data source from which the updated account data was read. Absent if not applicable for the current account type. *(optional)*
System Monitoring Initialized *(sysmon.initialized)* Indicates that the system monitoring subsystem was initialized and logs static system information.	Java process id *(jvm_pid)*	The PID of the main process running the JVM. Depending on the operating system, this process may either be the RCE executable or a separate Java process.
	Processor count (JVM) *(jvm_processor_count)*	The number of processors, as reported by the JVM.
	Heap limit (JVM) *(jvm_heap_limit)*	The configured heap (Java RAM) limit, as reported by the JVM.
	System RAM (native) *(system_total_ram)*	The total system RAM in bytes, as reported by the system monitoring library.
	Logical CPUs (native) *(system_logical_cpus)*	The number of "logical CPUs", as reported by the system monitoring library.
Custom Event *(custom)*	Custom event type id *(type)*	A custom type id describing the logged event. It should generally follow the naming

Event Type (Title / Id / Description)	Attribute (Title / Id)	Attribute Description
A custom event type, allowing extensions or plugins to make use of the event logging system. This event type is special in the way that its list of attributes is not pre-defined, except for "type". This allows custom events to log all kinds of extra attributes. Consequently, any event log validation code must have a special rule to accept those attributes. Note that while custom attribute keys are supported, there are still certain rules for them. For now, they must start with a-z, end with a-z or a digit, and not exceed the maximum length defined above. It is also recommended to keep them lowercase and dot-separated for consistency.		pattern of the standard events, and plugins/extensions should strive to make those ids collision-free.

Note

It is possible that event types not listed here were/are/will be logged by past, current, or future RCE versions. Therefore, any automated code for parsing event log data MUST tolerate event types not listed here.

Appendix A. Script API Reference

This section contains a reference for the API that is accessible via the script component.

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)

Method	Description
`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 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)