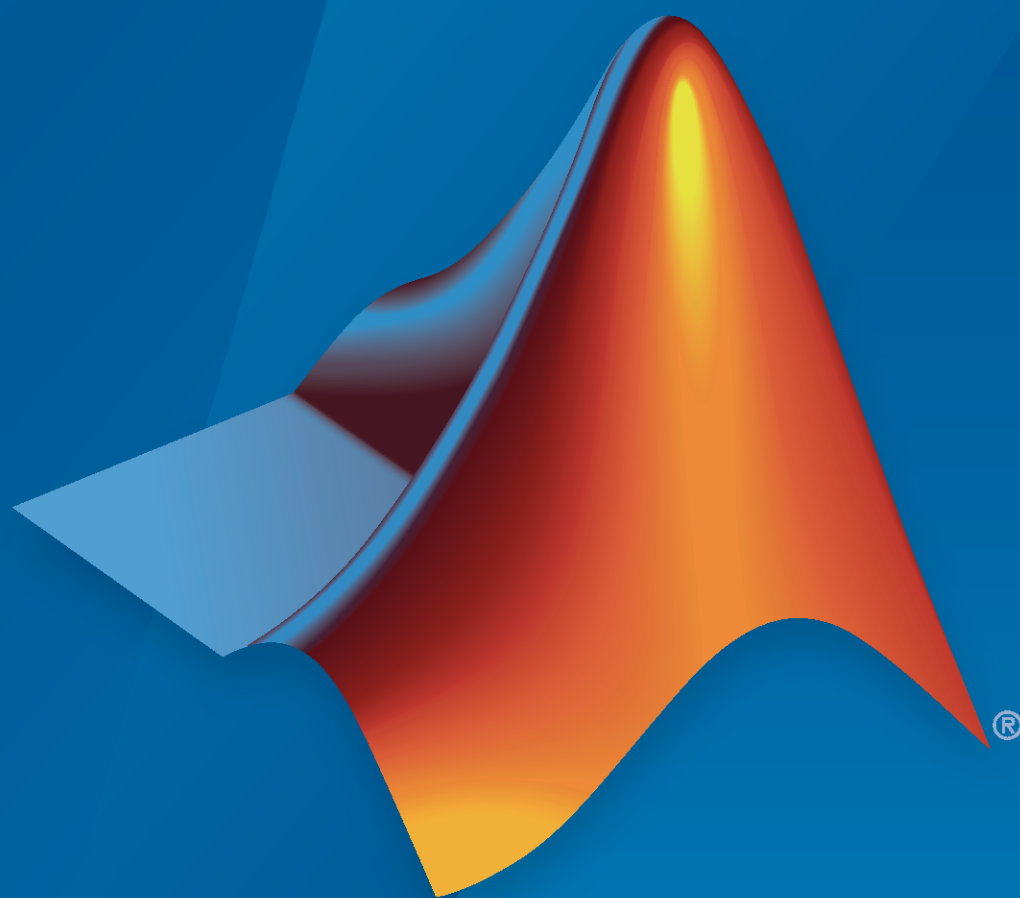


CI/CD Automation for Simulink® Check™ Support Package

Reference Book



MATLAB® & SIMULINK®

R2022b — R2024a



How to Contact MathWorks



Latest news: www.mathworks.com
Sales and services: www.mathworks.com/sales_and_services
User community: www.mathworks.com/matlabcentral
Technical support: www.mathworks.com/support/contact_us



Phone: 508-647-7000



The MathWorks, Inc.
1 Apple Hill Drive
Natick, MA 01760-2098

CI/CD Automation for Simulink® Check™ Reference Book PDF

© COPYRIGHT 2022-2024 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

Revision History

August 2022	PDF Only	Version 22.1.0 (R2022a)
September 2022	PDF Only	Version 22.1.1
October 2022	PDF Only	Versions 22.1.2 and 22.2.2 (R2022b)
November 2022	PDF Only	Versions 22.1.3 and 22.2.3
December 2022	PDF Only	Versions 22.1.4 and 22.2.4
February 2023	PDF Only	Versions 22.1.5 and 22.2.5
March 2023	PDF Only	Version 23.1.5 (R2023a)
April 2023	PDF Only	Versions 22.1.6, 22.2.6, and 23.1.6
June 2023	PDF Only	Versions 22.1.7, 22.2.7, and 23.1.7
July 2023	PDF Only	Versions 22.1.8, 22.2.8, and 23.1.8
August 2023	PDF Only	Versions 22.2.9, 23.1.9, and 23.2.0 (R2023b)
September 2023	PDF Only	Versions 22.1.9, 22.2.10, and 23.1.10
October 2023	PDF Only	Versions 22.1.10, 22.2.11, 23.1.11, and 23.2.1
November 2023	PDF Only	Versions 22.1.11, 22.2.12, 23.1.12, and 23.2.2
December 2023	PDF Only	Versions 22.1.12, 22.2.13, 23.1.13, and 23.2.3
February 2024	PDF Only	Versions 22.1.13
March 2024	PDF Only	Versions 22.2.14, 23.1.14, and 23.2.4
April 2024	PDF Only	Version 24.1.1 (R2024a)

1	<u>Reference Book</u>
2	<u>Process Modeling System API</u>
3	<u>Build System API</u>
4	<u>Pipeline Generator API</u>
5	<u>Report Generator API</u>
6	<u>Utilities</u>
7	<u>Process Advisor Example Projects</u>
8	<u>Artifact Types</u>
9	<u>Tokens</u>

Check Coding Standards or Prove Code Quality	10-3
Prerequisites	10-3
Add Task to Process	10-4
Reconfigure Task	10-4
Source Code	10-10
Check Modeling Standards	10-11
Add Task to Process	10-11
Reconfigure Task	10-11
Source Code	10-16
Detect Design Errors	10-17
Add Task to Process	10-17
Reconfigure Task	10-17
Source Code	10-19
Generate Code	10-20
Add Task to Process	10-20
Reconfigure Task	10-20
Source Code	10-22
Generate Model Comparison	10-23
Prerequisites	10-23
Add Task to Process	10-23
Reconfigure Task	10-23
Launch Tool Action	10-24
Source Code	10-25
Generate SDD Report	10-26
Prerequisites	10-26
Add Task to Process	10-26
Reconfigure Task	10-26
Source Code	10-28
Generate Simulink Web View	10-30
Prerequisites	10-30
Add Task to Process	10-30
Reconfigure Task	10-30
Source Code	10-32
Inspect Code	10-33
Add Task to Process	10-33
Reconfigure Task	10-33
Source Code	10-34
Merge Test Results	10-35
Prerequisites	10-35
Add Task to Process	10-35
Reconfigure Task	10-35
Source Code	10-39

Run Tests (per model)	10-40
Add Task to Process	10-40
Reconfigure Task	10-40
Source Code	10-43
Run Tests (per test case)	10-44
Add Task to Process	10-44
Reconfigure Task	10-44
Source Code	10-45

Built-In Query Library

11

padv.builtin.query.FindArtifacts	11-3
Syntax	11-3
Input Arguments	11-3
Methods	11-4
Use in Process Model	11-4
Test Outside Process Model	11-4
padv.builtin.query.FindCodeForModel	11-6
Syntax	11-6
Input Arguments	11-6
Methods	11-6
Use in Process Model	11-6
padv.builtin.query.FindExternalCodeCache	11-9
Syntax	11-9
Input Arguments	11-9
Methods	11-9
Use in Task Definition	11-9
Test Query from Command Window	11-10
padv.builtin.query.FindFilesWithLabel	11-11
Syntax	11-11
Input Arguments	11-11
Methods	11-12
Use in Process Model	11-12
padv.builtin.query.FindFileWithAddress	11-13
Syntax	11-13
Input Arguments	11-13
Methods	11-14
Use in Process Model	11-14
Test Outside Process Model	11-15
padv.builtin.query.FindMAJustificationFileForModel	11-16
Syntax	11-16
Input Arguments	11-16
Use in Process Model	11-16

padv.builtin.query.FindModels	11-18
Syntax	11-18
Input Arguments	11-18
Methods	11-19
Use in Process Model	11-19
Test Outside Process Model	11-20
padv.builtin.query.FindModelsWithLabel	11-21
Syntax	11-21
Input Arguments	11-21
Methods	11-22
Use in Process Model	11-22
padv.builtin.query.FindModelsWithTestCases	11-23
Syntax	11-23
Input Arguments	11-23
Methods	11-23
Use in Process Model	11-24
padv.builtin.query.FindProjectFile	11-25
Syntax	11-25
Methods	11-25
Use in Process Model	11-25
padv.builtin.query.FindRefModels	11-26
Syntax	11-26
Input Arguments	11-23
Methods	11-26
Use in Process Model	11-27
padv.builtin.query.FindRequirements	11-28
Syntax	11-28
Input Arguments	11-28
Methods	11-29
Use in Process Model	11-29
padv.builtin.query.FindRequirementsForModel	11-30
Syntax	11-30
Input Arguments	11-30
Methods	11-31
padv.builtin.query.FindTestCasesForModel	11-32
Syntax	11-32
Input Arguments	11-32
Methods	11-33
Use in Process Model	11-33
padv.builtin.query.FindTopModels	11-34
Syntax	11-34
Input Arguments	11-23
Methods	11-34
Use in Process Model	11-35
padv.builtin.query.GetDependentArtifacts	11-36
Syntax	11-36

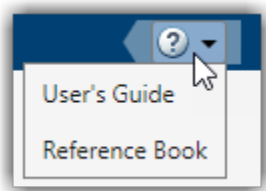
Methods	11-36
Use in Task	11-36
padv.builtin.query.GetIterationArtifact	11-38
Syntax	11-38
Methods	11-38
Use in Task	11-38
padv.builtin.query.GetOutputsOfDependentTask	11-40
Syntax	11-40
Input Arguments	11-40
Methods	11-40
Use in Task	11-40

Reference Book

This PDF is a Reference Book with information on the API, artifact types, built-in tasks, and built-in queries.

For examples and general information, see the User's Guide PDF. You can access the PDFs from either:

- <https://www.mathworks.com/matlabcentral/fileexchange/115220-ci-cd-automation-for-simulink-check>
- The question mark icon in the Process Advisor app



Process Modeling System API

The support package provides a customizable process modeling system that you can use to define your build and verification process. You define your pipeline of tasks in the process model. The process model is a file (`processmodel.p` or `processmodel.m`) that specifies the tasks in the process, the queries that determine which artifacts to use for each task, the artifacts associated with each task, and the dependencies between tasks. Open the Process Advisor app or use the function `createprocess` to create a process model for your project. Inside the process model file, you can add, remove, and reconfigure tasks and the dependencies between tasks.

For examples of how to create a process model, see the "Author Your Process Model" chapter in the User's Guide PDF.

Classes

Class	Description
<code>padv.Artifact</code>	Store artifact information
<code>padv.ProcessModel</code>	Define tasks and process for project
<code>padv.Query</code>	Select set of artifacts from project
<code>padv.Subprocess</code>	Group tasks
<code>padv.Task</code>	Define single step in process
<code>padv.TaskResult</code>	Create and access results from task

Functions

Create and Access Process Model

Function	Description
<code>createprocess</code>	Create a process model
<code>getprocess</code>	Get process model object for process model in project

createprocess

Create process model

Syntax

```
processModelPath = createprocess()  
processModelPath = createprocess(Name=Value)
```

Description

`processModelPath = createprocess()` creates a process model at the project root and returns the path to the created process model. The process model is saved as `processmodel.m`.

By default, the process model is a default process model that can create a model-based design pipeline. You can only call `createprocess` if you have a project open.

`processModelPath = createprocess(Name=Value)` specifies the output process model using one or more `Name=Value` arguments.

Examples

Create Process Model

Open a project that does not have a process model and use `createprocess` to create a copy of the default process into the project.

Open an example project, for example `matlab.project.example.timesTable`, that does not have a process model.

Create a process model for the project.

```
processModelPath = createprocess
```

`createprocess` copies the default process model into the project root and saves the path to the process model to `processModelPath`.

Create a project object for the currently loaded project.

```
myProject = currentProject;
```

Add the process model file to the current project.

```
addFile(myProject,processModelPath)
```

Open the Process Advisor app in a standalone window to view the tasks associated with the project and project artifacts.

```
processAdvisorWindow
```

Overwrite Process Model with Empty Process

Open a project and overwrite the process model with an empty process model.

Open the **Process Advisor** example project, which contains an example process model.

```
processAdvisorExampleStart
```

Use `createprocess` to overwrite the existing process model with an empty process model.

```
processModelPath = createprocess(Template="empty",Overwrite=true)
```

Open the created process model to view the commented-out example code.

```
open(processModelPath)
```

Input Arguments

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `processModelPath = createprocess(Overwrite=true)`

Template — Name of predefined process model template

"default" (default) | "empty" | "parallel"

Name of predefined process model template, specified as either:

- "default" — Process model file that includes several built-in tasks
- "empty" — Process model file that contains commented-out example code for adding built-in and custom tasks
- "parallel" — Process model file designed for parallel CI jobs. For information, see "Parallel Pipeline Architectures".

Example: "empty"

Data Types: char | string

Overwrite — Setting to overwrite existing process model

false or 0 (default) | true or 1

Setting to overwrite existing process model, specified as a numeric or logical 0 (false) or 1 (true).

Example: true

Data Types: logical

Output Arguments

processModelPath — Path to created process model

character vector

Path to created process model, returned as a character vector.

By default, `createprocess` creates a process model at the project root.

Alternative Functionality

App

If a project does not have a process model, you can use the Process Advisor app to create the default process model. To open the Process Advisor app for a project, in the MATLAB® Command Window, enter:

```
processAdvisorWindow
```

When you open the Process Advisor app on a project that does not have a process model, the app automatically creates a copy of the default process model at the root of the project.

getprocess

Get process model object for process model in project

Syntax

```
processModelObject = getprocess()
```

Description

`processModelObject = getprocess()` returns a process model object, `processModelObject`, for the process model in the project. You can use the process model object to view the properties of the process model in the project. For more information, see the documentation for "padv.ProcessModel" in this PDF.

If the current project does not have a process model, the function `getprocess` automatically creates a new process model at the root of the project.

Examples

Find the Default Query for the Current Process

Use `getprocess` to find the default query that the current process model uses. If you have a task that does not specify an iteration query, the default query defines which artifacts the process iterates over. By default, custom tasks run once per project because the default query is "padv.builtin.query.FindProjectFile".

Open the **Process Advisor** example project, which contains an example process model.

```
processAdvisorExampleStart
```

Get the properties of the current process model.

```
currentProcessModel = getprocess()
```

Get the default query for the current process model.

```
defaultQuery = currentProcessModel.DefaultQueryName
```

```
defaultQuery =
```

```
    "padv.builtin.query.FindProjectFile"
```

You can use the `findTask` and `findQuery` functions on the loaded process model to find specific tasks and queries in the process.

```
findTask(currentProcessModel, "padv.builtin.task.RunModelStandards")
```

Output Arguments

processModelObject — Properties of process model

`padv.ProcessModel` object

Properties of process model, returned as a `padv.ProcessModel` object.

The `padv.ProcessModel` object returns the names of the tasks, queries, default query, and root process model file for the process.

padv.Artifact

Store artifact information

Description

A `padv.Artifact` object represents an artifact that you can run a task on in the process defined in your process model. For example, you can use a `padv.Artifact` object as the input to functions like `runprocess` and `generateProcessTasks` when you only want to run or generate tasks associated with a specific artifact.

Creation

Syntax

```
artifactObject = padv.Artifact(artifactType,artifactAddress)
artifactObject = padv.Artifact( ____,Name=Value)
```

Description

`artifactObject = padv.Artifact(artifactType,artifactAddress)` stores artifact information in a `padv.Artifact` object, `artifactObject`. You can use the artifact information when you want to get the ID for a specific task iteration.

`artifactObject = padv.Artifact(____,Name=Value)` specifies the artifact using one or more `Name=Value` arguments.

Input Arguments

artifactType — Type of artifact

string

Type of artifact, specified as a string. For example:

- "sl_model_file" for Simulink® models
- "m_file" for MATLAB M files

For a list of valid artifact types, see the chapter "Artifact Types" in this PDF.

Example: "sl_model_file"

Example: "m_file"

Example: "sl_test_case"

Data Types: string

artifactAddress — Address of artifact

`padv.util.ArtifactAddress` object

Address of artifact, specified as an `padv.util.ArtifactAddress` object. Note that the address is relative to the project root.

Example:

```
padv.util.ArtifactAddress(fullfile("02_Models", "AHRV_Voter", "specification", "AHRV_Voter.slx"))
```

Data Types: `string`

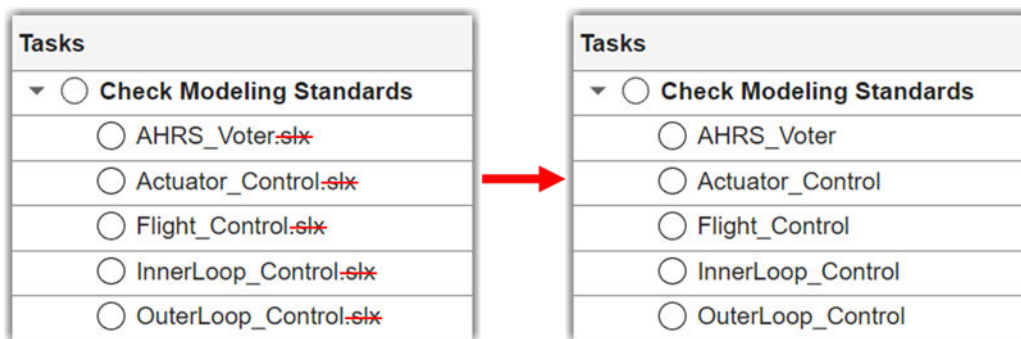
Properties

Alias — Human-readable name for artifact

empty string (default) | `string`

Human-readable name for the artifact in the Process Advisor user interface, specified as a string.

If you want to customize how artifact names appear in Process Advisor, create a custom query that updates the values of the `Alias` property for each `padv.Artifact` object that the query returns. For an example of how to update the alias to remove the `.slx` file extension for models shown in the **Tasks** column, see "Hide File Extension in Process Advisor".



Data Types: `string`

Type — Type of artifact

`string`

Type of artifact, specified as a string. For example:

- `"sl_model_file"` for Simulink models
- `"m_file"` for MATLAB M files

For a list of valid artifact types, see the chapter "Artifact Types" in this PDF.

Example: `"sl_model_file"`

Example: `"sl_test_case"`

Example: `"m_file"`

Data Types: `string`

Parent — Reference to parent artifact

`padv.Artifact` object

Reference to parent artifact, specified as a `padv.Artifact` object.

ArtifactAddress — Address of artifact in project

padv.util.ArtifactAddress object

Address of artifact in project, specified as a padv.util.ArtifactAddress object.

Object Functions

Object Function	Description
getTypes	Get artifact type. TYPES = getTypes(artifactObj)
getKey	Get unique key for artifact. A key is a unique address for a file. KEY = getKey(artifactObj)
hasType	Check if artifact has type. TYPE = hasType(artifactObj)

Examples**Run Task Associated with Model**

Suppose you have a process model with several tasks, but right now you only want to run test cases associated with a single model. You can use a padv.Artifact object to specify the model and use the runprocess function to run the test cases for that model.

Open the Process Advisor example project, which contains an example process model.

```
processAdvisorExampleStart
```

The example process contains a **Run Tests** task (padv.builtin.task.RunTestsPerTestCase) that runs the test cases in the project.

Create a padv.Artifact object that represents the model that you want to run. For this example, the artifact type is "sl_model_file" because the artifact is a Simulink model and the address is the path to model AHRS_Voter.slx, relative to the project root.

```
model = padv.Artifact(...
    "sl_model_file",...
    fullfile("02_Models", "AHRS_Voter", "specification", "AHRS_Voter.slx"));
```

Run the **Run Tests** task on the test cases associated with the model AHRS_Voter.slx by specifying the name-value arguments of the runprocess function.

```
runprocess(...
    Tasks = "padv.builtin.task.RunTestsPerTestCase",...
    FilterArtifact = model)
```

The build system only runs the test cases associated with the specified model.

padv.ProcessModel

Define tasks and process for project

Description

A `padv.ProcessModel` object represents the process model that defines the tasks and process for a project. A *task* performs an action and is a single step in your process. A *process* is a series of tasks that run in a specific order. The process model defines the tasks that you can perform on the project, and the order and relationships between tasks in the process. You can use tasks and queries to dynamically perform actions and find artifacts in the project. Use the `addTask` object function to add tasks to the process model. You can use the function `runprocess` to run the tasks defined in the process model. Certain `padv.ProcessModel` properties use tokens, like `$PROJECTROOT$`, as placeholders for dynamic path resolution during run-time. For information on the tokens, see the "Tokens" section in this PDF.

Creation

Syntax

```
pm = padv.ProcessModel()
```

Description

`pm = padv.ProcessModel()` creates an empty process model object, `pm`.

Properties

TaskNames — Tasks added to process model object

string array

Tasks added to process model object, returned as string array.

Use the object function `addTask` to add a task instance to a process model.

```
Example: ["padv.builtin.task.GenerateSimulinkWebView"  
"padv.builtin.task.RunModelStandards"]
```

Data Types: string

QueryNames — Queries added to process model object

string array

Queries added to process model object, returned as string array.

Use the object function `addQuery` to add a query instance to a process model.

```
Example: ["padv.builtin.query.FindModels" "padv.builtin.query.FindProjectFile"]
```

Data Types: string

DefaultQueryName — Default query for tasks added to process model object

"padv.builtin.query.FindProjectFile" (default) | name of padv.Query query

Default query for tasks added to process model, specified as the name of a padv.Query query.

Example: "padv.builtin.query.FindModels"

Data Types: string

DefaultQueryName — Name of default project query

"padv.builtin.query.FindProjectFile" (default) | task name or instance

Name of default project query, specified as a task name or padv.Task instance.

Example: "padv.builtin.query.FindModels"

DefaultOutputDirectory — Default output directory for results

fullfile("\$PROJECTROOT\$", "PA_Results") (default) | string array

Default output directory, specified as a string array. Set the default output directory to a path inside your project. The path can be either a relative or absolute path. Consider using the path relative to the project root to promote consistency across local environments and CI systems, and allow for more portable builds.

By default, Process Advisor and the build system output results in a folder PA_Results in the project root.

Example: fullfile("\$PROJECTROOT\$", "Process_Results")

Data Types: string

DefaultRootFileName — Default name of process model file

"processmodel.m" (default) | string

Default name of process model file, specified as a string.

Data Types: string

JUnitReportName — Name of generated JUnit-style XML report

"\$TASKNAME\$_\$ITERATIONARTIFACT\$_JUnit.xml" (default) | string array

Name of generated JUnit-style XML report, specified as a string array.

By default, the generated JUnit report for a task has the format *taskName_iterationArtifact_JUnit.xml*.

Example: "\$TASKNAME\$_\$ITERATIONARTIFACT\$_JUnitReport.xml"

Data Types: string

JUnitReportPath — Location for JUnit-style XML report

fullfile("\$DEFAULTOUTPUTDIR\$", "junit") (default) | string array

Location for JUnit-style XML report, specified as a string array.

Example: fullfile("\$DEFAULTOUTPUTDIR\$", "junit", "reports")

Data Types: string

RootFileName — Name of process model file

string

Name of process model file, returned as a string.

`RootFileName` uses `processmodel.m` as the name of the process model file, unless a `processmodel.p` file exists. If you have both a P-code file and a `.m` file, the P-code file takes precedence over the corresponding `.m` file for execution, even after modifications to the `.m` file.

The default name of the process model file is specified by `DefaultRootFileName`.

Data Types: string

Object Functions

reset	Removes tasks and queries from process model <code>pm = padv.ProcessModel();</code> <code>reset(pm);</code>
reload	Load process model by executing process model file for project <code>pm = padv.ProcessModel();</code> <code>reload(pm);</code>
addSubprocess	Add subprocess instance to process model <code>addSubprocess(pm, "MySubprocess");</code>
addTask	Add task instance to process model <code>addTask(pm, "MyCustomTask", Action=@SayHello, ...</code> <code>IterationQuery=padv.builtin.query.FindModels);</code> For information, see "addTask".
addQuery	Add query instance to process model <code>addQuery(pm, "MyCustomQuery")</code> For information, see "addQuery".
findQuery	Find query instance by name <code>pm = padv.ProcessModel();</code> <code>QUERY = findQuery(pm, ...</code> <code>"padv.builtin.query.FindModels")</code>
findTask	Find task instance by name <code>pm = padv.ProcessModel();</code> <code>TASK = findTask(pm, ...</code> <code>"padv.builtin.task.RunModelStandards");</code>
exists	Check if process model exists for project <code>[FOUND, PATH] = padv.ProcessModel.exists()</code>

Examples


Add Tasks to Process Model Object

You can use the object function `addTask` to add the tasks to a `padv.ProcessModel` object.

Open the Process Advisor example project.

```
processAdvisorExampleStart
```

The model `AHRS_Voter` opens with the Process Advisor pane to the left of the Simulink canvas.

In the Process Advisor pane, click the **Edit process model**  button to open the `processmodel.m` file for the project.

Replace the contents of the `processmodel.m` file with this code:

```
function processmodel(pm)
    arguments
        pm padv.ProcessModel
    end

    addTask(pm, "taskA");
    addTask(pm, "taskB");

end
```

The function `addTask` adds the task objects to the `padv.ProcessModel` object.

Use the function `getprocess` to get the process model object for the project.

```
pm = getprocess;
```

Get the task object for "taskA" defined in the current process model.

```
taskAObj = findTask(pm, "taskA");
```

`taskAObj` is a `padv.Task` object that you can use to view the properties of the task "taskA".

addQuery

Namespace: padv

Add query instance to process model

Syntax

```
queryObj = addQuery(pm, queryNameOrInstance)
queryObj = addQuery( ____, Name=Value)
```

Description

`queryObj = addQuery(pm, queryNameOrInstance)` adds the query specified by `queryNameOrInstance` to the process model. You can access the query using the query object `queryObj`.

`queryObj = addQuery(____, Name=Value)` specifies the properties of the query using one or more `Name=Value` arguments.

Input Arguments

pm — Process for project

`padv.ProcessModel` object (default) |

Process for project, specified as a `padv.ProcessModel` object.

Example: `pm = padv.ProcessModel`

queryNameOrInstance — Name or instance of query

string | `padv.Query` object

Name or instance of a query, specified as a string or `padv.Query` object.

Example: `"NameOfMyQuery"`

Example: `padv.builtin.query.FindModels`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, . . . , NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example:

DefaultArtifactType — Artifact type returned by query

`"padv_output_file"` (default) | valid value for the `Type` property of a `padv.Artifact` object

Artifact type returned by the query, specified as a valid value for the `Type` property of a `padv.Artifact` object.

Example: `DefaultArtifactType = "sl_model_file"`

Title — Human readable name

Name property of query (default) | string

Human readable name for the query, specified as a string. By default, the `Title` property of the query is the same as the `Name`.

Example: `Title = "My Query"`

Data Types: `string`

FunctionHandle — Handle to function that runs when you run query object`function_handle`

Handle to function that runs when you run query object, specified as a `function_handle`.

When you call the `run` function on a query object, `run` runs the function specified by the `function_handle`.

Example: `FunctionHandle = @FunctionForQuery`

Data Types: `function_handle`

Parent — Initial query run before iteration query`[0×0 string] (default) | padv.Query object | Name of padv.Query object`

Initial query run before iteration query, specified as either a `padv.Query` object or the `Name` of a `padv.Query` object. When you specify a `padv.Query` object as the iteration query for a task, the `Parent` query is the initial query that the build system runs before running the specified iteration query.

For example, the built-in `querypadv.builtin.query.FindModelsWithTestCases` has the `Parent` query `padv.builtin.query.FindModels`. If you specify `padv.builtin.query.FindModelsWithTestCases` as the iteration query for a task, you are specifying that you want the task to run once for each model with a test case. The build system runs the `Parent` query `padv.builtin.query.FindModels` first, to find the models in the project, and then the build system runs the iteration query `padv.builtin.query.FindModelsWithTestCases` to find the models with test cases.

The build system ignores the `Parent` query when you specify a query as an input query or dependency query for a task.

Example: `Parent = "padv.builtin.query.FindModels"`

SortArtifacts — Setting for automatically sorting artifacts by address`true or 1 (default) | false or 0`

Setting for automatically sorting artifacts by address, specified as a numeric or logical `1 (true)` or `0 (false)`. When a query returns artifacts, the artifacts should be in a consistent order. By default, the build system sorts artifacts by the artifact address.

Alternatively, you can sort artifacts in a different order by overriding the internal `sortArtifacts` method in a subclass that defines a custom sort behavior. For an example, see "Sort Artifacts in Specific Order" in the User's Guide PDF.

The build system automatically calls the `sortArtifacts` method when using the process model. The `sortArtifacts` method expects two input arguments: a `padv.Query` object and a list of

`padv.Artifact` objects returned by the `run` function. The `sortArtifacts` method should return a list of sorted `padv.Artifact` objects.

Example: `SortArtifacts = false`

Data Types: `logical`

Output Arguments

queryObj – Query object

`padv.Query` object

Query object, returned as a `padv.Query` object.

For more information, see the documentation for "`padv.Query`" in this PDF.

addTask

Namespace: padv

Add task instance to process model

Syntax

```
taskObj = addTask(pm,taskNameOrInstance)
taskObj = addTask( ____,Name=Value)
```

Description

`taskObj = addTask(pm,taskNameOrInstance)` adds the task specified by `taskNameOrInstance` to the process model. You can access the task using the task object `taskObj`.

`taskObj = addTask(____,Name=Value)` specifies the properties of the task using one or more `Name=Value` arguments.

Examples


Add Tasks to Process Model

You can use the `addTask` function to create function-based tasks directly in the process model.

Open the Process Advisor example project.

```
processAdvisorExampleStart
```

The model `AHRS_Voter` opens with the Process Advisor pane to the left of the Simulink canvas.

In the Process Advisor pane, click the **Edit process model**  button to open the `processmodel.m` file for the project.

Replace the contents of the `processmodel.m` file with this code:

```
function processmodel(pm)
    arguments
        pm padv.ProcessModel
    end

    addTask(pm,"MyCustomTask",Action=@SayHello,...
        IterationQuery=padv.builtin.query.FindModels);

end

function results = SayHello(~)
    disp("Hello, World!");
    results = padv.TaskResult;
    results.ResultValues.Pass = 1;
end
```

This code adds a task, `MyCustomTask` to the process model while specifying that the task runs the function `SayHello` one time for each model found in the project. The function `SayHello` also specifies the results returned by the task.

Input Arguments

pm – Process for project

`padv.ProcessModel` object (default)

Process for project, specified as a `padv.ProcessModel` object.

Example: `pm = padv.ProcessModel`

taskNameOrInstance – Name or instance of task

string | `padv.Task` object

Name or instance of a task, specified as a string or `padv.Task` object.

Example: `"NameOfMyTask"`

Example: `padv.builtin.task.RunModelStandards`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, . . . , NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example:

```
addTask(pm, "RunOnceForEachModel", IterationQuery=padv.builtin.query.FindModels)
```

Title – Human readable name that appears in Process Advisor app

Name property of task (default) | string

Human readable name that appears in the **Tasks** column of the Process Advisor app, specified as a string. By default, the Process Advisor app uses the `Name` property of the task as the `Title`.

Example: `"My Task"`

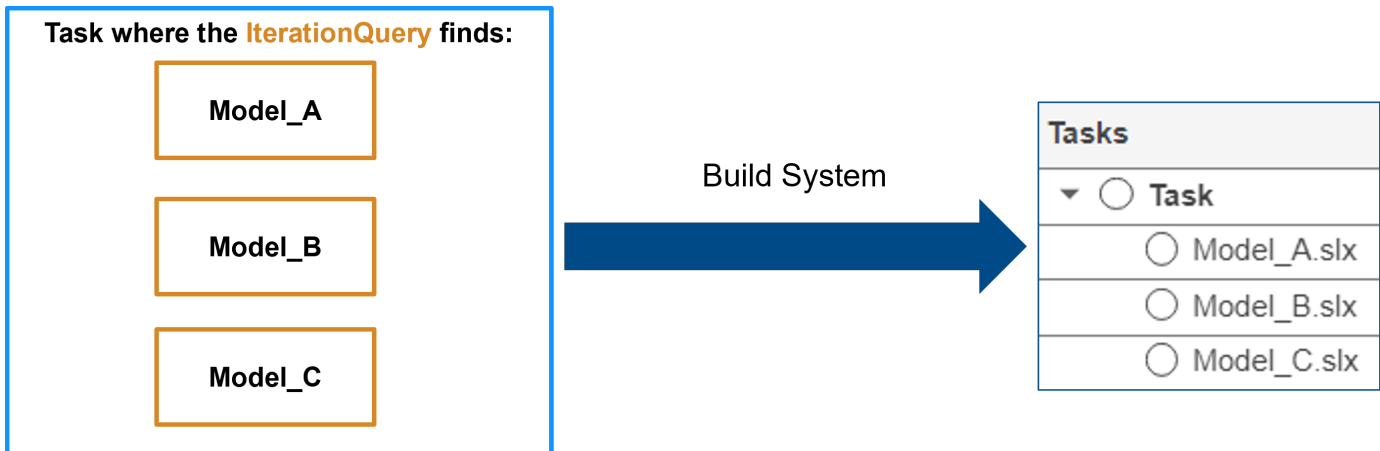
Data Types: `string`

IterationQuery – Artifacts that task iterates over

`padv.Query` object | name of `padv.Query` object

Artifacts that task iterates over, specified as a `padv.Query` object or the name of a `padv.Query` object. By default, task objects run one time and are associated with the project. When you specify `IterationQuery`, the task runs one time *for each* artifact specified by the `padv.Query`. In the Process Advisor app, the artifacts specified by `IterationQuery` appear under task title.

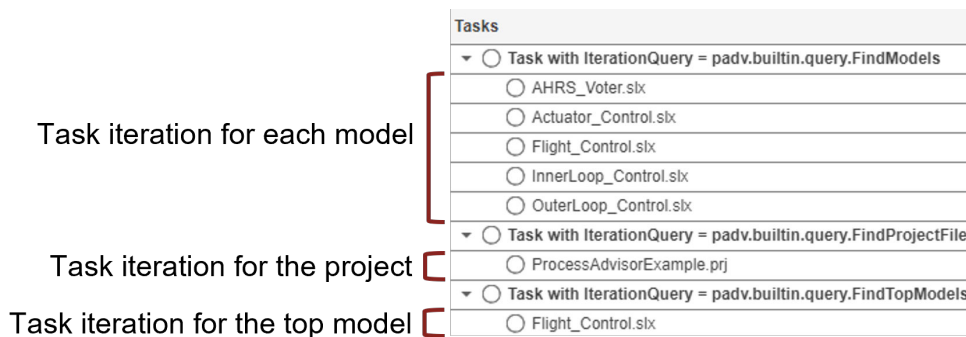
For example, if the `IterationQuery` for a task finds three models, `Model_A`, `Model_B`, and `Model_C`, the build system creates three task iterations under the title of the task in the **Tasks** column.



Each of the artifacts under the task title represents a *task iteration*.

For an example of the effect of different `IterationQuery` values:

- If you have a task where the `IterationQuery` uses `padv.builtin.query.FindModels` to find each of the models in the project, the build system creates a task iteration for each model.
- If you have a task where the `IterationQuery` uses `padv.builtin.query.FindProjectFile` to find the project file, the build system creates a task iteration for the project file.
- If you have a task where the `IterationQuery` uses `padv.builtin.query.FindTopModels` to find top models in the project, the build system creates a task iteration for each top model.



Example: `IterationQuery = padv.builtin.query.FindModels`

Data Types: `string`

InputQueries – Inputs to task

`padv.Query` object | name of `padv.Query` object | array of `padv.Query` objects

Inputs to the task, specified as:

- a `padv.Query` object
- the name of `padv.Query` object
- an array of `padv.Query` objects
- an array of names of `padv.Query` objects

By default, the task does not specify any artifacts as inputs. When you specify `InputQueries`, the task tasks the artifacts specified by the specified query or queries as an input.

Suppose a task runs once for each model in the project and you want to provide the models as inputs to the task. If you specify `InputQueries` as the built-in query `padv.builtin.query.GetIterationArtifact`, the query returns each artifact that the tasks iterates over, which in this example is each of the models in the project.

Example: `InputQueries = padv.builtin.query.GetIterationArtifact`

InputDependencyQuery – Artifact dependencies for task inputs

`padv.Query` object | name of `padv.Query` object

Artifact dependencies for task inputs, specified as a `padv.Query` object or the name of a `padv.Query` object.

The build system runs the query specified by `InputDependencyQuery` to find the dependencies for the task inputs, since those dependencies can affect whether task results are up-to-date. Typically, you specify `InputDependencyQuery` as `padv.builtin.query.GetDependentArtifacts` to get the dependent artifacts for each task input. For example, if you specify a model as an input to a task and you specify `InputDependencyQuery` as `padv.builtin.query.GetDependentArtifacts`, the build system can find artifacts, such as data dictionaries, that the model uses.

Example: `InputDependencyQuery = padv.builtin.query.GetDependentArtifacts`

Action – Function that task runs

function handle

Function that the task runs, specified as the function handle. When you run the task, the task runs the function specified by the function handle.

For example, if you want the task to run the function `myFunction`, specify `Action` as `@myFunction`.

Example: `Action = @myFunction`

Data Types: `function_handle`

RequiredIterationArtifactType – Artifact type that task can run on

string

Artifact type that the task can run on, specified by a string. The required iteration artifact type must be the artifact type supported by the `IterationQuery` property of the task.

For a list of valid artifact types, see the chapter "Artifact Types" in this PDF.

Example: `RequiredIterationArtifactType = "sl_model_file"`

Data Types: `string`

Licenses – List of licenses that task requires

string array

List of licenses that the task requires, specified as a string array.

Example: `Licenses = ["matlab_report_gen" "simulink_report_gen"]`

Data Types: `string`

AllLicenseRequired — Setting to require all licenses for task`true` or `1` (default) | `false` or `0`

Setting to require all licenses for task, specified as a numeric or logical `1` (`true`) or `0` (`false`). By default, all licenses in the `Licenses` property of the task are required for the task to run. Specify `0` (`false`) if the task can run without all licenses listed in the `Licenses` property.

Example: `AllLicenseRequired = false`

Data Types: `logical`

DescriptionText — Task description`string`

Task description, specified as a string.

Example: "This task runs myScript."

Data Types: `string`

DescriptionCSH — Path to task documentation`string`

Path to task documentation, specified as a string.

Example: `DescriptionCSH =
fullfile(pwd, "taskHelpFiles", "myTaskDocumentation.pdf")`

Data Types: `string`

Output Arguments**taskObj — Task object**`padv.Task` object

Task object, returned as a `padv.Task` object.

For more information, see the documentation for "padv.Task" in this PDF.

padv.Query

Select set of artifacts from project

Description

A `padv.Query` object represents a query that you can use to select a set of artifacts from a project. Use the input arguments to define the set of artifacts that the query selects. Queries can either be function-based or class-based. Use `FunctionHandle` to specify a function for a function-based query or use inheritance for a class-based query.

Creation

Syntax

```
Q = padv.Query(Name)
Q = padv.Query( ____, Name = Value)
```

Description

`Q = padv.Query(Name)` creates a query object with the name `Name`.

`Q = padv.Query(____, Name = Value)` specifies query properties using one or more name-value arguments. For example, `DefaultArtifactType = "sl_model_file"` changes the default artifact type for the query from a generic output file, `"padv_output_file"`, to a model file, `"sl_model_file"`.

Input Arguments

Name — Unique identifier for query

character vector | string

Unique identifier for query, specified as character vector or string. You can only specify a query name when you create a query object. You cannot change the query name after you create the query object.

Each query in the process model must have a unique name.

Example: `"CustomQueryForArtifacts"`

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `DefaultArtifactType = "sl_model_file"`

Title — Human-readable name for query

character vector | string

Human-readable name for query, specified as character vector or string.

Example: `Title = "Custom Query for Artifacts"`

Data Types: `char` | `string`

DefaultArtifactType — Expected artifact type

`"padv_output_file"` (default) | valid value for the `Type` property of a `padv.Artifact` object

Expected artifact type, specified as a valid value for the `Type` property of a `padv.Artifact` object. `padv.Task` objects use the `DefaultArtifactType` to confirm that the artifacts output by the query are the types of artifacts required by the `padv.Task` object.

When you use the `run` function on a query object, the `DefaultArtifactType` is the default value for artifacts returned by the function.

Example: `DefaultArtifactType = "sl_model_file"`

Parent — Initial query run before iteration query

`padv.Query` object | Name of `padv.Query` object

Initial query run before iteration query, specified as either a `padv.Query` object or the Name of a `padv.Query` object. When you specify a `padv.Query` object as the iteration query for a task, the Parent query is the initial query that the build system runs before running the specified iteration query.

For example, the built-in query `padv.builtin.query.FindModelsWithTestCases` has the Parent query `padv.builtin.query.FindModels`. If you specify `padv.builtin.query.FindModelsWithTestCases` as the iteration query for a task, you are specifying that you want the task to run once for each model with a test case. The build system runs the Parent query `padv.builtin.query.FindModels` first, to find the models in the project, and then the build system runs the iteration query `padv.builtin.query.FindModelsWithTestCases` to find the models with test cases.

The build system ignores the Parent query when you specify a query as an input query or dependency query for a task.

Example: `"padv.builtin.query.FindModels"`

ShowFileExtension — Show file extensions for returned artifacts

`0` (false) | `1` (true)

Show file extensions in the `Alias` property of returned artifacts, specified as a numeric or logical `1` (true) or `0` (false). The `Alias` property controls the display name for the artifact in the **Tasks** column in Process Advisor.

By default, queries strip file extensions from the `Alias` property of each task iteration artifact. To show file extensions for all artifacts in the **Tasks** column, select the project setting **Show file extensions**. To keep file extensions in the results for a specific query, specify the query property `ShowFileExtension` as `true`.

Example: `true`

Data Types: `logical`

SortArtifacts — Setting for automatically sorting artifacts by address

`true` or `1` (default) | `false` or `0`

Setting for automatically sorting artifacts by address, specified as a numeric or logical 1 (`true`) or 0 (`false`). When a query returns artifacts, the artifacts should be in a consistent order. By default, the build system sorts artifacts by the artifact address.

Alternatively, you can sort artifacts in a different order by overriding the internal `sortArtifacts` method in a subclass that defines a custom sort behavior. For an example, see "Sort Artifacts in Specific Order" in the User's Guide PDF.

The build system automatically calls the `sortArtifacts` method when using the process model. The `sortArtifacts` method expects two input arguments: a `padv.Query` object and a list of `padv.Artifact` objects returned by the run function. The `sortArtifacts` method should return a list of sorted `padv.Artifact` objects.

Example: `SortArtifacts = false`

Data Types: `logical`

FunctionHandle — Handle to function that runs when you run query object

`function_handle`

Handle to function that runs when you run query object, specified as a `function_handle`.

When you call the run function on a query object, run runs the function specified by the `function_handle`.

Example: `FunctionHandle = @FunctionForQuery`

Data Types: `function_handle`

run

Return artifacts from query

Syntax

```
artifacts = run(queryObj)
artifacts = run(queryObj, inputArtifact)
```

Description

`artifacts = run(queryObj)` returns the artifacts in the project folder that match the criteria specified by the query `queryObj`.

Typically, you use queries inside your process model and the build system automatically runs the queries as needed, but you can use the `run` function to run a query outside of your process model to confirm which artifacts the query returns. For examples of how to run specific built-in queries, see "Built-In Query Library".

`artifacts = run(queryObj, inputArtifact)` returns the artifacts in the project folder that match the criteria specified by the query `queryObj` and are associated with the artifact `inputArtifact`. If you use the query as an iteration query or dependency query, the build system can use `inputArtifact` to determine the scope of the artifacts that the query returns, which can be helpful for queries that need an input artifact from a parent query.

Examples

Test Query Outside Process Model

Although you typically use queries inside your process model, you can run queries outside of your process model to confirm which artifacts the query returns.

- 1 Open a project. For this example, you can open the Process Advisor example project.

```
processAdvisorExampleStart
```

- 2 Create an instance of a query. For this example, you can create an instance of the built-in query `padv.builtin.query.FindArtifacts`. You can use the arguments of the query to filter the query results. For example, you can use the `IncludeLabel` argument to have the query only return artifacts that use the `Design` project label from the `Classification` project label category.

```
q = padv.builtin.query.FindArtifacts(...
IncludeLabel = {'Classification', 'Design'});
```

For a list of the built-in queries, see "Built-In Query Library". If your use case requires custom queries instead, see "Create Custom Query" in the User's Guide PDF.

- 3 Run the query and inspect the array of artifacts that the query returns.

```
artifacts = run(q)
```

```
artifacts =  
    1x24 Artifact array with properties:  
        Type  
        Parent  
        ArtifactAddress  
        Alias
```

Input Arguments

queryObj — Query object

`padv.Query` object | built-in query object

Query object, specified as a `padv.Query` object, built-in query object, or any object whose class that inherits from the `padv.Query` class or a built-in query class.

For information on the built-in queries, see "Built-In Query Library". If your use case requires custom queries instead, see "Create Custom Query" in the User's Guide PDF.

Example: `q = padv.Query("myQueryName")`

Example: `q = padv.builtin.query.FindArtifacts`

inputArtifact — Input artifact that query needs

`padv.Artifact`

Input artifact that the query needs, specified as a `padv.Artifact` object.

Output Arguments

artifacts — Artifacts that query returns

`padv.Artifact`

Artifacts that query returns, returned as an array of `padv.Artifact` objects.

padv.Subprocess

Group tasks

Description

Creation

A `padv.Subprocess` object represents a group of tasks in a `padv.ProcessModel` process. In your process model, use the object functions `addTask` and `addSubprocess` to group tasks and subprocesses inside your subprocess. You can use the object functions `dependsOn` and `runsAfter` to specify the dependencies and desired execution order for a subprocess.

Properties

Title — Human readable name that appears in Process Advisor app

string

Human readable name that appears in the **Tasks** column of the Process Advisor app, returned as a string. By default, the Process Advisor app uses the `Name` property of the task as the `Title`.

Example: `padv.Task("myTask", Title = "My Task")`

Data Types: string

DescriptionText — Task description

string

Task description, returned as a string.

Example: `padv.Task("myTask", DescriptionText = "This is my task.")`

Data Types: string

DescriptionCSH — Path to task documentation

string

Path to task documentation, returned as a string.

Example: `padv.Task("myTask", DescriptionCSH = fullfile(pwd, "taskHelpFiles", "myTaskDocumentation.pdf"))`

Data Types: string

RequiredIterationArtifactType — Artifact type that subprocess can run on

string

Artifact type that the subprocess can run on, returned as a string. The required iteration artifact type must be the artifact type supported by the `IterationQuery` property of the subprocess. For a list of valid artifact types, see the chapter "Artifact Types" in this PDF.

Data Types: `string`

LaunchToolAction — Function that launches a tool

function handle

Function that launches a tool, returned as the function handle.

When the property `LaunchToolAction` is specified, you can point to the task in the Process Advisor app and click the ellipsis (...) and then **Open Tool Name** to open the tool associated with the task.

For tasks that are not built-in tasks, the task options show the ellipsis (...) and then **Launch Tool**.

Example: `padv.Task("myTask", LaunchToolAction = @openTool)`

Data Types: `function_handle`

LaunchToolText — Description of action that LaunchToolAction property performs

"Launch Tool" (default) | `string scalar`

Description of the action that the `LaunchToolAction` property performs, returned as a string scalar.

Example: `padv.Task("myTask", LaunchToolAction = @openTool, LaunchToolText = "Open tool.")`

Data Types: `string`

Enabled — Controls if the padv.Task is enabled in the process model

`true` or `1` (default) | `false` or `0`

Controls if the `padv.Task` is enabled in the process model, returned as a numeric or logical `1` (`true`) or `0` (`false`).

Example: `padv.Task("myTask", Enabled = false)`

Data Types: `logical`

OutputDirectory — Location for standard outputs that tasks in subprocess produce

`string`

Location for standard outputs that tasks in the subprocess produce, specified as a string.

Example: `fullfile("folder", "subfolder")`

Data Types: `string`

CacheDirectory — Location for additional cache files that tasks in subprocess produce

`string`

Location for additional cache files that tasks in the subprocess produce, specified as a string. The cache directory can contain temporary files that do not need to be either saved in the task results or archived by a CI system.

Example: `fullfile("folder", "subfolder")`

Data Types: `string`

Object Functions

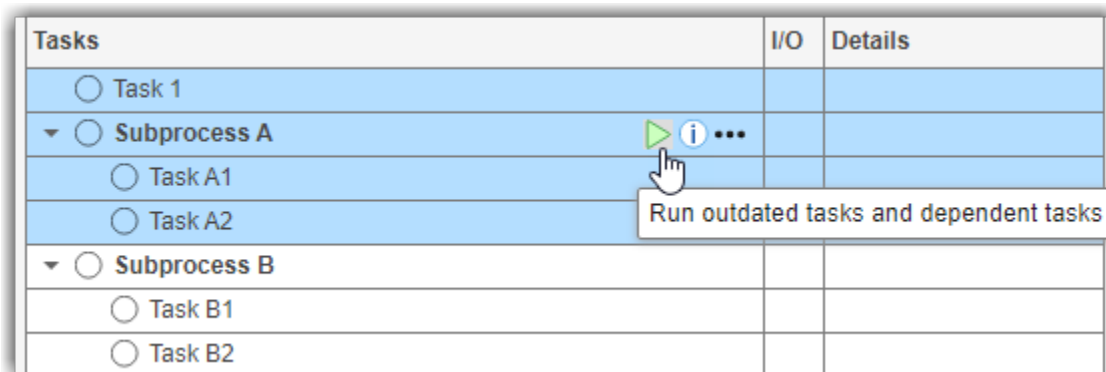
- `addTask(subprocessObject, taskNameOrInstance, NAME, VALUE, ...)`
- `addSubprocess(subprocessObject, subprocessNameOrInstance, NAME, VALUE, ...)`
- `dependsOn(subprocessObject, DEPENDENCIES, NAME, VALUE, ...)`
- `runsAfter(subprocessObject, PREDECESSORS, NAME, VALUE, ...)`

Examples

Group Tasks Inside Subprocess

You can use a subprocess to group related tasks, create a hierarchy of tasks, and share parts of a process. A *subprocess* is a self-contained sequence of tasks, inside a process or other subprocess, that can run standalone.

Tasks	I/O	Details
<input type="radio"/> Task 1		
▼ <input type="radio"/> Subprocess A		
<input type="radio"/> Task A1		
<input type="radio"/> Task A2		
▼ <input type="radio"/> Subprocess B		
<input type="radio"/> Task B1		
<input type="radio"/> Task B2		



To group the tasks in your process model:

- 1 In the process model, add a subprocess by using `addSubprocess` on your process model object.

```
spA = pm.addSubprocess("Subprocess A");
```

- 2 Add your tasks directly to the subprocess by using `addTask`.

```
tA1 = spA.addTask("Task A1");
tA2 = spA.addTask("Task A2");
```

Note You do not need to add the task to both the subprocess and process model.

- 3 Specify the relationship between the tasks and subprocesses in your process.

You can use the `dependsOn` and `runsAfter` functions to define the relationships.

For example, the following process model defines a process in which Task 1 runs, then Subprocess A, and then Subprocess B.

```
function processmodel(pm)
    % Defines the project's processmodel
    arguments
```

```

    pm padv.ProcessModel
end

t1 = pm.addTask("Task 1");

spA = pm.addSubprocess("Subprocess A");
    tA1 = spA.addTask("Task A1");
    tA2 = spA.addTask("Task A2");
spB = pm.addSubprocess("Subprocess B");
    tB1 = spB.addTask("Task B1");
    tB2 = spB.addTask("Task B2");

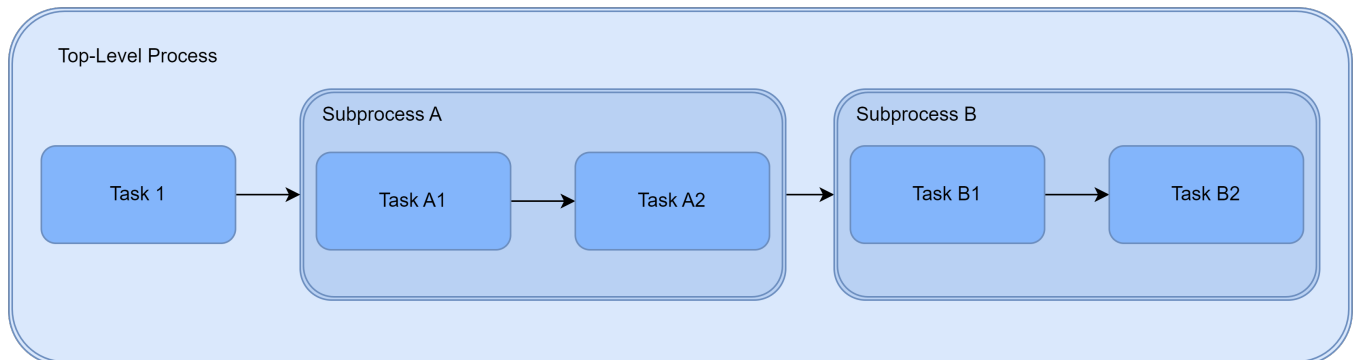
% Relationships
spA.dependsOn(t1);
    tA2.dependsOn(tA1);
spB.dependsOn(spA);
    tB2.dependsOn(tB1);

end

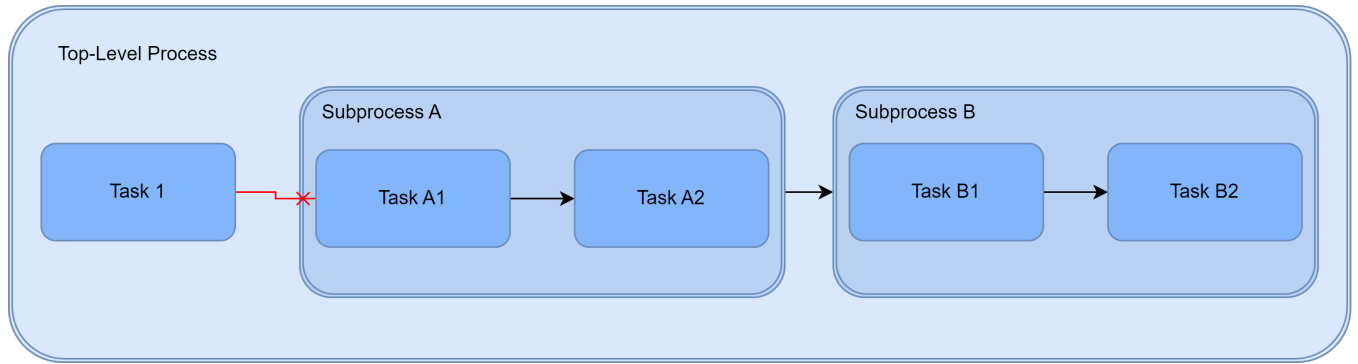
```

The build system executes each of the tasks inside a subprocess before existing the subprocess.

The following diagram shows a graphical representation of the relationships defined by that process model.



Note Relationships cannot cross any subprocess boundaries. For example, in this process model, you cannot directly specify that Task A1 depends on Task 1 because that relationship would enter into Subprocess A, crossing the subprocess boundary.



padv.Task Class

Namespace: padv

Single step in process

Description

A `padv.Task` object represents a single step in a `padv.ProcessModel` process. For example, a `padv.Task` object could represent a step like checking modeling standards, running tests, generating code, or performing a custom action. `padv.Task` objects can accept project artifacts as inputs, perform actions, generate assessments, and return project artifacts as outputs. You can add a task to your process model by using the function `addTask`. Then, in your process model, use the object functions `addInputQueries`, `dependsOn`, and `runsAfter` to specify the inputs, dependencies, and desired execution order for a task. You can execute tasks as part of a pipeline. Use the `runprocess` function to generate and run a pipeline of tasks.

Creation

Syntax

```
taskObject = padv.Task(Name)
taskObject = padv.Task( ____, Name=Value)
```

Description

`taskObject = padv.Task(Name)` represents a task, named `Name`, in a `padv.ProcessModel` process. Each task object in a process must have a unique `Name`.

`taskObject = padv.Task(____, Name=Value)` sets properties using one or more name-value arguments. For example, `padv.Task("myTask", IterationQuery=padv.builtin.query.FindModels)` creates a task object named `myTask` that runs once for each model.

Specify optional pairs of arguments as `Name1=Value1, . . . , NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Properties

Public Properties

Name — Unique identifier for task in process

string

Unique identifier for task in process, returned as a string. When you specify the `Name`, you specify the `Name` property of the task object.

Each task in the process model must have a unique `Name`. After you specify a `Name` for a `padv.Task` object, you cannot change the `Name`.

Example: `padv.Task("myTask")` creates a task with the Name `myTask`

Data Types: `string`

Title — Human readable name that appears in Process Advisor app

`string`

Human readable name that appears in the **Tasks** column of the Process Advisor app, returned as a `string`. By default, the Process Advisor app uses the `Name` property of the task as the `Title`.

Example: `padv.Task("myTask", Title = "My Task")`

Data Types: `string`

DescriptionText — Task description

`string`

Task description, returned as a `string`.

Example: `padv.Task("myTask", DescriptionText = "This is my task.")`

Data Types: `string`

DescriptionCSH — Path to task documentation

`string`

Path to task documentation, returned as a `string`.

Example: `padv.Task("myTask", DescriptionCSH = fullfile(pwd, "taskHelpFiles", "myTaskDocumentation.pdf"))`

Data Types: `string`

Action — Function that task runs

`function handle`

Function that the task runs, returned as the function handle. When you run the task, the task runs the function specified by the function handle.

For example, if you want the task to run the function `myFunction`, specify `Action` as `@myFunction`.

Example: `padv.Task("myTask", Action = @myFunction)`

Data Types: `function_handle`

RequiredIterationArtifactType — Artifact type that task can run on

`string`

Artifact type that the task can run on, returned by a `string`. The required iteration artifact type must be the artifact type supported by the `IterationQuery` property of the task.

For a list of valid artifact types, see the chapter "Artifact Types" in this PDF.

Example: `padv.Task("myTask", RequiredIterationArtifactType = "sl_model_file")`

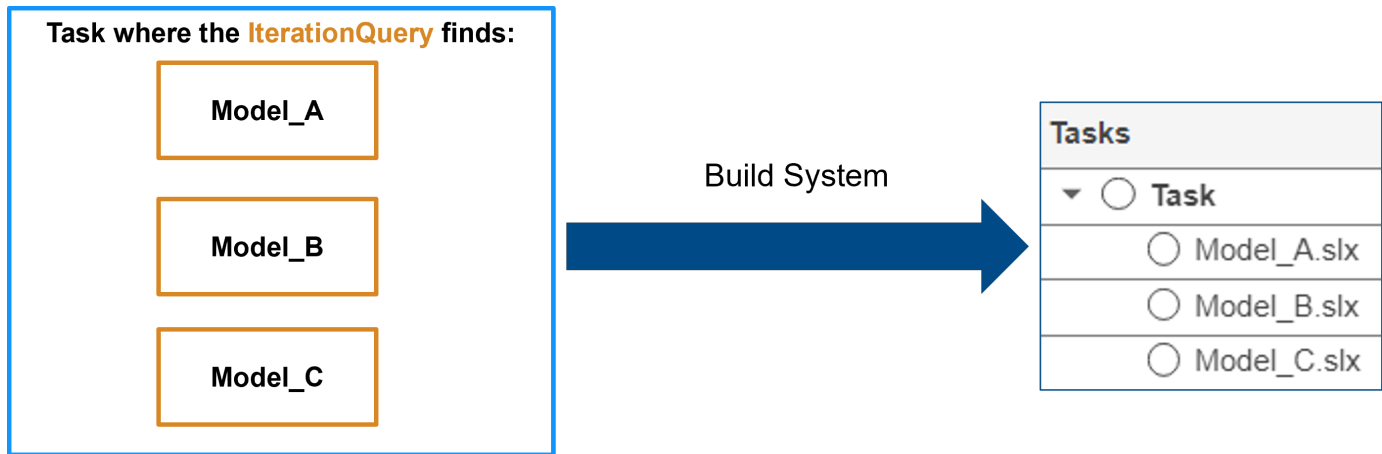
Data Types: `string`

IterationQuery — Artifacts that task iterates over

`padv.Query object` | name of `padv.Query` object

Artifacts that task iterates over, returned as a `padv.Query` object or the name of a `padv.Query` object. By default, task objects run one time and are associated with the project. When you specify `IterationQuery`, the task runs one time *for each* artifact returned by the `padv.Query`. In the Process Advisor app, the artifacts returned by `IterationQuery` appear under task title.

For example, if the `IterationQuery` for a task finds three models, `Model_A`, `Model_B`, and `Model_C`, the build system creates three task iterations under the title of the task in the **Tasks** column.



Each of the artifacts under the task title represents a *task iteration*.

For an example of the effect of different `IterationQuery` values:

- If you have a task where the `IterationQuery` uses `padv.builtin.query.FindModels` to find each of the models in the project, the build system creates a task iteration for each model.
- If you have a task where the `IterationQuery` uses `padv.builtin.query.FindProjectFile` to find the project file, the build system creates a task iteration for the project file.
- If you have a task where the `IterationQuery` uses `padv.builtin.query.FindTopModels` to find top models in the project, the build system creates a task iteration for each top model.

	Tasks
Task iteration for each model	<ul style="list-style-type: none"> Task with <code>IterationQuery = padv.builtin.query.FindModels</code> <ul style="list-style-type: none"> AHRS_Voter.slx Actuator_Control.slx Flight_Control.slx InnerLoop_Control.slx OuterLoop_Control.slx
Task iteration for the project	<ul style="list-style-type: none"> Task with <code>IterationQuery = padv.builtin.query.FindProjectFile</code> <ul style="list-style-type: none"> ProcessAdvisorExample.prj
Task iteration for the top model	<ul style="list-style-type: none"> Task with <code>IterationQuery = padv.builtin.query.FindTopModels</code> <ul style="list-style-type: none"> Flight_Control.slx

Example: `padv.Task("myTask", IterationQuery = padv.builtin.query.FindModels)`

Data Types: `string`

InputDependencyQuery — Artifact dependencies for task inputs

`padv.Query` object | name of `padv.Query` object

Artifact dependencies for task inputs, returned as a `padv.Query` object or the name of a `padv.Query` object.

Artifact dependencies for task inputs, specified as a `padv.Query` object or the name of a `padv.Query` object.

The build system runs the query specified by `InputDependencyQuery` to find the dependencies for the task inputs, since those dependencies can affect whether task results are up-to-date. Typically, you specify `InputDependencyQuery` as `padv.builtin.query.GetDependentArtifacts` to get the dependent artifacts for each task input. For example, if you specify a model as an input to a task and you specify `InputDependencyQuery` as `padv.builtin.query.GetDependentArtifacts`, the build system can find artifacts, such as data dictionaries, that the model uses.

Example: `InputDependencyQuery = padv.builtin.query.GetDependentArtifacts`

IncludeMatlabWarningsInResults — Automatically include number of MATLAB warning messages in padv.TaskResult

`false` or `0` (default) | `true` or `1`

Automatically include the number of MATLAB warning messages in the `padv.TaskResult`, returned as a numeric or logical `0` (`false`) or `1` (`true`).

Example: `true`

Data Types: `logical`

Licenses — List of licenses that task requires

string array

List of licenses that the task requires, returned as a string array.

Example: `padv.Task("myTask",Licenses = ["matlab_report_gen" "simulink_report_gen"])`

Data Types: `string`

Products — List of products that must be installed to run task

string array

List of products that must be installed to run the task, returned as a string array.

Data Types: `string`

AllLicenseRequired — Setting to require all licenses for task

`true` or `1` (default) | `false` or `0`

Setting to require all licenses for task, returned as a numeric or logical `1` (`true`) or `0` (`false`). By default, all licenses in the `Licenses` property of the task are required for the task to run. Specify `0` (`false`) if the task can run without all licenses listed in the `Licenses` property.

Example: `padv.Task("myTask",AllLicenseRequired = false)`

Data Types: `logical`

LaunchToolAction — Function that launches a tool

function handle

Function that launches a tool, returned as the function handle.

When the property `LaunchToolAction` is specified, you can point to the task in the Process Advisor app and click the ellipsis (...) and then **Open Tool Name** to open the tool associated with the task.

For tasks that are not built-in tasks, the task options show the ellipsis (...) and then **Launch Tool**.

Example: `padv.Task("myTask",LaunchToolAction = @openTool)`

Data Types: `function_handle`

LaunchToolText — Description of action that LaunchToolAction property performs

"Launch Tool" (default) | string scalar

Description of the action that the `LaunchToolAction` property performs, returned as a string scalar.

Example: `padv.Task("myTask",LaunchToolAction = @openTool, LaunchToolText = "Open tool.")`

Data Types: `string`

Enabled — Controls if the padv.Task is enabled in the process model

true or 1 (default) | false or 0

Controls if the `padv.Task` is enabled in the process model, returned as a numeric or logical 1 (true) or 0 (false).

Example: `padv.Task("myTask",Enabled = false)`

Data Types: `logical`

AlwaysRun — Always force task to run, even if the task results are already up to date

false or 0 (default) | true or 1

Always force task to run, even if the task results are already up to date, returned as a numeric or logical 0 (false) or 1 (true).

Example: `padv.Task("myTask",AlwaysRun = true)`

Data Types: `logical`

TrackOutputs — Track changes to output files

true or 1 (default) | false or 0

Track changes to output files, specified as a numeric or logical 1 (true) or 0 (false).

By default, the build system tracks changes to outputs files from tasks unless the files are outside the project. If you make a change to an output file, the task status and results are marked as outdated. If you specify `TrackOutputs` as false, any changes you make to the task output files do not make the task status and results outdated.

For more information, see "Turn Off Change Tracking for Task Outputs" in the User's Guide PDF.

Example: `false`

Data Types: `logical`

InputQueries — Inputs to task

`padv.Query` object | name of `padv.Query` object | array of `padv.Query` objects

Inputs to the task, returned as:

- a `padv.Query` object
- the name of `padv.Query` object
- an array of `padv.Query` objects
- an array of names of `padv.Query` objects

By default, the task does not specify any artifacts as inputs. When you specify `InputQueries`, the task tasks the artifacts returned by the specified query or queries as an input.

Suppose a task runs once for each model in the project and you want to provide the models as inputs to the task. If you specify `InputQueries` as the built-in query `padv.builtin.query.GetIterationArtifact`, the query returns each artifact that the tasks iterates over, which in this example is each of the models in the project.

```
Example: padv.Task("myTask", InputQueries =
padv.builtin.query.GetIterationArtifact)
```

OutputDirectory — Location for standard outputs that the task produces

"" (default) | string array

Location for standard outputs that the task produces, specified as a string.

Built-in tasks automatically specify `OutputDirectory`. If you do not specify `OutputDirectory` for a custom task, the build system stores task outputs in the `DefaultOutputDirectory` specified by `padv.ProcessModel`.

Data Types: `string`

CacheDirectory — Location for any additional cache files that the task generates

string array

Location for any additional cache files that the task generates, specified as a string. The cache directory can contain temporary files that do not need to be either saved in the task results or archived by a CI system.

Data Types: `string`

CISupportOutputsForTask — List of CI aware result file types generated for task

"JUnit" (default) | string array

List of CI aware result file types to be generated for task, specified as a string array.

Data Types: `string`

CISupportOutputsByTask — List of CI aware result file types generated by task

empty string (default) | string array

List of CI aware result file types generated by task, specified as a string array.

Data Types: `string`

Methods

Object Functions

Object Function	Description
addInputQueries	Add the input artifacts returned by <code>inputQueries</code> as inputs to the task represented by <code>taskObj</code> . <code>addInputQueries(taskObj, inputQueries)</code>
dependsOn	Create a dependency between a task, <code>taskObj</code> , and dependencies, <code>dependencies</code> . <code>dependsOn(taskObj, dependencies)</code>
run	Run task represented by <code>taskObj</code> . <code>taskResult = run(taskObj)</code> If the task requires inputs, specify the inputs using <code>inputArtifacts</code> . <code>taskResult = run(taskObj, inputArtifacts)</code>
runsAfter	Specify the preferred execution order for tasks by specifying the tasks, <code>predecessors</code> , that a task, <code>taskObj</code> , should run after. <code>runsAfter(taskObj, predecessors)</code>

See the next sections for more information on these object functions.

Examples


Create Task Objects and Add Tasks to Process Model

You can use `padv.Task` to create task objects and then use the `addTask` function to add the task objects to the `padv.ProcessModel` object.

Open the Process Advisor example project.

```
processAdvisorExampleStart
```

The model `AHRS_Voter` opens with the Process Advisor pane to the left of the Simulink canvas.

In the Process Advisor pane, click the **Edit process model**  button to open the `processmodel.m` file for the project.

Replace the contents of the `processmodel.m` file with this code:

```
function processmodel(pm)
    arguments
        pm padv.ProcessModel
    end
```



```
taskA = padv.Task("taskA");  
taskB = padv.Task("taskB");  
  
runsAfter(taskB, taskA);  
  
addTask(pm, taskA);  
addTask(pm, taskB);
```

```
end
```

This code uses `padv.Task` to create two task objects: `taskA` and `taskB`.

The object function `runsAfter` specifies that `taskB` should run after `taskA`.

The function `addTask` adds the task objects to the `padv.ProcessModel` object.

addInputQueries

Namespace: padv

Add input artifacts as inputs to task

Syntax

```
addInputQueries(taskObj, inputQueries)
```

Description

`addInputQueries(taskObj, inputQueries)` adds the input artifacts returned by `inputQueries` as inputs to the task represented by `taskObj`.

If the task already has input queries specified, `addInputQueries` adds `inputQueries` to the list of input queries in the `InputQueries` property.

Examples

Add Inputs to Task

Use `addInputQueries` to specify the models in the project as inputs to a task.

Create a new `padv.Task` object `myTaskObj` that represents a task named `runForEachModel`.

```
myTaskObj = padv.Task("runForEachModel");
```

By default, the task does not have any inputs.

Use the function `addInputQueries` to add the built-in query `padv.builtin.query.FindModels` as the input query for the task.

```
addInputQueries(myTaskObj, padv.builtin.query.FindModels);
```

When you run the task defined by `myTaskObj`, the query `padv.builtin.query.FindModels` finds each model in the project and provides the models as the input artifacts for the task.

Input Arguments

taskObj — Task object that represents task

`padv.Task` object

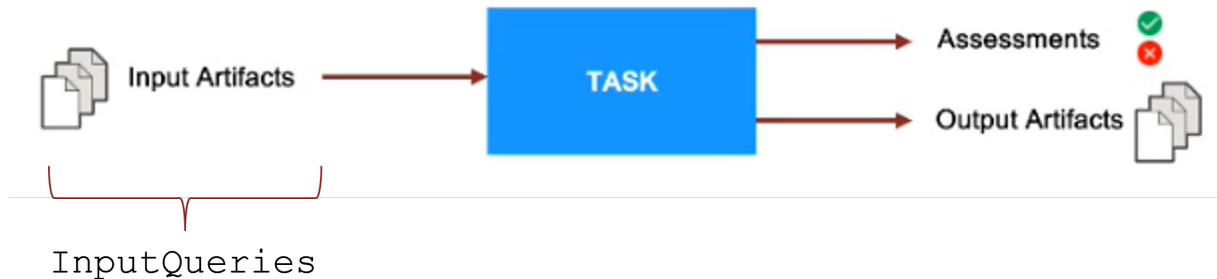
Task object that represents a task, specified as a `padv.Task` object.

Example: `myTaskObj = padv.Task("myTask");`

inputQueries — Queries that get input artifacts for task

`padv.Query` object | array of `padv.Query` objects

A query or queries that get the input artifacts for a task, specified as a `padv.Query` object or an array of `padv.Query` objects. Each artifact that the query or queries return becomes an input to the task.



For example, if you specify the `InputQueries` property for a task as the query `padv.builtin.query.FindModels`, the query returns each model and the models become input artifacts for the task.

Note You can only specify the following queries for the `inputQueries` argument:

- `padv.builtin.query.FindArtifacts`
- `padv.builtin.query.FindFileWithAddress`
- `padv.builtin.query.FindModels`
- `padv.builtin.query.FindProjectFile`
- `padv.builtin.query.FindRequirements`
- `padv.builtin.query.FindRequirementsForModel`
- `padv.builtin.query.FindTestCasesForModel`
- `padv.builtin.query.FindTopModels`
- `padv.builtin.query.GetDependentArtifacts`
- `padv.builtin.query.GetIterationArtifact`
- `padv.builtin.query.GetOutputsOfDependentTask`

You cannot specify the following queries for `inputQueries`:

- `padv.builtin.query.FindFilesWithLabel`
- `padv.builtin.query.FindModelsWithLabel`
- `padv.builtin.query.FindModelsWithTestCases`
- `padv.builtin.query.FindRefModels`

Example: `addInputQueries(myTaskObj, padv.builtin.query.FindModels)`

Example: `addInputQueries(myTaskObj, [padv.builtin.query.GetIterationArtifact, padv.builtin.query.GetDependentArtifacts])`

dependsOn

Namespace: padv

Create dependency between tasks

Syntax

```
dependsOn(taskObj,dependencies)  
dependsOn( ____,Name=Value)
```

Description

`dependsOn(taskObj,dependencies)` creates a dependency between `taskObj` and `dependencies`. `taskObj` runs only after the tasks specified by `dependencies` run and return a task status.

`dependsOn(____,Name=Value)` specifies how the build system handles dependencies using one or more `Name=Value` arguments.

Examples

Create Dependency Between Two Tasks

Use the `dependsOn` function to create a dependency between two tasks in a process model.

Open the Process Advisor example project.

```
processAdvisorExampleStart
```

Open the `processmodel.m` file. The file is at the root of the project.

Replace the contents of the `processmodel.m` file with the following code:

```
function processmodel(pm)  
    arguments  
        pm padv.ProcessModel  
    end  
  
    TaskA = padv.Task("TaskA");  
    TaskB = padv.Task("TaskB");  
  
    dependsOn(TaskB,TaskA);  
  
    addTask(pm,TaskA);  
    addTask(pm,TaskB);  
  
end
```

This code uses `padv.Task` to create two task objects: `TaskA` and `TaskB`.

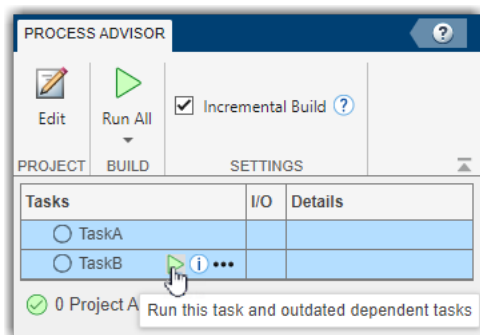
The object function `dependsOn` specifies that `TaskB` depends on `TaskA`.

The function `addTask` adds the task objects to the `padv.ProcessModel` object.

Open the Process Advisor app. In the MATLAB Command Window, enter:

```
processAdvisorWindow
```

In the **Tasks** column, point to the run button for **TaskB**. The Process Advisor app automatically highlights both tasks since **TaskA** is a dependent task. If you click the run button for **TaskB**, **TaskA** will run before **TaskB**.



Input Arguments

taskObj — Task object that represents task

`padv.Task` object

Task object that represents a task, specified as a `padv.Task` object.

Example: `myTaskObj = padv.Task("myTask");`

dependencies — Tasks that need to run before `taskObj` runs

string | character vector | `padv.Task` object

Tasks that need to run before `taskObj` runs, specified as either:

- The name of a task, specified as a string or character vector.
- A `padv.Task` object.

Example: `dependsOn(TaskB,"TaskA")`

Example: `dependsOn(TaskB,TaskA)`

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `dependsOn(TaskB,TaskA,WhenStatus=["Pass","Fail"])`

IterationArtifactMatching — Setting that controls which dependent task iterations run

`true` or `1` (default) | `false` or `0`

Setting that controls which dependent task iterations run, specified as a numeric or logical 1 (true) or 0 (false):

- `true` — When the build system runs the dependencies of a task, the build system runs only the task iterations that the tasks have in common.
- `false` — When the build system runs the dependencies of a task, the build system runs all task iterations. This behavior is useful when you have a task that creates new project artifacts and a task that runs on each artifact in the project. The second task depends on all project artifacts generated by the first task.


For example, suppose you have two tasks: TaskA and TaskB:

- TaskA runs on ModelA and ModelB.
- TaskB runs only on ModelB and depends on TaskA.

If you run TaskB and:


- `IterationArtifactMatching` is true, TaskA runs only on ModelB.

Tasks	Out	Details
▼ ○ TaskA		
○ ModelB.slx		
○ ModelA.slx		
▼ ○ TaskB		
○ ModelB.slx		

 Run this task and outdated dependent tasks

- `IterationArtifactMatching` is false, TaskA runs on both ModelA and ModelB.

Tasks	Out	Details
▼ ○ TaskA		
○ ModelB.slx		
○ ModelA.slx		
▼ ○ TaskB		
○ ModelB.slx		

 Run this task and outdated dependent tasks

Example: `dependsOn(TaskB,TaskA,IterationArtifactMatching=false)`

Data Types: `logical`

WhenStatus — Setting that controls when dependencies run

"Pass" (default) | ["Pass", "Fail"] | ["Pass", "Fail", "Error"]

Setting that controls when dependencies run, specified as either:

- "Pass" — Only run the task if the dependencies pass. For example, if TaskB depends on TaskA, TaskA needs to pass before TaskB runs. If TaskA fails or errors, TaskB does not run.
- ["Pass", "Fail"] — Only run the task if the dependencies either pass or fail. For example, if TaskB depends on TaskA, TaskA needs to either pass or fail before TaskB runs. If TaskA errors, TaskB does not run.
- ["Pass", "Fail", "Error"] — The task runs, even if the dependencies fail or error. For example, if TaskB depends on TaskA, TaskA can pass, fail, or error and TaskB still runs.

Example: `dependsOn(TaskB, TaskA, WhenStatus=["Pass", "Fail"])`

Data Types: string

run

Namespace: padv

Run task

Syntax

```
taskResult = run(taskObj)
taskResult = run(taskObj, inputArtifacts)
```

Description

`taskResult = run(taskObj)` runs the task represented by `taskObj` and returns the result from the task.

How a task runs depends on how the you define the task. You can define tasks using a function or a class:

- Function-based tasks — Runs the function specified by the `Action` property of the task.
- Class-based task — Runs the `run` function implemented inside the class definition.

By default, when you create a `padv.Task` object, the task is a function-based task, even if you do not specify an `Action` property for the task.

`taskResult = run(taskObj, inputArtifacts)` uses the artifacts specified by `inputArtifacts` as inputs to the task. The `InputQueries` property of the task specifies the query that provides the `inputArtifacts` for the task.

Examples

Run Task

Create a new `padv.Task` object and run the task.

Create a new `padv.Task` object `myTaskObj` that represents a task named `myTask`.

```
myTaskObj = padv.Task("myTask");
```

Use the `run` object function to run the task. Save the results to the variable `taskResults`.

```
taskResults = run(myTaskObj)
```

```
taskResults =
```

```
    TaskResult with properties:
```

```
        Status: Pass
    OutputArtifacts: [0x0 padv.Artifact]
        Details: [1x1 struct]
    ResultValues: [1x1 struct]
```


In this example, there is no `Action` associated with the task and the task returns a `padv.TaskResult` with a `Status` of `Pass`.

Input Arguments

taskObj — Task object that represents task

`padv.Task` object

Task object that represents a task, specified as a `padv.Task` object.

Example: `myTaskObj = padv.Task("myTask");`

inputArtifacts — Artifacts that are inputs to task

cell array of `padv.Artifact` objects

Artifacts that are inputs to the task, specified as a cell array of `padv.Artifact` objects.

If you specified the `InputQueries` property for a task, the `InputQueries` automatically passes a cell array of `padv.Artifact` objects to `inputArtifacts` when you run the task.

Output Arguments

taskResult — Result from task

`TaskResult` object

Result from the task, returned as a `TaskResult` object.

runsAfter

Namespace: padv

Specify preferred execution order for tasks

Syntax

```
runsAfter(taskObj, predecessors)  
runsAfter( ____, Name=Value)
```

Description

`runsAfter(taskObj, predecessors)` specifies a preferred execution order for tasks. If possible, the build system runs the predecessor tasks, specified by `predecessors`, before the task represented by `taskObj`.

`runsAfter(____, Name=Value)` specifies how the build system handles the preferred execution order using one or more `Name=Value` arguments.

Examples

Specify Preferred Execution Order for Two Tasks

Use the `runsAfter` function to specify that one task should run after another task.

Open the Process Advisor example project.

```
processAdvisorExampleStart
```

Open the `processmodel.m` file. The file is at the root of the project.

Replace the contents of the `processmodel.m` file with the following code:

```
function processmodel(pm)  
    arguments  
        pm padv.ProcessModel  
    end  
  
    FirstTask = padv.Task("FirstTask");  
    SecondTask = padv.Task("SecondTask");  
  
    runsAfter(SecondTask, FirstTask);  
  
    addTask(pm, FirstTask);  
    addTask(pm, SecondTask);  
  
end
```

This code uses `padv.Task` to create two task objects: `FirstTask` and `SecondTask`.

The object function `runsAfter` specifies that `SecondTask` should run after `FirstTask`.

The function `addTask` adds the task objects to the `padv.ProcessModel` object.

Open the Process Advisor app. In the MATLAB Command Window, enter:

```
processAdvisorWindow
```

In the toolbar, click the **Run All** button. You can see that **SecondTask** runs after **FirstTask**.

Input Arguments

taskObj — Task object that represents task

`padv.Task` object

Task object that represents a task, specified as a `padv.Task` object.

Example: `myTaskObj = padv.Task("myTask");`

predecessors — Tasks that should run before `taskObj` runs

string | character vector | `padv.Task` object

Tasks that should run before `taskObj` runs, specified as either:

- The name of a task, specified as a string or character vector.
- A `padv.Task` object.

Example: `runsAfter(SecondTask,"FirstTask")`

Example: `runsAfter(SecondTask,FirstTask)`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `runsAfter(SecondTask,FirstTask,StrictOrdering=true)`

IterationArtifactMatching — Setting that controls which predecessor task iterations run

`true` or `1` (default) | `false` or `0`

Setting that controls which predecessor task iterations run, specified as a numeric or logical `1` (`true`) or `0` (`false`):

- `true` — When the build system runs the predecessors of a task, the build system runs only the task iterations that the tasks have in common.
- `false` — When the build system runs the predecessor of a task, the build system runs all task iterations. This behavior is useful when you have a task that creates new project artifacts and a task that runs on each artifact in the project. The second task should run after all project artifacts are generated by the first task.

For example, suppose you have two tasks: `FirstTask` and `SecondTask`:

- `FirstTask` runs on `ModelA` and `ModelB`.
- `SecondTask` runs only on `ModelB` and should run after on `FirstTask`.

If you run `SecondTask` and:

- `IterationArtifactMatching` is `true`, `FirstTask` runs only on `ModelB`.
- `IterationArtifactMatching` is `false`, `FirstTask` runs on both `ModelA` and `ModelB`.

Example: `runsAfter(SecondTask,FirstTask,IterationArtifactMatching=false)`

Data Types: `logical`

StrictOrdering – Setting that controls whether build system ignores circular relationships between tasks

`false` or `0` (default) | `true` or `1`

Setting that controls whether the build system ignores circular relationships between tasks, specified as a numeric or logical `0` (`false`) or `1` (`true`). By default, if you specify a circular relationship between tasks, the build system ignores the relationship. For example, if you specify both `runsAfter(SecondTask,FirstTask)` and `runsAfter(FirstTask,SecondTask)`, the build system ignores the `runsAfter` relationship.

If you specify `StrictOrdering` as `true`, the build system generates an error when you try to run tasks that have a circular relationship.

Example: `runsAfter(SecondTask,FirstTask,StrictOrdering=true)`

Data Types: `string`

padv.TaskResult

Create and access results from task

Description

A `padv.TaskResult` object represents the results from a task. The `run` function of a `padv.Task` creates a `padv.TaskResult` object that you can use to access the results from the task. When you create a custom task, you can specify the results from your custom task. You can also use the function `getProcessTaskResults` to view a list of the last task results for a project. The Process Advisor app uses task results to determine the task statuses, output artifacts, and detailed task results that appear in the **Tasks**, **Out**, and **Details** columns of the app.

Creation

Syntax

```
resultObj = padv.TaskResult()
```

Description

`resultObj = padv.TaskResult()` creates a result object `resultObj` that represents the results from a task.

Properties

Status — Task result status

Pass (default) | Fail | Error

Task result status, returned as:

- **Pass** — A passing task status. The task completed successfully without any issues.
- **Fail** — A failing task status. The task completed, but the results were not successful.
- **Error** — An error task status. The task generated an error and did not complete.

The `Status` property determines the task status shown in the **Tasks** column in the Process Advisor app.

For custom tasks, you can specify the task result status as either:

- `padv.TaskStatus.Pass` — Sets the `Status` property to `Pass`.
- `padv.TaskStatus.Fail` — Sets the `Status` property to `Fail`.
- `padv.TaskStatus.Error` — Sets the `Status` property to `Error`.

For example, `taskResult.Status = padv.TaskStatus.Fail` sets the `Status` property of the task result object to `Fail` to represent a failing task status.

Example: `Fail`

OutputArtifacts — Artifacts created by task`padv.Artifact` object | array of `padv.Artifact` objects

Artifacts created by the task, returned as a `padv.Artifact` object or array of `padv.Artifact` objects.

The `OutputArtifacts` property determines the output artifacts shown in the **Out** column in the Process Advisor app.

The build system only manages output artifacts specified by the task result. For custom tasks, use the `OutputPaths` argument to define the output artifacts for the task result.

Details — Temporary storage for task-specific data`struct`

Temporary storage for task-specific data, returned as a `struct`. The build system uses `Details` to store task-specific data that other build steps can use.

Note that `Details` are temporary. The build system does not save `Details` with the task results after the build finishes.

Data Types: `struct`

ResultValues — Number of passing, warning, and failing conditions`struct` with `Pass: 0, Warn: 0, Fail: 0` (default) | `struct` with fields `Pass, Warn, Fail`

Number of passing, warning, and failing conditions, returned as a `struct` with fields:

- `Pass` — Number of passing conditions returned by task
- `Warn` — Number of warning conditions returned by task
- `Fail` — Number of failing conditions returned by task

The `ResultValues` property determines the detailed results shown in the **Details** column in the Process Advisor app.

For example, the task `padv.builtin.task.RunModelStandards` runs several Model Advisor checks and returns the number of passing, warning, and failing checks. If you run the task and one check passes, two checks generate a warning, and three checks fail, `ResultValue` returns:

```
ans =
```

```
  struct with fields:
```

```
    Pass: 1  
    Warn: 2  
    Fail: 3
```

Data Types: `struct`

OutputPaths — Define OutputArtifacts for task result`string`

This property is write-only.

`OutputArtifacts` for task result, specified as a string or string array.

The build system adds each path specified by `OutputArtifacts` to the `OutputArtifacts` argument as a `padv.Artifact` object with type `padv_output_file`.

Example: `taskResultObj.OutputPaths = string(fullfile(pwd, filename))`

Example: `taskResultObj.OutputPaths = [string(fullfile(pwd, filename1)), string(fullfile(pwd, filename2))]`

Data Types: `char` | `string`

Object Functions

- `applyStatus`

Examples


Create Task Result for Custom Task

Create a `padv.TaskResult` object for a custom task that has a failing task status, outputs a single `.json` file, and 1 passing condition, 2 warning conditions, and 3 failing conditions.

Open the Process Advisor example project.

```
processAdvisorExampleStart
```

The model `AHRS_Voter` opens with the Process Advisor pane to the left of the Simulink canvas.

In the Process Advisor pane, click the **Edit process model**  button to open the `processmodel.m` file for the project.

Replace the contents of the `processmodel.m` file with this example process model code:

```
function processmodel(pm)
    % Defines the project's processmodel

    arguments
        pm padv.ProcessModel
    end

    addTask(pm, "ExampleTask", Action=@ExampleAction);

end

function taskResult = ExampleAction(~)

    % Create a task result object that stores the results
    taskResult = padv.TaskResult();

    % Specify the task status shown in the Tasks column
    taskResult.Status = padv.TaskStatus.Fail;

    % Specify the output files shown in the Out column
    cp = currentProject;
    rf = cp.RootFolder;
    outputFile = fullfile(rf, "tools", "sampleChecks.json");
```

```

taskResult.OutputPaths = string(outputFile);

% Specify the values shown in the Details column
taskResult.ResultValues.Pass = 1;
taskResult.ResultValues.Warn = 2;
taskResult.ResultValues.Fail = 3;

```

end

Save the `processmodel.m` file.

Go back to the Process Advisor app and click **Refresh Tasks** to update the list of tasks shown in the app.

In the top-left corner of the Process Advisor pane, switch the filter from **Model** to **Project**.

In the top-right corner of the Process Advisor pane, click **Run All**.

- The **Tasks** column shows a failing task status to the left of **ExampleTask**. This code from the example process model specifies the task status shown in the **Tasks** column:

```
taskResult.Status = padv.TaskStatus.Fail;
```

- The **Out** column shows an output artifact associated with the task. This code from the example process model specifies the output artifact shown in the **Out** column:

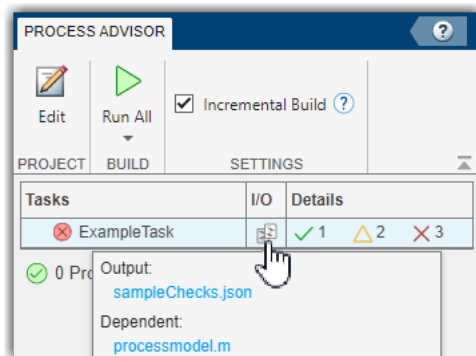
```
taskResult.OutputPaths = string(fullfile(pwd,outputFile));
```

- The **Details** column shows 1 passing condition, 2 warning conditions, and 3 failing conditions. This code from the example process model specifies the detailed task results shown in the **Details** column:

```

taskResult.ResultValues.Pass = 1;
taskResult.ResultValues.Warn = 2;
taskResult.ResultValues.Fail = 3;

```



applyStatus

Namespace: padv

Apply new task status if priority is higher

Syntax

```
applyStatus(resultObj, taskStatus)
```

Description

`applyStatus(resultObj, taskStatus)` applies a new task status `taskStatus` to the task result object `resultObj` if the priority level of `taskStatus` is higher than the current `Status` property of the task result object.

The priority levels from lowest to highest are:

- `padv.TaskStatus.Pass`
- `padv.TaskStatus.Fail`
- `padv.TaskStatus.Error`

Note The function `applyStatus` can only change the `Status` to a higher priority status. For example, if you apply a failing status and then apply a passing status, the status remains a failing status because the priority of `padv.TaskStatus.Fail` is higher than the priority of `padv.TaskStatus.Pass`.

```
taskResult = padv.TaskResult(); % By default, Status is Pass.
applyStatus(taskResult, padv.TaskStatus.Fail); % Status changes to Fail.
applyStatus(taskResult, padv.TaskStatus.Pass); % Status remains Fail.
taskResult
```

```
taskResult =
```

```
    TaskResult with properties:
```

```
        Status: Fail
    OutputArtifacts: [0x0 padv.Artifact]
        Details: [1x1 struct]
    ResultValues: [1x1 struct]
```

To set the `Status` property of a task result object to a specific value, manually set the property to either `padv.TaskStatus.Pass`, `padv.TaskStatus.Fail`, or `padv.TaskStatus.Error`. For example, to set the `Status` of a task result object `taskResult` to `Pass`, use `taskResult.Status = padv.TaskStatus.Pass`.

Examples

Apply Status to Task Result

Use `applyStatus` to update the `Status` property of a task result object. If the status is a higher priority status, `applyStatus` updates the `Status` property of the task result object.

Create a task result object. By default, the `Status` property of the task result object is specified as `Pass`.

```
taskResult = padv.TaskResult();
```

Suppose the task needs to generate an error. Use `applyStatus` to apply an error task status, specified by `padv.TaskStatus.Error`.

```
applyStatus(taskResult, padv.TaskStatus.Error);
```

`padv.TaskStatus.Error` has a higher priority than a passing task status, so `applyStatus` updates the `Status` property of the task result object.

Apply a passing task status to the task result object. A passing task status is specified by `padv.TaskStatus.Pass`.

```
applyStatus(taskResult, padv.TaskStatus.Pass);
```

`padv.TaskStatus.Pass` does not have a higher priority than an error task status, so `applyStatus` does not change the `Status` of the task result object.

Inspect the properties of the task result object.

```
taskResult
```

Suppose you want to reset the status of the task result object to a passing task status. Manually specify the `Status` property as `padv.TaskStatus.Pass`.

```
taskResult.Status = padv.TaskStatus.Pass
```

```
taskResult =
```

```
    TaskResult with properties:
```

```
        Status: Pass
    OutputArtifacts: [0×0 padv.Artifact]
        Details: [1×1 struct]
    ResultValues: [1×1 struct]
```

The task result object now has a passing task status.

Input Arguments

resultObj — Task result object

`padv.TaskResult` object

Task result object, specified as a `padv.TaskResult` object.

taskStatus — Task status

`padv.TaskStatus.Pass` | `padv.TaskStatus.Fail` | `padv.TaskStatus.Error`

Task status, specified as `padv.TaskStatus.Pass`, `padv.TaskStatus.Fail`, or `padv.TaskStatus.Error`.

Example: `padv.TaskStatus.Fail`

Build System API

The support package provides a build system that you can use to orchestrate and automate the steps in your model-based design (MBD) pipeline. The build system is software that can orchestrate tasks, efficiently execute tasks in the pipeline, and perform other actions related to the pipeline. You can call the build system either through the Process Advisor app or by using the `runprocess` function. When you call the build system, the build system loads the process model, analyzes the project, and orchestrates the create of a pipeline of tasks.

For examples of how to use the build system, see the "Control Builds" and "Integrate into CI" chapters in the user's guide.

Classes

Class	Description
<code>padv.BuildResult</code>	Result from build system build
<code>padv.Preferences</code>	(To be removed) Set <code>runprocess</code> function settings
<code>padv.ProjectSettings</code>	Build system settings for project
<code>padv.UserSettings</code>	Build system settings for user

Functions

Run Tasks

Function	Description
<code>runprocess</code>	Run task iterations defined by the process model

Get Task Iterations and Tasks Results

Function	Description
<code>createProcessTaskID</code>	Generate an ID for a specific task iteration defined by the process model
<code>generateProcessTasks</code>	Generate a list of the IDs for the task iterations defined by the process model
<code>getProcessTaskResults</code>	Get available results and result details for task iterations defined by the process model

runprocess

Generate and run model-based design (MBD) pipeline using build system

Syntax

```
[buildResult,exitCode] = runprocess()  
[buildResult,exitCode] = runprocess(Name=Value)
```

Description

[buildResult,exitCode] = runprocess() generate a model-based design (MBD) pipeline and run the pipeline using the build system. The process model (processmodel.p or processmodel.m) in the project defines the tasks for the pipeline.

[buildResult,exitCode] = runprocess(Name=Value) specifies how the MBD pipeline runs using one or more Name=Value arguments.

Examples

Run MBD Pipeline

Open a project and use runprocess to generate and run the MBD pipeline using the build system.

Open the **Process Advisor** example project, which contains an example process model. The process model defines the tasks for the pipeline.

```
processAdvisorExampleStart
```

Generate and run the MBD pipeline and store the results in the variable results.

```
results = runprocess()
```

Run Specific Tasks

Open a project and use runprocess. To only run a specific set of tasks, provide the task names to the Tasks argument.

Open the Process Advisor example project, which contains an example process model. The process model defines the tasks for the pipeline.

```
processAdvisorExampleStart
```

Run only the tasks **Generate Simulink Web View** (padv.builtin.task.GenerateSimulinkWebView) and **Check Modeling Standards** (padv.builtin.task.RunModelStandards) by specifying the Tasks argument.

```
% run the Generate Simulink Web View task  
% and the Check Modeling Standards tasks
```

```
runprocess(...
Tasks = ["padv.builtin.task.GenerateSimulinkWebView",...
"padv.builtin.task.RunModelStandards"])
```

Run Tasks Associated with Specific Artifact

Open a project and use `runprocess`. To only run the tasks associated with a specific artifact, provide a full path, relative path, or a `padv.Artifact` object to the `FilterArtifact` argument.

Open the Process Advisor example project, which contains an example process model. The process model defines the tasks for the pipeline.

```
processAdvisorExampleStart
```

Run tasks for the `AHRS_Voter` model by specifying the relative path to the model.

```
% run only the AHRS_Voter tasks
runprocess(...
FilterArtifact = fullfile(...
"02_Models", "AHRS_Voter", "specification", "AHRS_Voter.slx"))
```

Run Specific Task Iteration, Clean Task Results, and Delete Task Outputs

Open a project and run one specific task iteration in the pipeline.

Open the Process Advisor example project, which contains an example process model.

```
processAdvisorExampleStart
```

Get a list of the task iterations in the MBD pipeline.

```
tasks = generateProcessTasks;
```

Force `runprocess` to run one of the task iterations by specifying `Force` as `true` and `Tasks` as one of the tasks in `tasks`.

```
runprocess(Force=true, Tasks=tasks(1))
```

When `Force` is `true`, `runprocess` runs the pipeline, even if the pipeline already had results that were marked as up to date.

Clean task results and delete task outputs.

```
runprocess(Clean=true, DeleteOutputs=true)
```

When you clean task results and delete task outputs, it is as if the tasks were not run.

Input Arguments

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `[buildResult,exitCode] = runprocess(Force=true)`

Tasks — Names of tasks that you want to run

character vector | cell array of character vectors | string | string array

Names of tasks that you want to run, specified as a character vector, cell array of character vectors, string, or string array. The task name is defined by the `Name` property of the task.

Alternatively, you can specify the task iteration IDs for individual task iterations that you want to run. See "generateProcessTasks" and "createProcessTaskID" in this PDF for information.

Note You can only run tasks that are defined in the process model.

Example: `"padv.builtin.task.GenerateSimulinkWebView"`

Example: `["padv.builtin.task.GenerateSimulinkWebView", ...
"padv.builtin.task.RunModelStandards"]`

Data Types: `char` | `string`

Process — Name of process that you want to run

`padv.ProcessModel.DefaultProcessId` (default) | character vector | string

Name of process that you want to run, specified by a character vector or string.

Example: `"CIPipeline"`

Data Types: `char` | `string`

Subprocesses — Names of subprocesses that you want to run

character vector | cell array of character vectors | string | string array

Names of subprocesses that you want to run, specified as a character vector, cell array of character vectors, string, or string array. The subprocess name is defined by the `Name` property of the subprocess.

Example: `"SubprocessA"`

Example: `["SubprocessA",SubprocessB"]`

Data Types: `char` | `string`

FilterArtifact — Artifacts that you want to run tasks for

`string.empty` (default) | string | `padv.Artifact` object | array of `padv.Artifact` objects

Artifact or artifacts that you want to run tasks for, specified as either the full path to an artifact, relative path to an artifact, a `padv.Artifact` object that represents an artifact, or an array of `padv.Artifact` objects.

Example: `fullfile("C:\", "User", "projectA", "myModel.slx")`

Example: `fullfile("02_Models", "AHRS_Voter", "specification", "AHRS_Voter.slx")`

Example:

`padv.Artifact("sl_model_file",fullfile("02_Models", "AHRS_Voter", "specificatio
n", "AHRS_Voter.slx"))`

Data Types: `string`

Force — Skip or run up-to-date task iterations`false` or `0` (default) | `true` or `1`

Skip or run up-to-date tasks, specified as a numeric or logical `0` (`false`) or `1` (`true`). By default, `runprocess` does not run task iterations that have up to date results.

Example: `true`

Data Types: `logical`

Isolation — Include task dependencies`false` or `0` (default) | `true` or `1`

Include task dependencies, specified as a numeric or logical `0` (`false`) or `1` (`true`).

By default, `runprocess` includes task dependencies when running a task. Specify `Isolation` as `true` if you want to run a task in isolation, without running any task dependencies.

Note that you define task dependencies in the process model by using the function `dependsOn`.

Example: `true`

Data Types: `logical`

Clean — Clear task results and delete outputs`false` or `0` (default) | `true` or `1`

Clear task results and delete task outputs, specified as a numeric or logical `0` (`false`) or `1` (`true`).

If you specify `Clean` as `true`:

- The `runprocess` function ignores other name-value arguments, cleans the task results, and deletes task outputs.
- The `OutputDirectory` of the task might still contain files. The `runprocess` function only deletes the task outputs, specified by the `OutputPaths` property of the `padv.TaskResult` object for the task.
- You cannot specify `MarkStale` as `true`. The arguments are mutually exclusive.

Example: `true`

Data Types: `logical`

DeleteOutputs — Delete task outputs`false` or `0` (default) | `true` or `1`

Delete task outputs, specified as a numeric or logical `0` (`false`) or `1` (`true`).

Note To delete task outputs with `DeleteOutputs`, you must specify `Clean` as `true`.

Example: `true`

Data Types: `logical`

MarkStale — Mark task as outdated`false` or `0` (default) | `true` or `1`

Mark task as outdated, specified as a numeric or logical `0` (`false`) or `1` (`true`). When you mark a task as stale, the results appear outdated in the Process Advisor app.

Note If you specify `MarkStale` as `true`, then you cannot specify `Clean` as `true`. The arguments are mutually exclusive.

Example: `true`

Data Types: `logical`

ExitInBatchMode — Exit MATLAB when running in batch mode

`true` or `1` (default) | `false` or `0`

Exit MATLAB when running in batch mode, specified as a numeric or logical `1` (`true`) or `0` (`false`). By default, if you are running MATLAB in batch mode and `runprocess` finishes running, `runprocess` exits MATLAB.

The process exit codes are:

- `0` if the `Status` of `buildResult` is `PASS`
- `1` if the `Status` of `buildResult` is `ERROR`
- `2` if the `Status` of `buildResult` is `FAIL`

Example: `false`

Data Types: `logical`

GenerateReport — Automatically generate report at end of runprocess

`false` or `0` (default) | `true` or `1`

Automatically generate report after `runprocess` runs tasks, specified as a numeric or logical `1` (`true`) or `0` (`false`).

Example: `runprocess(GenerateReport = true)`

Data Types: `logical`

ReportFormat — File format for generated report

`"pdf"` (default) | `"html"` | `"html-file"` | `"docx"`

File format for the generated report, specified as one of these values:

- `"pdf"` — PDF file
- `"html"` — HTML report, packaged as a zipped file that contains the HTML file, images, style sheet, and JavaScript® files of the report
- `"html-file"` — HTML report
- `"docx"` — Microsoft® Word document

Note that for the `runprocess` function to generate a report, you must also specify the argument `GenerateReport` as `true`.

Example: `runprocess(GenerateReport = true, ReportFormat = "html-file")`

ReportPath — Name and path of generated report

"ProcessAdvisorReport" (default) | string array

Name and path of generated report, specified as a string array.

Note that for the `runprocess` function to generate a report, you must also specify the argument `GenerateReport` as `true`.

```
Example: runprocess(GenerateReport = true, ReportPath =
fullfile(pwd, "folderName", "reportName"))
```

Data Types: string

RerunFailedTasks — Rerun failed task iterations

false or 0 (default) | true or 1

Rerun failed task iterations, specified as a numeric or logical 0 (false) or 1 (true).

Example: true

Data Types: logical

RerunErroredTasks — Rerun errored task iterations

false or 0 (default) | true or 1

Rerun errored task iterations, specified as a numeric or logical 0 (false) or 1 (true).

Example: true

Data Types: logical

RefreshProcessModel — Automatically refresh before running tasks

true or 1 (default) | false or 0

Automatically refresh before running tasks, specified as a numeric or logical 1 (true) or 0 (false). By default, `runprocess` refreshes before running tasks so that the run uses the current state of the process model and project. If you specify `RefreshProcessModel` as `false`, `runprocess` does not refresh before running, but the run might not include the latest changes to tasks in the process model or artifacts in the project.

Example: false

Data Types: logical

ReanalyzeProjectAnalysisIssues — Automatically reanalyze project analysis issues that have severity level of error

true or 1 (default) | false or 0

Automatically reanalyze project analysis issues that have a severity level of error, specified as a numeric or logical 1 (true) or 0 (false).

If you are using R2022b Update 1 or later, you can specify `ReanalyzeProjectAnalysisIssues` as `false` to prevent the build system from reanalyzing project analysis issues that have a severity level of error. This might reduce the execution time for `runprocess`, but the build system might not generate the expected task iterations or detect outdated results.

Fix the issues listed in the **Project Analysis Issues** pane of the Process Advisor app to make sure the build system can fully analyze the project, generate the expected task iterations, and detect outdated results.

Example: `false`

Data Types: `logical`

GenerateJUnitForProcess — Generate JUnit-style XML report for process

`false` or `0` (default) | `true` or `1`

Generate JUnit-style XML report for each task in process, specified as a numeric or logical `0` (`false`) or `1` (`true`).

Example: `true`

Data Types: `logical`

EnableTaskLogging — Control command-line outputs from tasks

`logical.empty` (default) | `false` or `0` | `true` or `1`

Control command-line outputs from tasks, specified as:

- An empty logical array (`logical.empty`) — Tasks logging is disabled if the project setting `SuppressOutputWhenInteractive` is `true` and MATLAB is not running in batch mode.
- A numeric or logical `0` (`false`) — Task logging is disabled.
- A numeric or logical `1` (`true`) — Task logging is enabled.

When task logging is disabled, tasks no longer output information in the MATLAB Command Window.

Example: `false`

Data Types: `logical`

SuppressOutputWhenInteractive — Suppress command-line output from Process Advisor

`logical.empty` (default) | `1` or `true` | `0` or `false`

Suppress command-line output from Process Advisor during interactive MATLAB sessions, specified as either:

- An empty logical array (`logical.empty`) — No impact. `runprocess` follows the Process Advisor setting **Suppress outputs to command window**.
- A numeric or logical `1` (`true`) — Override the Process Advisor setting **Suppress outputs to command window** and suppress output to the MATLAB Command Window.
- A numeric or logical `0` (`false`) — Override the Process Advisor setting **Suppress outputs to command window** and show build logs and task execution messages in the MATLAB Command Window.

Note that this argument has no impact when you run MATLAB in batch mode, which is typically the case for CI systems.

Example: `true`

Data Types: `logical`

Output Arguments

buildResult — Results of run

`padv.BuildResult`

Results of run, returned as a `padv.BuildResult` object.

The `padv.BuildResult` object includes:

- The start time and end time of the run
- The status of the run (`Pass`, `Error`, `Fail`)
- Lists of the tasks that the passed, generated errors, were skipped, or failed during the run
- Input arguments to the run

exitCode — Exit code from run

0 | 1 | 2

Exit code from run, returned as a `double` representing the process error code.

- 0 if the `Status` of `buildResult` is `Pass`
- 1 if the `Status` of `buildResult` is `Error`
- 2 if the `Status` of `buildResult` is `Fail`

Alternative Functionality

App

You can also use the Process Advisor app to run each task or individual task iterations in the process. To open the Process Advisor app for a project, in the MATLAB Command Window, enter:

```
processAdvisorWindow
```

createProcessTaskID

Generate ID for specific task iteration defined by process model

Syntax

```
ID = createProcessTaskID(task, artifact)
```

Description

`ID = createProcessTaskID(task, artifact)` generates the identifier, `ID`, for an individual task iteration defined by the process model. A *task iteration* is the pairing of a task, `task`, to a specific project artifact, `artifact`.

Examples

Run One Task on One Artifact

Suppose you have a process model with several tasks, but right now you only want to run the task `padv.builtin.task.RunModelStandards` on the model `AHRS_Voter.slx`. Use the function `createProcessTaskID` to generate the ID for a specific task iteration, then use the function `runprocess` to run only that specific task iteration.

Open the **Process Advisor** example project, which contains an example process model.

```
processAdvisorExampleStart
```

Specify a task that exists in the process model. For this example, specify the built-in task for running Model Advisor checks, `padv.builtin.task.RunModelStandards`.

```
task = padv.builtin.task.RunModelStandards;
```

Use `padv.Artifact` to specify the project artifact that you want the task to run on. For this example, the artifact type is `sl_model_file` because the artifact is a Simulink model and the address is the path to `AHRS_Voter.slx`, relative to the project root.

```
artifactType = "sl_model_file";  
address = fullfile("02_Models", "AHRS_Voter", "specification", "AHRS_Voter.slx");  
artifact = padv.Artifact(artifactType, address);
```

Use the task instance and artifact to generate the ID for the specific task iteration.

```
runModelStandards_for_AHRS_Voter = createProcessTaskID(task, artifact)
```

```
runModelStandards_for_AHRS_Voter =
```

```
"padv.builtin.task.RunModelStandards|sl_model_file|02_Models/AHRS_Voter/specification/AHRS_Voter"
```

Use the function `runprocess` to run the task iteration.

```
runprocess(Tasks = runModelStandards_for_AHRS_Voter)
```

When you specify the `Tasks` value as the ID for a single task iteration, the function `runprocess` runs only the specified task iteration. For this example, `runprocess` runs only the task iteration associated with the task `padv.builtin.task.RunModelStandards` and the artifact `AHRS_Voter.slx`.

Note Alternatively, instead of creating and then running the task iterations, you can directly specify the `Task` and `FilterArtifact` arguments of the `runprocess` function to run the task on a specific artifact:

```
runprocess(...
Tasks = "padv.builtin.task.RunModelStandards",...
FilterArtifact = fullfile("02_Models","AHRS_Voter","specification","AHRS_Voter.slx"))
```

But note that you can only run the tasks if the tasks are defined in the process model and the artifacts exist in the project.

Input Arguments

task — Task name or subclass of `padv.Task`

string | character vector | `padv.Task` object

Either:

- Name of task, specified as a string or character vector. The name of a task is stored in the `Name` property of the task. For example, `"name_of_my_custom_task"`.
- Subclass of `padv.Task`, specified as a `padv.Task` object. Built-in tasks are subclasses of `padv.Task`. For example, you can specify the `padv.Task` object `padv.builtin.task.RunModelStandards` for the task argument.

Example: `"name_of_my_custom_task"`

Example: `"padv.builtin.task.RunModelStandards"`

Example: `padv.builtin.task.RunModelStandards`

Data Types: `char` | `string`

artifact — File in project

`padv.Artifact` object

File in project, specified as a `padv.Artifact` object.

Example: `padv.Artifact("project","ProcessAdvisorExample.prj")`

Example: `padv.Artifact("sl_model_file", "02_Models/AHRS_Voter/specification/AHRS_Voter.slx")`

Output Arguments

ID — Identifier for task iteration defined by process model

string

Identifier for task iteration defined by the process model, returned as a string.

IDs take the form: "*taskNameOrObject|fileType|relativePath*", where *relativePath* is the path relative to the project root.

Example IDs:

- "myCustomProjectTask|project|ProcessAdvisorExample.prj"
- "padv.builtin.task.RunModelStandards|sl_model_file|02_Models/AHRS_Voter/specification/AHRS_Voter.slx"
- "padv.builtin.task.RunTestsPerTestCase|sl_test_case|ced877ff-cfb8-4fa8-9bbf-aaa29b1d926b"

Alternative Functionality

App

You can also use the Process Advisor app to run individual task iterations in the process. To open the Process Advisor app for a project, in the MATLAB Command Window, enter:

```
processAdvisorWindow
```


generateProcessTasks

Get list of IDs for task iterations in MBD pipeline

Syntax

```
IDs = generateProcessTasks()
IDs = generateProcessTasks(Name=Value)
```

Description

`IDs = generateProcessTasks()` returns identifiers, `IDs`, for each of the task iterations in the model-based design (MBD) pipeline.

By default, `generateProcessTasks` returns an ID for each combination of tasks and associated project artifacts in the MBD pipeline.

`IDs = generateProcessTasks(Name=Value)` filters the list of IDs using one or more `Name=Value` arguments.

Examples

List IDs for Each Task Iteration in MBD Pipeline

Suppose you have a process model that adds several tasks to the process. Use the function `generateProcessTasks` to list the IDs for each task iteration in the MBD pipeline.

Open the Process Advisor example project, which contains an example process model.

```
processAdvisorExampleStart
```

List the IDs for each task iteration in the MBD pipeline.

```
IDs = generateProcessTasks()
```

Run Each Task Associated with an Artifact

Suppose you have a process model that adds several tasks to the process, but right now you only want to run the tasks associated with one specific artifact. You can use the function `generateProcessTasks`, but filter the list of IDs to only include task iterations associated with a specific model in the project, `AHRS_Voter.slx`.

Open the **Process Advisor** example project, which contains an example process model.

```
processAdvisorExampleStart
```

Use `padv.Artifact` to specify the project artifact that you want the task to run on. For this example, the artifact type is `sl_model_file` because the artifact is a Simulink model and the address is the path to `AHRS_Voter.slx`, relative to the project root.

```
artifactType = "sl_model_file";  
address = fullfile("02_Models", "AHRV_Voter", "specification", "AHRV_Voter.slx");  
artifact = padv.Artifact(artifactType, address);
```

Get a list of the IDs for the task iterations in the MBD pipeline, but filter the list to include only task iterations associated with the artifact `AHRV_Voter.slx`.

```
IDs_AHRV_Voter = generateProcessTasks(FilterArtifact=artifact);
```

Use the function `runprocess` to run only the task iterations associated with the artifact `AHRV_Voter.slx`.

```
runprocess(Tasks=IDs_AHRV_Voter)
```

When you specify the `Tasks` value as a list of IDs for task iterations, the function `runprocess` runs only the specified task iterations. For this example, `runprocess` runs only the task iterations associated with the artifact `AHRV_Voter.slx`.

Note Alternatively, instead of generating and then running the task iterations, you can directly specify the `FilterArtifact` argument of the `runprocess` function to run the tasks associated with the artifact:

```
runprocess(FilterArtifact = fullfile("02_Models", "AHRV_Voter", "specification", "AHRV_Voter.slx"))
```

But note that you can only run the tasks if the tasks are defined in the process model and the artifacts exist in the project.

Input Arguments

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `generateProcessTasks(Tasks = "padv.builtin.task.GenerateSimulinkWebView")`

FilterArtifact — Artifacts that you want to run tasks for

`string.empty` (default) | `string` | `padv.Artifact` object | array of `padv.Artifact` objects

Artifact or artifacts that you want to generate IDs for, specified as either the full path to an artifact, relative path to an artifact, a `padv.Artifact` object that represents an artifact, or an array of `padv.Artifact` objects.

Example: `fullfile("C:\", "User", "projectA", "myModel.slx")`

Example: `fullfile("02_Models", "AHRV_Voter", "specification", "AHRV_Voter.slx")`

Example:

```
padv.Artifact("sl_model_file", fullfile("02_Models", "AHRV_Voter", "specification", "AHRV_Voter.slx"))
```

Data Types: `string`

Process — Name of process that you want to generate IDs for

`padv.ProcessModel.DefaultProcessId` (default) | character vector | string

Name of process that you want to generate IDs for, specified by a character vector or string.

Example: "CIPipeline"

Data Types: char | string

Subprocesses — Names of subprocesses that you want to generate IDs for

character vector | cell array of character vectors | string | string array

Names of subprocesses that you want to generate IDs for, specified as a character vector, cell array of character vectors, string, or string array. The subprocess name is defined by the Name property of the subprocess.

Example: "SubprocessA"

Example: ["SubprocessA",SubprocessB"]

Data Types: char | string

Tasks — Names of tasks that you want to generate IDs for

character vector | cell array of character vectors | string | string array

Names of tasks that you want to generate IDs for, specified as a character vector, cell array of character vectors, string, or string array. The task name is defined by the Name property of the task.

Example: "padv.builtin.task.GenerateSimulinkWebView"

Example: ["padv.builtin.task.GenerateSimulinkWebView", ...
"padv.builtin.task.RunModelStandards"]

Data Types: char | string

Output Arguments**IDs — Identifiers for task iterations defined by process model**

string

Identifiers for task iterations in the MBD pipeline, returned as a string.

IDs take the form: "*taskNameOrObject* | *fileType* | *relativePath*", where *relativePath* is the path relative to the project root.

Example IDs:

- "myCustomProjectTask|project|ProcessAdvisorExample.prj"
- "padv.builtin.task.RunModelStandards|sl_model_file|02_Models/AHRS_Voter/specification/AHRS_Voter.slx"
- "padv.builtin.task.RunTestsPerTestCase|sl_test_case|ced877ff-cfb8-4fa8-9bbf-aaa29b1d926b"

Alternative Functionality

App

You can also use the Process Advisor app to run individual task iterations in the process or to view task iterations for a specific model.

- To open the Process Advisor app for a project, in the MATLAB Command Window, enter:
`processAdvisorWindow`
- To open the Process Advisor app for a specific model, provide the name of the model, *modelName*, to the function `processadvisor`:

```
processadvisor(modelName)
```

getProcessTaskResults

Get available task results and result details for task iterations in MBD pipeline

Syntax

```
[IDsWithTaskResults,taskResults,taskResultsOutdated] =
getProcessTaskResults()
[IDsWithTaskResults,taskResults,taskResultsOutdated] = getProcessTaskResults(
Name=Value)
```

Description

[IDsWithTaskResults,taskResults,taskResultsOutdated] = `getProcessTaskResults()` returns available task results and result details for the task iterations in the MBD pipeline. The function returns the identifiers for task iterations that have task results, `IDsWithTaskResults`, the current task results, `taskResults`, and a logical value that indicates if the task results are outdated, `taskResultsOutdated`.

If you do not have task results, use the function `runprocess` to run tasks and generate results. The function `getProcessTaskResults` only returns information related to task iterations that are defined in the process model. If you have task results from a task iteration that is not in the process model, the function does not return information related to those task results.

[IDsWithTaskResults,taskResults,taskResultsOutdated] = `getProcessTaskResults(Name=Value)` specifies options using one or more name-value arguments.

Examples

Get Output Artifacts from Task Results

Get the available task results for a task iteration and use the result details to find information about the output artifacts of the task iteration.

Open the Process Advisor example project, which contains an example process model.

```
processAdvisorExampleStart
```

List the IDs for each task iteration in the MBD pipeline.

```
IDs = generateProcessTasks();
```

Run the first task iteration in the list.

```
runprocess(Tasks=IDs(1))
```

For this example, the build system runs the task `padv.builtin.task.GenerateSimulinkWebView` for the model `AHRS_Voter.slx`.

Get the available task results and result details.

```
[IDsWithResults,results,outdated] = getProcessTaskResults()
IDsWithResults =
    "padv.builtin.task.GenerateSimulinkWebView|sl_model_file|02_Models/AHRS_Voter/specification/"

results =
    TaskResult with properties:
        Status: Pass
        OutputArtifacts: [1x1 padv.Artifact]
        Details: [1x1 struct]
        ResultValues: [1x1 struct]

outdated =
    logical
    0
```

Get the output artifacts from the result. For this example, the result is a Simulink Web View for the model `AHRS_Voter.slx`.

```
webView = results.OutputArtifacts
webView =
    Artifact with properties:
        Type: "padv_output_file"
        Parent: [0x0 padv.Artifact]
        ArtifactAddress: [1x1 padv.util.ArtifactAddress]
        Alias: ""
```

Get Output Artifacts from Task Results for Specific Model

Get the available task results for a specific model.

Open the Process Advisor example project, which contains an example process model.

```
processAdvisorExampleStart
```

Check modeling standards for the model `AHRS_Voter.slx` by using the built-in task `padv.builtin.task.RunModelStandards`. The task uses Model Advisor to run checks on the model.

```
runprocess(...
Tasks = "padv.builtin.task.RunModelStandards",...
FilterArtifact = fullfile("02_Models","AHRS_Voter","specification","AHRS_Voter.slx"));
```

Get the task results and result details.

```
[IDsWithResults,results,outdated] = getProcessTaskResults(...
Tasks = "padv.builtin.task.RunModelStandards",...
FilterArtifact = fullfile("02_Models","AHRV_Voter","specification","AHRV_Voter.slx"))

IDsWithResults =

    "padv.builtin.task.RunModelStandards|sl_model_file|ProcessAdvisorExample|02_Models/AHRV_Voter

results =

    TaskResult with properties:

        Status: Pass
    OutputArtifacts: [1x1 padv.Artifact]
        Details: [1x1 struct]
    ResultValues: [1x1 struct]

outdated =

    logical

    0
```

Input Arguments

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `[~,results,~] = getProcessTaskResults(Tasks="maTask", FilterArtifact=fullfile("models","myModel.slx"));`

Tasks — Names of tasks that you want to run

character vector | cell array of character vectors | string | string array

Names of tasks that you want to run, specified as a character vector, cell array of character vectors, string, or string array. The task name is defined by the `Name` property of the task.

Alternatively, you can specify the task iteration IDs for individual task iterations that you want to run. See "generateProcessTasks" and "createProcessTaskID" in this PDF for information.

Note You can only run tasks that are defined in the process model.

Example: "padv.builtin.task.GenerateSimulinkWebView"

Example: ["padv.builtin.task.GenerateSimulinkWebView", ... "padv.builtin.task.RunModelStandards"]

Data Types: char | string

Process — Name of process that you want to run

padv.ProcessModel.DefaultProcessId (default) | character vector | string

Name of process that you want to run, specified by a character vector or string.

Example: "CIPipeline"

Data Types: char | string

Subprocesses — Names of subprocesses that you want to run

character vector | cell array of character vectors | string | string array

Names of subprocesses that you want to run, specified as a character vector, cell array of character vectors, string, or string array. The subprocess name is defined by the `Name` property of the subprocess.

Example: "SubprocessA"

Example: ["SubprocessA", SubprocessB"]

Data Types: char | string

FilterArtifact — Artifacts that you want to run tasks for

string.empty (default) | string | padv.Artifact object | array of padv.Artifact objects

Artifact or artifacts that you want to run tasks for, specified as either the full path to an artifact, relative path to an artifact, a `padv.Artifact` object that represents an artifact, or an array of `padv.Artifact` objects.

Example: `fullfile("C:\", "User", "projectA", "myModel.slx")`

Example: `fullfile("02_Models", "AHRV_Voter", "specification", "AHRV_Voter.slx")`

Example:

`padv.Artifact("sl_model_file", fullfile("02_Models", "AHRV_Voter", "specification", "AHRV_Voter.slx"))`

Data Types: string

Output Arguments

IDsWithTaskResults — Identifiers for task iterations that have task results and are defined in process model

string | string array

Identifiers for task iterations that have task results and are defined in the process model, returned as a string or string array.

- If you do not have task results for task iterations in your process model, `IDsWithTaskResults` returns an empty array, []. You can use the function `runprocess` to run tasks and generate results.
- If you have task results for task iterations that are not in your process model, `IDsWithTaskResults` returns an empty array, [].
- If you have task results for task iterations that are in your process model, `IDsWithTaskResults` returns the IDs for the task iterations that have task results.

IDs take the form: "*taskNameOrObject* | *fileType* | *relativePath*", where *relativePath* is the path relative to the project root.

Example IDs:

- "myCustomProjectTask|project|ProcessAdvisorExample.prj"
- "padv.builtin.task.RunModelStandards|sl_model_file|02_Models/AHRS_Voter/specification/AHRS_Voter.slx"
- "padv.builtin.task.RunTestsPerTestCase|sl_test_case|ced877ff-cfb8-4fa8-9bbf-aaa29b1d926b"

taskResults – Results for task iterations

padv.TaskResult | padv.TaskResult array

Results for task iterations, returned as a padv.TaskResult or padv.TaskResult array.

- If you do not have task results for task iterations in your process model, taskResults returns an empty array, [].
- If you have task results for task iterations that are not in your process model, taskResults returns an empty array, [].
- If you have task results for task iterations that are in your process model, taskResults returns a padv.TaskResult or padv.TaskResult array.

padv.TaskResult objects contain properties for the result status, output artifacts, details, and result values for the number of passing, warning, and failing results for task iterations.

taskResultsOutdated – Whether task results are outdated or up-to-date

logical | logical array

Status of task results, returned as a logical value or logical array. Values of 1 indicate that the results for the task iteration are outdated and might not reflect the current state of the project or task. Values of 0 indicate that the results for the task iteration are up-to-date. The result is an empty array, [], when there are not task results.

padv.BuildResult

Result from build system build

Description

Use the build result, `padv.BuildResult`, to find the properties of the build system build, including a list of the tasks that the build system ran and the settings the build system used.

Creation

Syntax

Description

`buildResultObj = padv.BuildResult()` stores the results from a build system build.

Properties

StartTime — Start time of build

`datetime`

Start time of build, returned as `datetime`.

Example: 09-Aug-2022 14:32:05

Data Types: `datetime`

EndTime — End time of build

`datetime`

End time of build, returned as `datetime`.

Example: 09-Aug-2022 14:32:37

Data Types: `datetime`

Status — Overall status for build

`Pass` (default) | `Fail` | `Error`

Overall status for build, returned as the `padv.TaskStatus` enumeration value:

- `Error` if any task iteration in the build returns an error.
- `Fail` if no task iterations in the build return an error, but at least one task iteration fails.
- `Pass` if no task iterations were run, or if no task iterations in the build return an error or fail.

Example: `Pass`

ResultValues — Task iteration result values

`[1×1 struct]` (default)

Task iteration result values, returned as a structure array with fields:

- Pass
- Warn
- Fail

For example, if the build runs one task iteration and the task iteration returns one passing result and five warning results, the structure array contains:

```
struct with fields:

    Pass: 1
    Warn: 5
    Fail: 0
```

Data Types: struct

PassTasks — IDs for task iterations that passed during the build

cell array

IDs for task iterations that passed during the build, returned as a cell array.

If the build system runs one task iteration and the task iteration passes, `PassTasks` returns a one-dimensional cell array. For example, if the build system only ran the task `padv.builtin.task.GenerateCode` on the model `AHRS_Voter.slx` and the task iteration passed, `PassTasks` returns:

```
{'padv.builtin.task.GenerateCode|sl_model_file|02_Models/AHRS_Voter/specification/AHRS_Voter.slx'}
```

If multiple task iterations pass, `PassTasks` returns one cell for each task iteration that passed. For example:

```
{'padv.builtin.task.GenerateCode|sl_model_file|02_Models/AHRS_Voter/specification/AHRS_Voter.slx'
'padv.builtin.task.GenerateCode|sl_model_file|02_Models/Actuator_Control/specification/Actuator_Control.slx'
'padv.builtin.task.GenerateCode|sl_model_file|02_Models/Flight_Control/specification/Flight_Control.slx'
'padv.builtin.task.GenerateCode|sl_model_file|02_Models/InnerLoop_Control/specification/InnerLoop_Control.slx'
'padv.builtin.task.GenerateCode|sl_model_file|02_Models/OuterLoop_Control/specification/OuterLoop_Control.slx'}
```

Data Types: cell

ErrorTasks — IDs for task iterations that returned an error during the build

cell array

IDs for task iterations that returned an error during the build, returned as a cell array.

If the build system runs one task iteration and the task iteration returns an error, `ErrorTasks` returns a one-dimensional cell array. For example, if the build system tried to run a custom task, `customTask`, on the model `AHRS_Voter.slx`, but the task iteration returned an error, `ErrorTasks` returns:

```
{'customTask|sl_model_file|02_Models/AHRS_Voter/specification/AHRS_Voter.slx'}
```

If multiple task iterations error, `ErrorTasks` returns one cell for each task iteration that returned an error. For example:

```
{'customTask|sl_model_file|02_Models/AHRS_Voter/specification/AHRS_Voter.slx' }
{'customTask|sl_model_file|02_Models/Actuator_Control/specification/Actuator_Control.slx' }
```

```
{'customTask|sl_model_file|02_Models/Flight_Control/specification/Flight_Control.slx'      }
{'customTask|sl_model_file|02_Models/InnerLoop_Control/specification/InnerLoop_Control.slx'}
{'customTask|sl_model_file|02_Models/OuterLoop_Control/specification/OuterLoop_Control.slx'}
```

Data Types: cell

SkippedTasks — IDs for task iterations that the build system skipped

cell array

IDs for task iterations that the build system skipped, returned as a cell array. The build system skips task iterations that already have up-to-date results, unless you specify `Force` as `true` when you call the function `runprocess`.

If the build system skips one task iteration, `SkippedTasks` returns a one-dimensional cell array. For example, if you instructed the build system to run the task `padv.builtin.task.GenerateCode` on the model `AHRS_Voter.slx`, but the task iteration already had up-to-date results, `SkippedTasks` returns:

```
{'padv.builtin.task.GenerateCode|sl_model_file|02_Models/AHRS_Voter/specification/AHRS_Voter.slx'}
```

If the build system skips multiple task iterations, `SkippedTasks` returns one cell for each task iteration that the build system skipped. For example:

```
{'padv.builtin.task.GenerateCode|sl_model_file|02_Models/AHRS_Voter/specification/AHRS_Voter.slx'
{'padv.builtin.task.GenerateCode|sl_model_file|02_Models/Actuator_Control/specification/Actuator_
{'padv.builtin.task.GenerateCode|sl_model_file|02_Models/Flight_Control/specification/Flight_Con
{'padv.builtin.task.GenerateCode|sl_model_file|02_Models/InnerLoop_Control/specification/InnerLo
{'padv.builtin.task.GenerateCode|sl_model_file|02_Models/OuterLoop_Control/specification/OuterLo
```

Data Types: cell

FailedTasks — IDs for task iterations that failed during the build

cell array

IDs for task iterations that failed during the build, returned as a cell array.

If the build system runs only one task iteration and the task iteration fails, `FailedTasks` returns a one-dimensional cell array. For example, if the build system ran the task `padv.builtin.task.GenerateCode` on the model `AHRS_Voter.slx` and the task iteration failed, `FailedTasks` returns:

```
{'padv.builtin.task.GenerateCode|sl_model_file|02_Models/AHRS_Voter/specification/AHRS_Voter.slx'}
```

If multiple task iterations fail, `FailedTasks` returns one cell for each task iteration that failed. For example:

```
{'padv.builtin.task.GenerateCode|sl_model_file|02_Models/AHRS_Voter/specification/AHRS_Voter.slx'
{'padv.builtin.task.GenerateCode|sl_model_file|02_Models/Actuator_Control/specification/Actuator_
{'padv.builtin.task.GenerateCode|sl_model_file|02_Models/Flight_Control/specification/Flight_Con
{'padv.builtin.task.GenerateCode|sl_model_file|02_Models/InnerLoop_Control/specification/InnerLo
{'padv.builtin.task.GenerateCode|sl_model_file|02_Models/OuterLoop_Control/specification/OuterLo
```

Data Types: cell

InputArgs — Input arguments that defined how the build system ran the build

[1×1 struct] (default) | structure array

Input arguments that defined how the build system ran the build, returned as a structure array with fields:

- `TasksToBuild` — List of task iteration IDs that you want the build system to run
- `Isolation` — Setting to include or ignore task dependencies
- `Force` — Setting to skip or run up-to-date task iterations
- `RerunFailedTasks` — Setting to ignore or rerun failed task iterations
- `RerunErroredTasks` — Setting to ignore or rerun task iterations that returned an error

For example, the `InputArgs` for a build result could return:

```
struct with fields:
    TasksToBuild: [1x5 string]
    Isolation: 0
    Force: 0
    RerunFailedTasks: 0
    RerunErroredTasks: 0
```

For more information, see the function `runprocess`.

Data Types: `struct`

Examples

Get List of Passed Task Iterations and Build Settings

Open a project, run a build, and use the build result, `padv.BuildResult`, to get a list of the passed task iterations and the settings that the build system used when running the build.

Open the **Process Advisor** example project, which contains an example process model.

```
processAdvisorExampleStart
```

Generate a list of the tasks defined by the process model.

```
tasks = generateProcessTasks;
```

Run the first five task iterations in `tasks` and specify `Force` as `true`.

```
buildResult = runprocess(Force=true,Tasks=tasks(1:5))
```

Use the build result, `buildResult`, to get a list of the task iterations that passed.

```
passed = buildResult.PassTasks'
```

```
passed =
```

```
5x1 cell array
```

```
{'padv.builtin.task.GenerateSimulinkWebView|sl_model_file|02_Models/AHRS_Voter/specification,
{'padv.builtin.task.GenerateSimulinkWebView|sl_model_file|02_Models/Actuator_Control/specific
{'padv.builtin.task.GenerateSimulinkWebView|sl_model_file|02_Models/Flight_Control/specifica
{'padv.builtin.task.GenerateSimulinkWebView|sl_model_file|02_Models/InnerLoop_Control/specif
{'padv.builtin.task.GenerateSimulinkWebView|sl_model_file|02_Models/OuterLoop_Control/specif:
```

When you used the function `runprocess`, you specified `Force` as `true`. You can see that information in the `InputArgs` property of the build result, `buildResult`.

```
runprocessInputs = buildResult.InputArgs
```

```
runprocessInputs =
```

```
    struct with fields:
```

```
        TasksToBuild: ["padv.builtin.task.GenerateSimulinkWebView|sl_model_file|02_Models/AHRS_V...
            Isolation: 0
            Force: 1
        RerunFailedTasks: 0
        RerunErroredTasks: 0
```

The build result shows that the `Force` setting was `1 (true)` when the build system ran.

padv.Preferences

(To be removed) Specify settings for build system

Description

There are several settings that you can use to customize the behavior of the build system. These behaviors impact how the Process Advisor app and `runprocess` function run tasks. For example, you can use settings to use incremental builds, enable model caching, and customize other behaviors. The build system saves these settings in `padv.Preferences`. You can use the preferences, `padv.Preferences`, to specify settings for the Process Advisor app and settings for how the `runprocess` function runs builds.

Note The `padv.Preferences` class will be removed in a future release. Use the new classes `padv.ProjectSettings` and `padv.UserSettings` instead. The new classes allow you to programmatically control the settings for incremental builds, build system logging, and other behaviors, without needing to create a project startup script to persist run-time settings.

For information, see the "Version History" for `padv.Preferences` below.

Creation

Syntax

Description

`P = padv.Preferences()` gets the handle to the global preferences object, `P`. There is only one set of preference properties.

The `padv.Preferences` class is a `handle` class.

Properties

Project Settings

These settings are stored in the project and are shared with everyone using the project.

IncrementalBuild — Automatically detect changes and mark task results as outdated

1 (true) | 0 (false)

Automatically detect changes and mark task results as outdated, specified as a numeric or logical 1 (true) or 0 (false).

When `IncrementalBuild` is `true` and you make a change to an artifact in your project, the build system marks any related task results as outdated.

This property is equivalent to the **Incremental build** setting in the Process Advisor Settings dialog box.

Example: `true`

Data Types: `logical`

EnableModelCaching — Allow build system to cache models during build

0 (false) (default) | 1 (true)

Allow the build system to cache models during a build, specified as a numeric or logical 1 (true) or 0 (false).

If you specify the property `EnableModelCaching` as `true`, you allow the build system to cache models instead of reloading the same models multiple times within a build. For information, see "Cache Models Used During Build" in the User's Guide PDF.

This property is equivalent to the **Enable model caching** setting in the Process Advisor Settings dialog box.

Example: `true`

Data Types: `logical`

MaxNumModelsInCache — Maximum number of models in cache

1 (default) | positive value

Maximum number of models in the model cache, specified as a positive value.

Example: 2

MaxNumTestResultsInCache — Maximum number of test results in cache

20 (default) | positive value

Maximum number of test results in the cache, specified as a positive value.

Example: 30

SuppressOutputWhenInteractive — Suppress command-line output from Process Advisor

0 (false) (default) | 1 (true)

Suppress command-line output from Process Advisor during interactive MATLAB sessions, specified as a numeric or logical 1 (true) or 0 (false).

You can use this setting to suppress command-line outputs from the build system, such as the build log and task execution messages from Process Advisor and the `runprocess` function.

Note that the build system automatically ignores this setting when you run MATLAB in batch mode, which is typically the case for CI systems.

This property is equivalent to the **Suppress outputs to command window** setting in the Process Advisor Settings dialog box.

Example: `true`

Data Types: `logical`

Run-Time Settings

DetectDuplicateOutputs — Generate error message when multiple tasks attempt to write to same output file

1 (true) (default) | 0 (false)

Setting that controls whether the build system generates an error message when multiple tasks attempt to write to the same output file, specified as a numeric or logical 1 (true) or 0 (false).

By default, the build system generates an error if multiple tasks attempt to write to the same output file.

This property is equivalent to the **Detect duplicate outputs** setting in the Process Advisor Settings dialog box.

Example: false

Data Types: logical

GarbageCollectTaskOutputs — Setting for automatically cleaning task results for tasks and artifacts that do not match current process model or project

true or 1 (default) | false or 0

Setting for automatically cleaning task results for tasks and artifacts that do not match current process model or project, specified as a numeric or logical 1 (true) or 0 (false).

By default, when you use the build system, the build system cleans task results that are no longer relevant for the current process model or project. For example, if you had task results from a specific task and then you remove that task from the process model, the build system automatically deletes the task results associated with the task. If you had task results associated with a specific project artifact and then you removed that artifact from the project, the build system automatically deletes the task results associated with the artifact. Note that the build system does not delete generated artifacts like generated code.

If you specify `GarbageCollectTaskOutputs` as false, the build system does not automatically clean task results associated with tasks and artifacts that are not in the current process model or project.

This property is equivalent to the **Garbage collect task outputs** setting in the Process Advisor Settings dialog box.

Example: false

Data Types: logical

FilteredDigitalThreadMessages — List of filtered digital thread messages

[13×1 string] (default) | string

List of filtered digital thread messages, specified as a string.

By default, Process Advisor and the build system do not display certain messages from the digital thread. You can add or remove messages in the list, or reset the list of filtered messages, by using the `padv.Preferences` object functions. For information, see the Object Functions for `padv.Preferences`.

Data Types: string

ShowDetailedErrorMessage — Setting to show more information in error messages

false or 0 (default) | true or 1

Setting to show more information in error messages, specified as a numeric or logical 0 (false) or 1 (true).

By default, error messages from the build system are not verbose.

If you specify `ShowDetailedErrorMessages` as `true`, the build system shows full stack traces in error messages. You might want to see full stack traces when you are debugging a process model.

This property is equivalent to the **Show detailed error messages** setting in the Process Advisor Settings dialog box.

Example: `true`

Data Types: `logical`

TrackProcessModel — Setting for tracking changes to process model

`true` or `1` (default) | `false` or `0`

Setting for tracking changes to process model, specified as a numeric or logical `1` (`true`) or `0` (`false`).

By default, if you make a change to the process model file, the build system marks each task status and task result as outdated because the tasks in the updated process model might not match the tasks that generated the task results from the previous version of the process model. For example, if you ran the built-in task `padv.builtin.task.RunModelStandards` with the default Model Advisor configuration, updated the process model to specify a different Model Advisor configuration file for the task, and then ran the task again, the task results are now outdated because they are the task results from the default configuration.

If you specify `TrackProcessModel` as `false` and make a change to the process model, the build system will not mark the task statuses and task results as outdated.

This property is equivalent to the **Add process model as dependency** setting in the Process Advisor Settings dialog box.

Example: `false`

Data Types: `logical`

Object Functions

- `addFilteredDigitalThreadMessages(obj, IssueId)` adds the message, specified by the issue ID `IssueId`, to the list of filtered messages in the property `FilteredDigitalThreadMessages`.

To get a list of issue messages and issue IDs, use the function `getArtifactIssues`:

```
metric_engine = metric.Engine();
issues = getArtifactIssues(metric_engine)
issuesMessages = issues.IssueMessage
issueIDs = issues.IssueId
```

Suppose that you want to filter out the issue message associated with the issue ID `"alm:artifact_service:CannotResolveElement"`. You can use the function `addFilteredDigitalThreadMessages` to add the issue message to the list of filtered messages:

```
p = padv.Preferences;
addFilteredDigitalThreadMessages(p, ...
    "alm:artifact_service:CannotResolveElement")
```

- `removeFilteredDigitalThreadMessages(obj, IssueId)` removes the message, specified by `messageID`, to the list of filtered messages in the property `FilteredDigitalThreadMessages`.

For example:

```
p = padv.Preferences;
removeFilteredDigitalThreadMessages(p, ...
    "alm:simulink_handlers:ModelCallbacksDeactivated")
```

- `resetFilteredDigitalThreadMessages(obj)` resets the list of filtered messages in the property `FilteredDigitalThreadMessages`.

For example:

```
p = padv.Preferences;
resetFilteredDigitalThreadMessages(p)
```

Examples

Specify Preferences for Builds

Use `padv.Preferences` to specify preferences for the Process Advisor app and build system.

Create a `padv.Preferences` object.

```
PREF = padv.Preferences
```

Specify `IncrementalBuild` as 0.

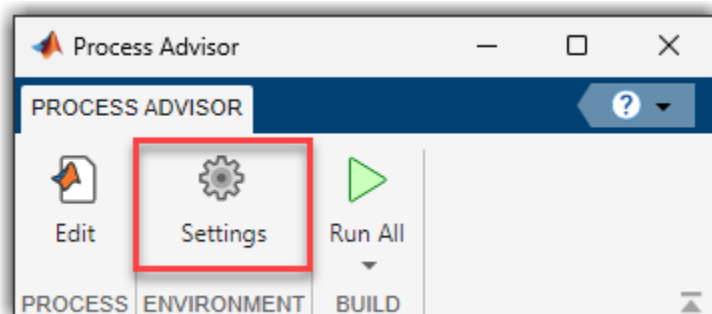
```
PREF.IncrementalBuild = 0;
```

Now, when you run tasks, incremental builds are disabled and the build system forces tasks to run, even if the tasks have up to date results.

Alternative Functionality

App

In Process Advisor, in the toolstrip, click **Settings** to access and change the settings for the build system.



Version History

R2022b: `padv.Preferences` class will be removed in a future release

Warns starting in R2022b

The class `padv.Preferences` will be removed in a future release. Update your code to replace instances of `padv.Preferences` with either `padv.UserSettings.get()` or `padv.ProjectSettings.get()`, depending on which property you need to access.

padv.Preferences Property	Update
DetectDuplicateOutputs	Replace instances of <code>padv.Preferences</code> with <code>padv.UserSettings.get()</code> .
GarbageCollectTaskOutputs	
ShowDetailedErrorMessage	
TrackProcessModel	
FilteredDigitalThreadMessages	Replace instances of <code>padv.Preferences</code> with <code>padv.ProjectSettings.get()</code> .
IncrementalBuild	
EnableModelCaching	
MaxNumModelsInCache	
MaxNumTestResultsInCache	
SuppressOutputWhenInteractive	

For example:

Functionality	Use This Instead
<pre>% changing run-time setting p1 = padv.Preferences; p1.DetectDuplicateOutputs = false;</pre>	<pre>p1 = padv.UserSettings.get(); p1.DetectDuplicateOutputs = false;</pre>
<pre>% changing project setting p1 = padv.Preferences; p1.IncrementalBuild = false;</pre>	<pre>p1 = padv.ProjectSettings.get(); p1.IncrementalBuild = false;</pre>

padv.ProjectSettings Class

Namespace: padv

Build system settings for project

Description

The `padv.ProjectSettings` class is a handle class.

Creation

Syntax

```
padv.ProjectSettings
```

Description

`padv.ProjectSettings` is a handle class that you can use to customize the behavior of the build system. These behaviors impact how the Process Advisor app and `runprocess` function run tasks. For example, you can use the project settings to use incremental builds, enable model caching, and customize other behaviors.

Project settings are persistent, are stored in the project, and are shared with everyone using the project. There is only one set of project settings for a project. To get the active project settings object, use the `get` method.

To specify settings that apply only to your machine, use `padv.UserSettings`.

Properties

IncrementalBuild — Automatically detect changes and mark task results as outdated

```
1 (true) | 0 (false)
```

Automatically detect changes and mark task results as outdated, specified as a numeric or logical 1 (true) or 0 (false).

When `IncrementalBuild` is `true` and you make a change to an artifact in your project, the build system marks any related task results as outdated.

This property is equivalent to the **Incremental build** setting in the Process Advisor Settings dialog box.

Example: `true`

Attributes:

```
GetAccess          public
SetAccess          public
```

Data Types: `logical`

EnableModelCaching — Allow build system to cache models during build

0 (false) | 1 (true)

Allow the build system to cache models during a build, specified as a numeric or logical 1 (true) or 0 (false).

If you specify the property `EnableModelCaching` as `true`, you allow the build system to cache models instead of reloading the same models multiple times within a build. For information, see "Cache Models Used During Build" in the User's Guide PDF.

This property is equivalent to the **Enable model caching** setting in the Process Advisor Settings dialog box.

Example: `true`

Attributes:

<code>GetAccess</code>	<code>public</code>
<code>SetAccess</code>	<code>public</code>

Data Types: `logical`

MaxNumModelsInCache — Maximum number of models in cache

1 (default) | positive value

Maximum number of models in the model cache, specified as a positive value.

For information about caching, see "Cache Models and Other Artifacts Used During Build" in the User's Guide PDF.

Example: `2`

Attributes:

<code>GetAccess</code>	<code>public</code>
<code>SetAccess</code>	<code>public</code>

MaxNumTestResultsInCache — Maximum number of test results in cache

20 (default) | positive value

Maximum number of test results in the cache, specified as a positive value.

For information about caching, see "Cache Models and Other Artifacts Used During Build" in the User's Guide PDF.

Example: `30`

Attributes:

<code>GetAccess</code>	<code>public</code>
<code>SetAccess</code>	<code>public</code>

SuppressOutputWhenInteractive — Suppress command-line output from Process Advisor

0 (false) (default) | 1 (true)

Suppress command-line output from Process Advisor during interactive MATLAB sessions, specified as a numeric or logical 1 (true) or 0 (false).

You can use this setting to suppress command-line outputs from the build system, such as the build log and task execution messages from Process Advisor and the `runprocess` function.

Note that the build system automatically ignores this setting when you run MATLAB in batch mode, which is typically the case for CI systems.

This property is equivalent to the **Suppress outputs to command window** setting in the Process Advisor Settings dialog box. If you want to override this setting when you use the function `runprocess`, you can use the `runprocess` argument `SuppressOutputWhenInteractive`.

Example: `true`

Attributes:

<code>GetAccess</code>	<code>public</code>
<code>SetAccess</code>	<code>public</code>

Data Types: `logical`

ShowFileExtension — Show file extensions for task iteration artifacts

`0` (false) | `1` (true)

Show file extensions for task iteration artifacts, specified as a numeric or logical `1` (true) or `0` (false).

By default, queries strip file extensions from the `Alias` property of each task iteration artifact. The `Alias` property controls the display name for the artifact in the **Tasks** column in Process Advisor.

To show file extensions for all task iteration artifacts in the **Tasks** column, specify this setting as `true`. To keep file extensions in the results for a specific query, specify the query property `ShowFileExtension` as `true`.

This property is equivalent to the **Show file extensions** setting in the Process Advisor Settings dialog box.

Example: `true`

Attributes:

<code>GetAccess</code>	<code>public</code>
<code>SetAccess</code>	<code>public</code>

Data Types: `logical`

FilteredDigitalThreadMessages — List of filtered digital thread messages

`[13×1 string]` (default) | `string`

List of filtered digital thread messages, specified as a string.

By default, Process Advisor and the build system do not display certain messages from the digital thread. You can add or remove messages in the list, or reset the list of filtered messages, by using the methods for `padv.ProjectSettings`. For information, see the "Methods" section below.

Attributes:

<code>GetAccess</code>	<code>public</code>
<code>SetAccess</code>	<code>public</code>

Data Types: string

Methods

Public Methods

Get or Reset Settings for Project

Method	Description
get	Get build system settings for current project PREF = padv.ProjectSettings.get()
resetToDefaultValues	Reset build system settings for current project PREF.resetToDefaultValues() To see the changes, use the get method to get the latest setting values. PREF = padv.ProjectSettings.get()

Filter Messages

Method	Description
addFilteredDigitalThreadMessages	Add message to list of filtered messages ps = padv.ProjectSettings.get(); ps.addFilteredDigitalThreadMessages(... "alm:artifact_service:CannotResolveElement"); To get a list of issue messages and issue IDs, use the function getArtifactIssues: metric_engine = metric.Engine(); issues = getArtifactIssues(metric_engine) issuesMessages = issues.IssueMessage issueIDs = issues.IssueId
removeFilteredDigitalThreadMessages	Remove message from list of filtered messages ps = padv.ProjectSettings.get(); ps.removeFilteredDigitalThreadMessages(... "alm:simulink_handlers:ModelCallbacksDeactivated");
resetFilteredDigitalThreadMessages	Reset list of filtered messages ps = padv.ProjectSettings.get(); ps.resetFilteredDigitalThreadMessages();

Examples

Get Build System Settings for Project

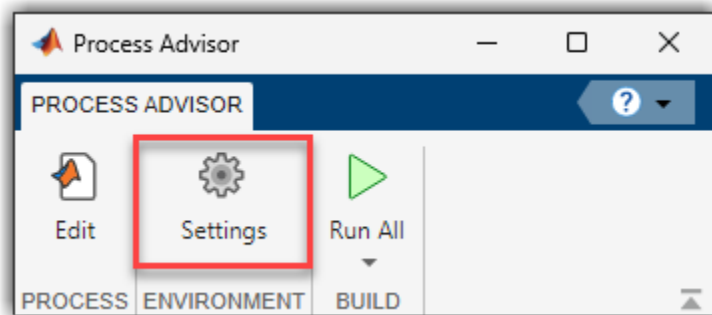
Get for build system settings for the currently open project.


```
PREF = padv.ProjectSettings.get()
```

Alternative Functionality

App

In Process Advisor, in the toolbar, click **Settings** to access and change the settings for the build system.



padv.UserSettings Class

Namespace: padv

Build system settings for user

Description

The `padv.UserSettings` class is a `handle` class.

Creation

Syntax

```
padv.UserSettings
```

Description

`padv.UserSettings` is a `handle` class that you can use to customize the behavior of the build system on your machine. These behaviors impact how the Process Advisor app and `runprocess` function run tasks on your machine. For example, you can use the user settings to show detailed error messages, remove the process model as a dependency, and customize other behaviors.

User settings are persistent and do not reset when you restart MATLAB or call `clear classes`. There is only one set of user settings. To get the active user settings object, use the `get` method.

To specify settings that apply to everyone that uses your project, use `padv.ProjectSettings`.

Properties

DetectDuplicateOutputs — Generate error message when multiple tasks attempt to write to same output file

1 (true) (default) | 0 (false)

Setting that controls whether the build system generates an error message when multiple tasks attempt to write to the same output file, specified as a numeric or logical 1 (true) or 0 (false).

By default, the build system generates an error if multiple tasks attempt to write to the same output file.

This property is equivalent to the **Detect duplicate outputs** setting in the Process Advisor Settings dialog box.

Example: `false`

Data Types: `logical`

GarbageCollectTaskOutputs — Setting for automatically cleaning task results for tasks and artifacts that do not match current process model or project

true or 1 (default) | false or 0

Setting for automatically cleaning task results for tasks and artifacts that do not match current process model or project, specified as a numeric or logical 1 (`true`) or 0 (`false`).

By default, when you use the build system, the build system cleans task results that are no longer relevant for the current process model or project. For example, if you had task results from a specific task and then you remove that task from the process model, the build system automatically deletes the task results associated with the task. If you had task results associated with a specific project artifact and then you removed that artifact from the project, the build system automatically deletes the task results associated with the artifact. Note that the build system does not delete generated artifacts like generated code.

If you specify `GarbageCollectTaskOutputs` as `false`, the build system does not automatically clean task results associated with tasks and artifacts that are not in the current process model or project.

This property is equivalent to the **Garbage collect task outputs** setting in the Process Advisor Settings dialog box.

Example: `false`

Data Types: `logical`

ShowDetailedErrorMessages — Setting to show more information in error messages

`false` or 0 (default) | `true` or 1

Setting to show more information in error messages, specified as a numeric or logical 0 (`false`) or 1 (`true`).

By default, error messages from the build system are not verbose.

If you specify `ShowDetailedErrorMessages` as `true`, the build system shows full stack traces in error messages. You might want to see full stack traces when you are debugging a process model.

This property is equivalent to the **Show detailed error messages** setting in the Process Advisor Settings dialog box.

Example: `true`

Data Types: `logical`

TrackProcessModel — Setting for tracking changes to process model

`true` or 1 (default) | `false` or 0

Setting for tracking changes to process model, specified as a numeric or logical 1 (`true`) or 0 (`false`).

By default, if you make a change to the process model file, the build system marks each task status and task result as outdated because the tasks in the updated process model might not match the tasks that generated the task results from the previous version of the process model. For example, if you ran the built-in task `padv.builtin.task.RunModelStandards` with the default Model Advisor configuration, updated the process model to specify a different Model Advisor configuration file for the task, and then ran the task again, the task results are now outdated because they are the task results from the default configuration.

If you specify `TrackProcessModel` as `false` and make a change to the process model, the build system will not mark the task statuses and task results as outdated.

This property is equivalent to the **Add process model as dependency** setting in the Process Advisor Settings dialog box.

Example: `false`

Data Types: `logical`

Methods

Public Methods

Get Settings for User

Method	Description
<code>get</code>	Get build system settings for current user <code>PREF = padv.UserSettings.get()</code>
<code>resetToDefaultValues</code>	Reset build system settings for current user <code>PREF.resetToDefaultValues()</code> To see the changes, use the <code>get</code> method to get the latest setting values. <code>PREF = padv.UserSettings.get()</code>

Examples

Get Build System Settings for User

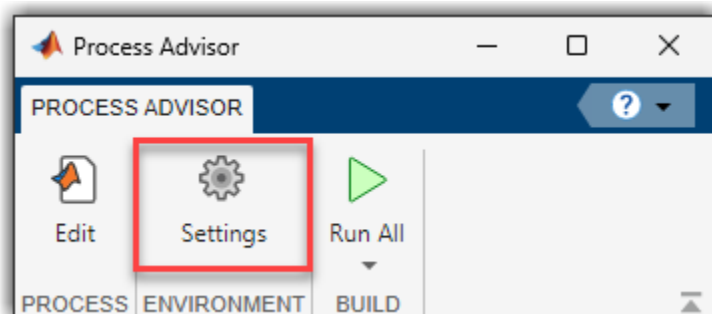
Get for build system settings for the current user.

```
PREF = padv.UserSettings.get()
```

Alternative Functionality

App

In Process Advisor, in the toolbar, click **Settings** to access and change the settings for the build system.



Pipeline Generator API

The support package provides example pipeline configuration files that you can add to your project to automatically execute your pipeline on a continuous integration (CI) platform, like GitHub® Actions, GitLab®, and Jenkins®. The example pipeline configuration files use the pipeline generator API to automatically generate and execute pipelines for your specific project and process so that you do not need to manually update any pipeline files when you make changes to your project.

For examples of how to integrate into a specific CI platform, see the "Integrate into CI" chapter in the user's guide.

Classes

CI Platform Options

Class	Description
<code>padv.pipeline.GitHubOptions</code>	Settings that control how a generated GitHub pipeline runs
<code>padv.pipeline.GitLabOptions</code>	Settings that control how a generated GitLab pipeline runs
<code>padv.pipeline.JenkinsOptions</code>	Settings that control how a generated Jenkins pipeline runs

Functions

Generate Pipeline for CI

Function	Description
<code>padv.pipeline.generatePipeline</code>	Generate pipeline configuration file for CI platform

padv.pipeline.generatePipeline

Namespace: padv.pipeline

Generate pipeline file for CI platform

Syntax

```
generatorResults = padv.pipeline.generatePipeline(platformOptions)
```

Description

`generatorResults = padv.pipeline.generatePipeline(platformOptions)` generates a pipeline file for the CI platform and options specified by `platformOptions`. The function `padv.pipeline.generatePipeline` is a pipeline generator that can automatically generate a pipeline file. The generated pipeline file can configure a pipeline that runs your process in CI.

Examples

Generate YML File for GitLab Pipeline

Suppose that you want to run your process using GitLab.

```
padv.pipeline.generatePipeline(padv.pipeline.GitLabOptions)
```

The generated pipeline file is 'simulink_pipeline.yml'.

For information on how to use the pipeline generator to integrate into GitLab, see "Integrate into GitLab".

Generate Jenkinsfile for Jenkins Pipeline

Suppose that you want to run your process using Jenkins.

```
padv.pipeline.generatePipeline(padv.pipeline.JenkinsOptions)
```

The generated pipeline file is 'simulink_pipeline'.

For information on how to use the pipeline generator to integrate into Jenkins, see "Integrate into Jenkins".

Input Arguments

platformOptions — Options for generating CI pipeline

`padv.pipeline.GitLabOptions` object | `padv.pipeline.JenkinsOptions` object

Options for generating CI pipeline, specified as:

- A `padv.pipeline.GitLabOptions` object to generate a YML file that you can use to run the generated pipeline in a GitLab CI system.
- A `padv.pipeline.JenkinsOptions` object to generate a Jenkinsfile that you can use to run the generated pipeline in Jenkins CI system.

Example: `padv.pipeline.generatePipeline(padv.pipeline.GitLabOptions)`

Example: `padv.pipeline.generatePipeline(padv.pipeline.JenkinsOptions)`

Output Arguments

generatorResults — Results from pipeline generator

`padv.pipeline.GeneratorResults` object

Results from pipeline generator, returned as a `padv.pipeline.GeneratorResults` object. The filename for the generated pipeline file is stored in the property `GeneratedPipelineFiles`.

padv.pipeline.GitHubOptions

Options for generating GitHub pipeline configuration file

Description

Use the `padv.pipeline.GitHubOptions` object to represent the desired options for generating a GitHub pipeline configuration file. To generate a GitHub pipeline configuration file, use `padv.pipeline.GitHubOptions` as an input argument to the `padv.pipeline.generatePipeline` function.

Note For information on how to use the pipeline generator to integrate into a GitHub CI system, see "Integrate into GitHub".

Note If you run MATLAB using the `-nodisplay` option or you use a machine that does not have a display (like many CI runners and Docker® containers), you should set up a virtual display server before you include the following built-in tasks in your process model:

- **Generate SDD Report**
- **Generate Simulink Web View**
- **Generate Model Comparison**

For information, see "Set Up Virtual Display for No-Display Machine" in the User's Guide.

Creation

Description

`options = padv.pipeline.GitHubOptions` returns configuration options for generating a GitHub pipeline configuration file.

`options = padv.pipeline.GitHubOptions(Name=Value)` sets properties using one or more name-value arguments. For example, `padv.pipeline.GitHubOptions(RunnerLabels = "Linux")` creates an options object that specifies that a generated pipeline configuration file use Linux as the GitHub Action runner label.

Properties

RunnerLabels — GitHub runner labels

"self-hosted" (default) | string

GitHub runner labels, specified as a string.

The labels determine which GitHub runner can execute the job. For more information, see <https://docs.github.com/en/actions/using-jobs/choosing-the-runner-for-a-job#targeting-runners-in-a-group>.

Example: `padv.pipeline.GitHubOptions(RunnerLabels = "Linux")`

Data Types: string

ArtifactZipFileName — Name of ZIP file for job artifacts

"padv_artifacts.zip" (default) | string

Name of ZIP file for job artifacts, specified as a string.

Example: `padv.pipeline.GitHubOptions(ArtifactZipFileName = "my_job_artifacts.zip")`

Data Types: string

RetentionDays — How many days GitHub stores workflow artifacts

"30" (default) | string

How many days GitHub stores workflow artifacts, specified as a string. This property corresponds to the job keyword "retention-days" in GitHub. After the specified number of retention days, the artifacts expire and GitHub deletes the artifacts.

Example: `padv.pipeline.GitHubOptions(RetentionDays = "90")`

Data Types: string

GeneratedYMLFileName — File name of generated GitLab pipeline file

"simulink_pipeline" (default) | string

File name of generated GitLab pipeline file, specified as a string.

By default, the generated pipeline generates into the subfolder **derived > pipeline**, relative to the project root. To change where the pipeline file generates, specify `GeneratedPipelineDirectory`.

Example: `padv.pipeline.GitHubOptions(GeneratedYMLFileName = "padv_generated_pipeline_file")`

Data Types: string

MatlabInstallationLocation — Path to MATLAB installation location

"PATH_TO_MATLAB" (default) | string

Path to MATLAB installation location, specified as a string.

Make sure the path that you specify uses the correct MATLAB root folder location and file separators for the operating system of your GitHub runner.

Example: "C:\Program Files\MATLAB\R2023a\bin"

Example: "/usr/local/MATLAB/R2023a/bin"

Example: "/Applications/MATLAB_R2023a.app/bin"

Data Types: string

EnableArtifactCollection — When to collect build artifacts

"always", 1, or true (default) | "never", 0, or false | "on_success" | "on_failure"

When to collect build artifacts, specified as:

- "never", 0, or false — Never collect artifacts
- "on_success" — Only collect artifacts when the pipeline succeeds

- "on_failure" — Only collect artifacts when the pipeline fails
- "always", 1, or true — Always collect artifacts

If the pipeline collects artifacts, the child pipeline contains a job, `Collect_Artifacts`, that compresses the build artifacts into a ZIP file and attaches the file to the job.

Example: `padv.pipeline.GitHubOptions(EnableArtifactCollection=false)`

Data Types: `logical | string`

ShellEnvironment — Shell environment GitHub uses to launch MATLAB

"bash" (default) | "pwsh"

Shell environment GitHub uses to launch MATLAB, specified as one of these values:

- "bash" — UNIX® shell script
- "pwsh" — PowerShell Core script

Example: `padv.pipeline.GitHubOptions(ShellEnvironment = "pwsh")`

Data Types: `string`

CheckoutSubmodules — Checkout Git™ submodules

"false" (default) | "true" | "recursive"

Checkout Git submodules at the beginning of each pipeline stage, specified as either:

- "false"
- "true"
- "recursive"

This property uses the GitHub Action `checkout@v3`. For information about the submodule input values, see <https://github.com/marketplace/actions/checkout-submodules>.

Example: `padv.pipeline.GitHubOptions(CheckoutSubmodules = "true")`

Data Types: `string`

PipelineArchitecture — Number of stages and grouping of tasks in CI pipeline

`padv.pipeline.Architecture.SingleStage` (default) |

`padv.pipeline.Architecture.SerialStages` |

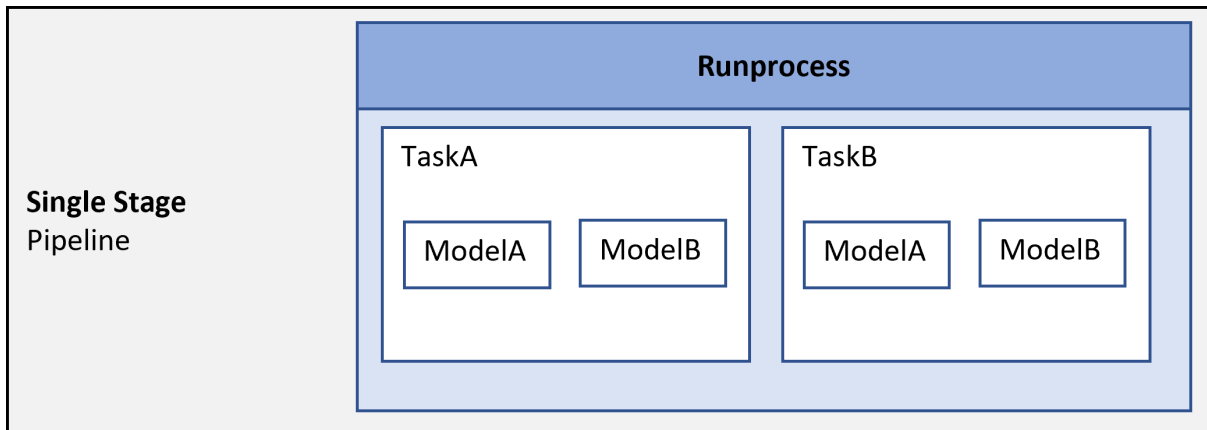
`padv.pipeline.Architecture.SerialStagesGroupPerTask`

Number of stages and grouping of tasks in CI pipeline, specified as either:

- `padv.pipeline.Architecture.SingleStage` — Single stage runs all tasks

For example, a pipeline with one stage that runs each of the tasks in the process:

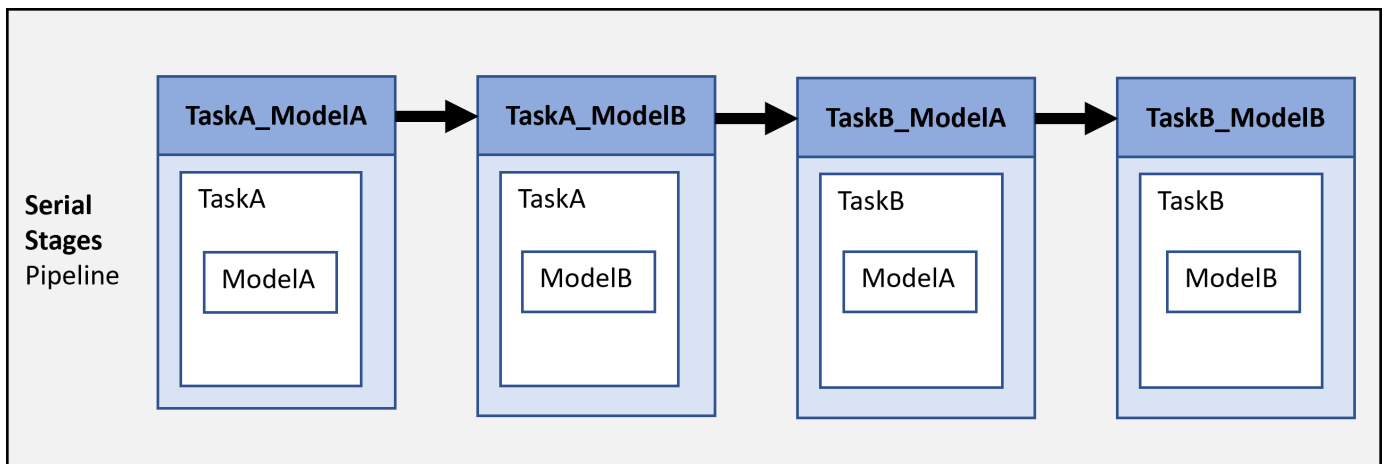
1 Runprocess



- `padv.pipeline.Architecture.SerialStages` — One stage for each task iteration

For example, a pipeline with four stages:

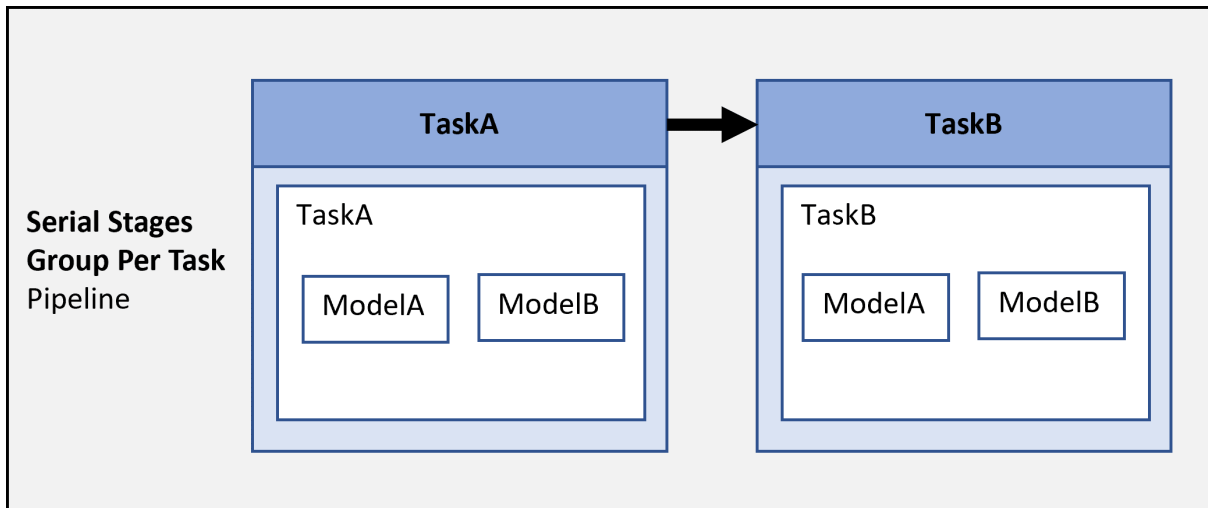
- 1 **TaskA_ModelA** — Runs a task TaskA on the model ModelA
- 2 **TaskA_ModelB** — Runs a task TaskA on the model ModelB
- 3 **TaskB_ModelA** — Runs a task TaskB on the model ModelA
- 4 **TaskB_ModelB** — Runs a task TaskB on the model ModelB



- `padv.pipeline.Architecture.SerialStagesGroupPerTask` — One stage for each type of task

For example, a pipeline with two stages:

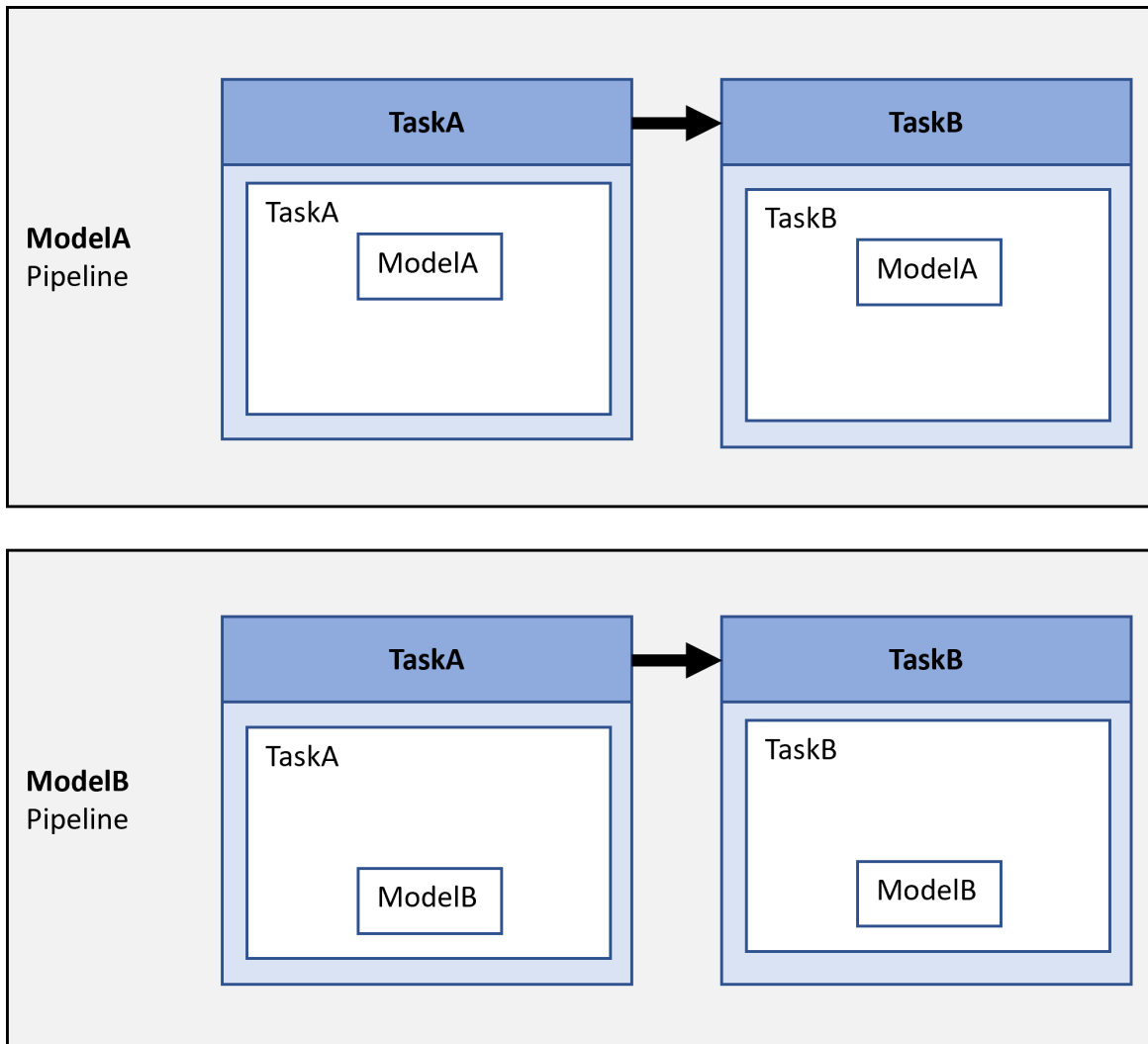
- 1 **TaskA** — Runs a task TaskA on each model in the project
- 2 **TaskB** — Runs a task TaskB on each model in the project



- `padv.pipeline.Architecture.IndependentModelPipelines`— Parallel, downstream pipelines for each model. Each pipeline independently runs the tasks associated with the model.

For example, a pipeline with parallel downstream pipelines:

- **ModelA** — Runs TaskA and TaskB on ModelA.
- **ModelB** — Runs TaskA and TaskB on ModelB.



Example: `padv.pipeline.GitHubOptions(PipelineArchitecture = padv.pipeline.Architecture.SerialStages)`

ForceRunAllTasks – Pipeline runs both up to date and outdated tasks

0 (false) (default) | 1 (true)

Pipeline runs both up to date and outdated tasks, specified as a numeric or logical 1 (true) or 0 (false).

The property defines the Force argument for the `runprocess` function in the generated pipeline file.

Example: `padv.pipeline.GitHubOptions(ForceRunAllTasks=true)`

Data Types: logical

ExitInBatchMode – Exits MATLAB if MATLAB was run with -batch startup option

1 (true) (default) | 0 (false)

Exits MATLAB if MATLAB was run with the `-batch` startup option, specified as a numeric or logical `0` (`false`) or `1` (`true`).

This property defines the `ExitInBatchMode` argument for the `runprocess` function in the generated pipeline file.

Example: `padv.pipeline.GitHubOptions(ExitInBatchMode=false)`

Data Types: `logical`

RerunFailedTasks — Treats all tasks which previously failed as being outdated

`0` (`false`) (default) | `1` (`true`)

Treats all tasks which previously failed as being outdated, specified as a numeric or logical `1` (`true`) or `0` (`false`).

This property defines the `RerunFailedTasks` argument for the `runprocess` function in the generated pipeline file.

Example: `padv.pipeline.GitHubOptions(RerunFailedTasks=true)`

Data Types: `logical`

RerunErroredTasks — Treats all tasks which previously generated errors as outdated

`0` (`false`) (default) | `1` (`true`)

Treats all tasks which previously generated errors as outdated, specified as a numeric or logical `1` (`true`) or `0` (`false`).

This property defines the `RerunErroredTasks` argument for the `runprocess` function in the generated pipeline file.

Example: `padv.pipeline.GitHubOptions(RerunErroredTasks=true)`

Data Types: `logical`

MatlabLaunchCmd — Command to start MATLAB program

`"matlab"` (default) | `string`

Command to start MATLAB program, specified as a string.

Use this property to specify how the pipeline starts the MATLAB program. This property defines how the script in the generated pipeline file launches MATLAB.

Example: `padv.pipeline.GitHubOptions(MatlabLaunchCmd = "matlab")`

Data Types: `string`

MatlabStartupOptions — Command-line startup options for MATLAB

`"-nodesktop -logfile output.log"` (default) | `string`

Command-line startup options for MATLAB, specified as a string.

Use this property to specify the command-line startup options that the pipeline uses when starting the MATLAB program. This property defines the command-line startup options that appear next to the `-batch` option and `MatlabLaunchCmd` value in the `"script"` section of the generated pipeline file. The pipeline starts MATLAB with the specified startup options.

By default, the support package launches MATLAB using the `-batch` option. If you need to run MATLAB without the `-batch` option, specify the property `AddBatchStartupOption` as `false`.

Note If you run MATLAB using the `-nodisplay` option or you use a machine that does not have a display (like many CI runners and Docker containers), you should set up a virtual display server before you include the following built-in tasks in your process model:

- **Generate SDD Report**
- **Generate Simulink Web View**
- **Generate Model Comparison**

For information, see "Set Up Virtual Display for No-Display Machine" in the User's Guide.

Example: `padv.pipeline.GitHubOptions(MatlabStartupOptions = "-nodesktop -logfile mylogfile.log")`

Data Types: `string`

AddBatchStartupOption — Specify whether to open MATLAB using `-batch` startup option

`1` (true) (default) | `0` (false)

Specify whether to open MATLAB using `-batch` startup option, specified as a numeric or logical `0` (false) or `1` (true).

By default, the support package launches MATLAB in CI using the `-batch` startup option.

If you need to launch MATLAB with options that are not compatible with `-batch`, specify `AddBatchStartupOption` as `false`.

Example: `padv.pipeline.GitHubOptions(AddBatchStartupOption = false)`

Data Types: `logical`

GeneratedPipelineDirectory — Specify where the generated pipeline file generates

`fullfile("derived","pipeline")` (default) | `string`

Specify where the generated pipeline file generates, specified as a string.

This property defines the directory where the generated pipeline file generates.

By default, the generated pipeline file is named `"simulink_pipeline.yml"`. To change the name of the generated pipeline file, specify `GeneratedYMLFileName`.

Example: `padv.pipeline.GitHubOptions(GeneratedPipelineDirectory = fullfile("derived","pipeline","test"))`

Data Types: `string`

GenerateReport — Generate Process Advisor build report

`true` or `1` (default) | `false` or `0`

Generate Process Advisor build report, specified as a numeric or logical `1` (true) or `0` (false).

Example: `padv.pipeline.GitHubOptions(GenerateReport = false)`

Data Types: `logical`

ReportFormat — File format for generated report

"pdf" (default) | "html" | "html-file" | "docx"

File format for the generated report, specified as one of these values:

- "pdf" — PDF file
- "html" — HTML report, packaged as a zipped file that contains the HTML file, images, style sheet, and JavaScript files of the report
- "html-file" — HTML report
- "docx" — Microsoft Word document

Example: `padv.pipeline.GitHubOptions(ReportFormat = "html-file")`

ReportPath — Name and path of generated report

"ProcessAdvisorReport" (default) | string array

Name and path of generated report, specified as a string array.

By default, the report generates in the current working folder with the name "ProcessAdvisorReport".

Example: `padv.pipeline.GitHubOptions(ReportPath = "myReport")`

Data Types: `string`

StopOnStageFailure — Stop running pipeline after stage fails

0 (false) (default) | 1 (true)

Stop running pipeline after stage fails, specified as a numeric or logical 0 (false) or 1 (true).

By default, the pipeline continues to run, even if a stage in the pipeline fails.

Example: `padv.pipeline.GitHubOptions(StopOnStageFailure = true)`

Data Types: `logical`

CheckOutdatedResultsAfterMerge — Check for outdated results after merge

1 (true) (default) | 0 (false)

Check for outdated results after merge, specified as a numeric or logical 1 (true) or 0 (false).

When specified as `true`, the pipeline checks if task results are still up-to-date after merging artifact database files from parallel jobs. Outdated results are not expected if the merge is successful. If there are outdated results, there could be an issue with the merge.

Example: `false`

Data Types: `logical`

Examples

Specify GitHub Configuration Options When Generating Pipeline Configuration File

Create a `padv.pipeline.GitHubOptions` object and change the options. When you generate a pipeline configuration file, the file uses the specified options.

This example shows how to use the pipeline generator API. For information on how to use the pipeline generator to integrate into a GitHub CI system, see "Integrate into GitHub".

Load a project. For this example, you can load a Process Advisor example project. In the MATLAB Command Window, enter:

```
processAdvisorExampleStart
```

Specify your GitHub pipeline configuration options by creating a `padv.pipeline.GitHubOptions` object and modifying the object properties. For example, if you have a GitHub runner that uses a MATLAB installation at `/opt/matlab/r2023a`:

```
GitHubOptions = padv.pipeline.GitHubOptions  
GitHubOptions.MatlabInstallationLocation = "/opt/matlab/r2023a";
```

Generate a GitHub pipeline configuration file by using the function `padv.pipeline.generatePipeline` with the specified options.

```
padv.pipeline.generatePipeline(GitHubOptions);
```

Note Calling `padv.pipeline.generatePipeline(GitHubOptions)` is equivalent to calling `padv.pipeline.generateGitHubPipeline(GitHubOptions)`.

By default, the generated pipeline configuration file is named `simulink_pipeline.yml` and is located under the project root, in the subfolder **derived > pipeline**.

The `GeneratedYMLFileName` and `GeneratedPipelineDirectory` properties of the `padv.pipeline.GitHubOptions` object control the name and location of the generated pipeline configuration file.

For information on how to use the pipeline generator to integrate into a GitHub CI system, see "Integrate into GitHub" in the User's Guide.

padv.pipeline.GitLabOptions

Options for generating GitLab pipeline configuration file

Description

Use the `padv.pipeline.GitLabOptions` object to represent the desired options for generating a GitLab pipeline configuration file. To generate a GitLab pipeline configuration file, use `padv.pipeline.GitLabOptions` as an input argument to the `padv.pipeline.generatePipeline` function.

Note For information on how to use the pipeline generator to integrate into a GitLab CI system, see "Integrate into GitLab".

Note If you run MATLAB using the `-nodisplay` option or you use a machine that does not have a display (like many CI runners and Docker containers), you should set up a virtual display server before you include the following built-in tasks in your process model:

- **Generate SDD Report**
- **Generate Simulink Web View**
- **Generate Model Comparison**

For information, see "Set Up Virtual Display for No-Display Machine" in the User's Guide.

Creation

Syntax

```
options = padv.pipeline.GitLabOptions  
options = padv.pipeline.GitLabOptions(Name=Value)
```

Description

`options = padv.pipeline.GitLabOptions` returns configuration options for generating a GitLab pipeline configuration file.

`options = padv.pipeline.GitLabOptions(Name=Value)` sets properties using one or more name-value arguments. For example, `padv.pipeline.GitLabOptions(Tags="high_memory")` creates an options object that specifies that a generated pipeline configuration file use `high_memory` as the GitLab CI/CD tag.

Properties

Tags — GitLab CI/CD tags

string | string array

GitLab CI/CD tags, specified as a string or string array. Use this property to specify the tags that appear next to the `tags` keyword in a generated GitLab pipeline configuration file.

The GitLab CI/CD tags select a GitLab Runner for a job. The property `Tags` specifies which CI/CD tags appear next to the `tags` keyword in a generated pipeline configuration file.

For more information on the `tags` keyword, see <https://docs.gitlab.com/ee/ci/yaml/#tags>.

Example: `options = padv.pipeline.GitLabOptions(Tags="high_memory")`

Data Types: `string`

EnableArtifactCollection — When to collect build artifacts

"always", 1, or true (default) | "never", 0, or false | "on_success" | "on_failure"

When to collect build artifacts, specified as:

- "never", 0, or false — Never collect artifacts
- "on_success" — Only collect artifacts when the pipeline succeeds
- "on_failure" — Only collect artifacts when the pipeline fails
- "always", 1, or true — Always collect artifacts

If the pipeline collects artifacts, the child pipeline contains a job, `Collect_Artifacts`, that compresses the build artifacts into a ZIP file and attaches the file to the job.

This property creates an "artifacts" section in the generated pipeline file. For more information, see the GitLab documentation: <https://docs.gitlab.com/ee/ci/yaml/#artifacts>.

Example: `padv.pipeline.GitLabOptions(EnableArtifactCollection="on_failure")`

Data Types: `logical | string`

ArtifactZipFileName — Name of ZIP file for job artifacts

"padv_artifacts.zip" (default) | string

Name of ZIP file for job artifacts, specified as a string.

This property specifies the file name that appears next to the "name" keyword in the generated pipeline file. For more information, see the GitLab documentation for "artifacts:name": <https://docs.gitlab.com/ee/ci/yaml/#artifactsname>.

Example: `padv.pipeline.GitLabOptions(ArtifactZipFileName = "my_job_artifacts.zip")`

Data Types: `string`

ArtifactsExpireIn — How long GitLab stores job artifacts before the artifacts expire

"30 days" (default) |

How long GitLab stores job artifacts before the artifacts expire, specified as a string.

Use this property to specify how long GitLab stores job artifacts before the artifacts expire and GitLab deletes the artifacts. This property specifies the expiry time that appears next to the "expire_in" keyword in the generated pipeline file. For a list of valid possible inputs, see the GitLab documentation for "artifacts:expire_in": https://docs.gitlab.com/ee/ci/yaml/#artifactsexpire_in.

Example: `padv.pipeline.GitLabOptions(ArtifactsExpireIn = "60 days")`

Data Types: `string`

ArtifactsWhen — When GitLab uploads job artifacts

"always" (default) | "on_success" | "on_failure"

Warning This property will be removed in a future release. Use the property `EnableArtifactCollection` instead.

When GitLab uploads job artifacts, specified as either:

- "on_success"
- "on_failure"
- "always"

Use this property to specify when GitLab uploads job artifacts. This property specifies the input that appears next to the "when" keyword in the generated pipeline file. For more information, see the GitLab documentation for "artifacts:when": <https://docs.gitlab.com/ee/ci/yaml/#artifactswhen>.

Example: `padv.pipeline.GitLabOptions(ArtifactsWhen = "on_success")`

GeneratedYMLFileName — File name of generated GitLab pipeline file

"simulink_pipeline" (default) | `string`

File name of generated GitLab pipeline file, specified as a string.

By default, the generated pipeline generates into the subfolder **derived > pipeline**, relative to the project root. To change where the pipeline file generates, specify `GeneratedPipelineDirectory`.

Example: `padv.pipeline.GitLabOptions(GeneratedYMLFileName = "padv_generated_pipeline_file")`

Data Types: `string`

PipelineArchitecture — Number of stages and grouping of tasks in CI pipeline

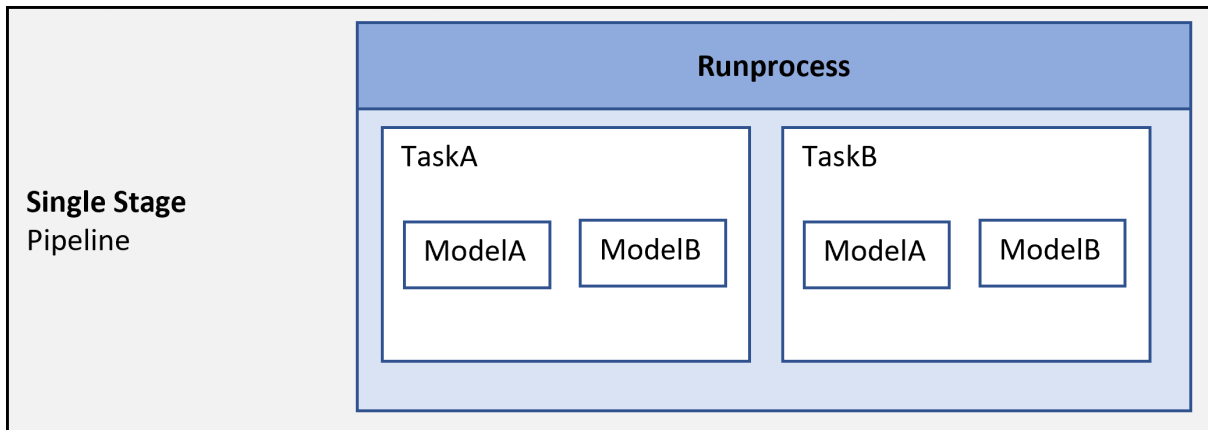
`padv.pipeline.Architecture.SingleStage` (default) |
`padv.pipeline.Architecture.SerialStages` |
`padv.pipeline.Architecture.SerialStagesGroupPerTask`

Number of stages and grouping of tasks in CI pipeline, specified as either:

- `padv.pipeline.Architecture.SingleStage` — Single stage runs all tasks

For example, a pipeline with one stage that runs each of the tasks in the process:

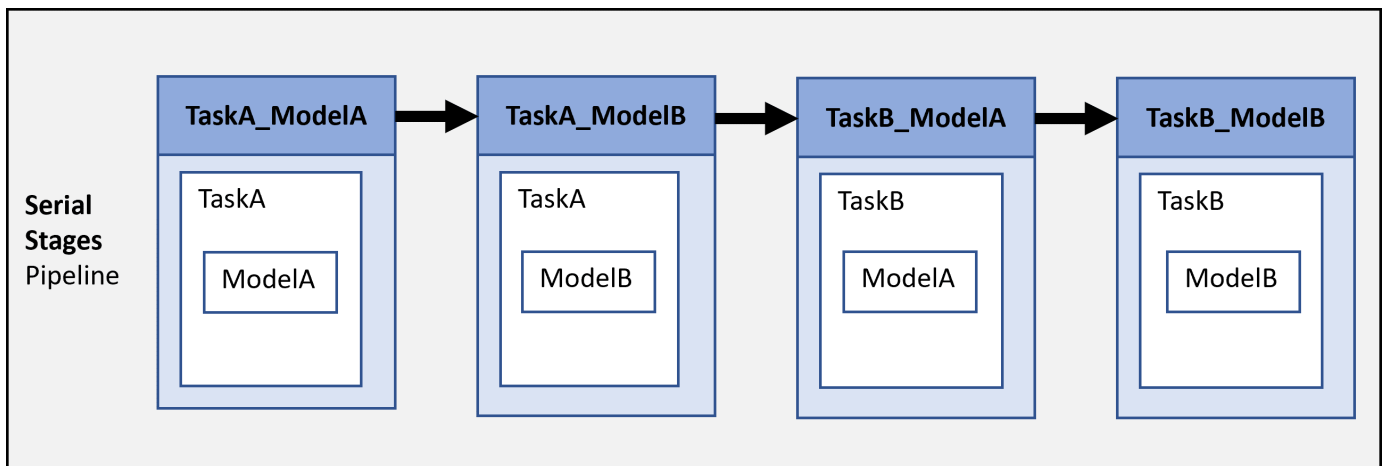
1 Runprocess



- `padv.pipeline.Architecture.SerialStages` — One stage for each task iteration

For example, a pipeline with four stages:

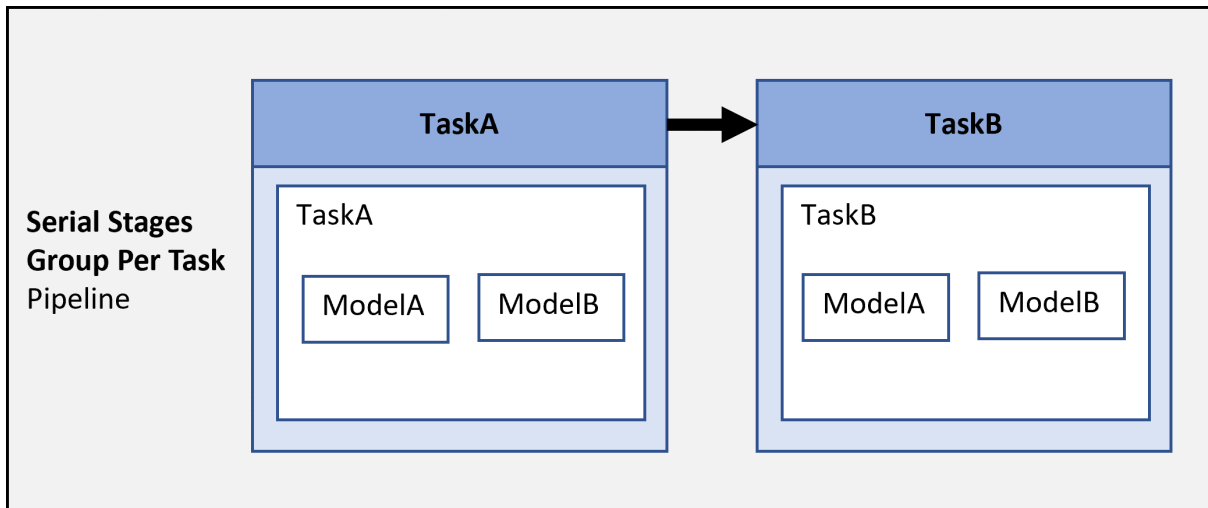
- 1 **TaskA_ModelA** — Runs a task TaskA on the model ModelA
- 2 **TaskA_ModelB** — Runs a task TaskA on the model ModelB
- 3 **TaskB_ModelA** — Runs a task TaskB on the model ModelA
- 4 **TaskB_ModelB** — Runs a task TaskB on the model ModelB



- `padv.pipeline.Architecture.SerialStagesGroupPerTask` — One stage for each type of task

For example, a pipeline with two stages:

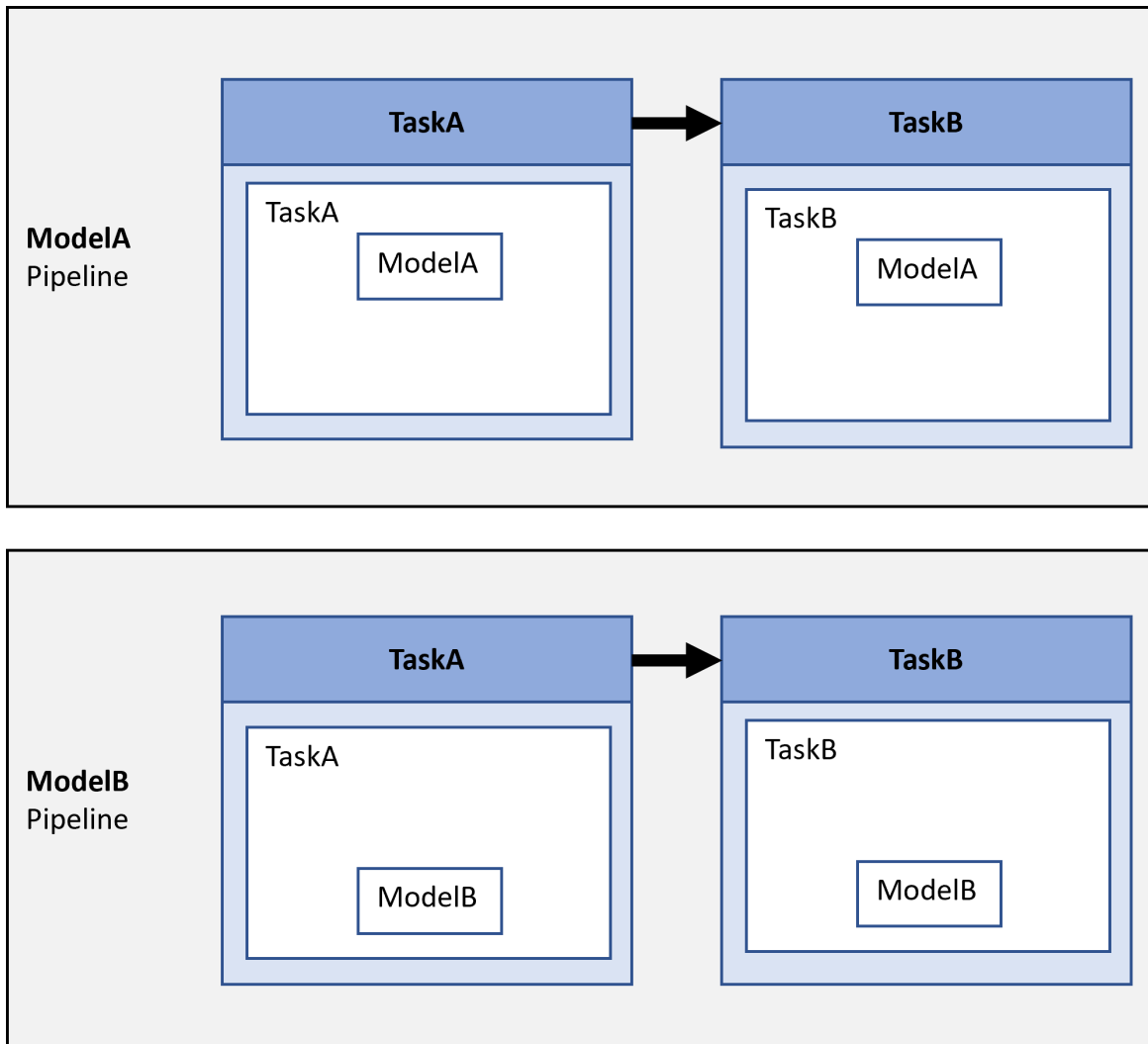
- 1 **TaskA** — Runs a task TaskA on each model in the project
- 2 **TaskB** — Runs a task TaskB on each model in the project



- `padv.pipeline.Architecture.IndependentModelPipelines`— Parallel, downstream pipelines for each model. Each pipeline independently runs the tasks associated with the model.

For example, a pipeline with parallel downstream pipelines:

- **ModelA** — Runs TaskA and TaskB on ModelA.
- **ModelB** — Runs TaskA and TaskB on ModelB.



To make sure the jobs run in parallel, make sure that you either:

- Have multiple runners available. See <https://docs.gitlab.com/ee/ci/yaml/#parallel>.
- Configure your runner to run multiple jobs concurrently by specifying the `concurrent` setting. See <https://docs.gitlab.com/runner/configuration/advanced-configuration.html>.

For more information on pipeline architectures, see the "Customize Pipeline Architecture" section in "Integrate into GitLab".

```
Example: padv.pipeline.GitLabOptions(PipelineArchitecture =
padv.pipeline.Architecture.SerialStages)
```

ForceRunAllTasks — Pipeline runs both up to date and outdated tasks

0 (false) (default) | 1 (true)

Pipeline runs both up to date and outdated tasks, specified as a numeric or logical 1 (true) or 0 (false).

The property defines the Force argument for the `runprocess` function in the generated pipeline file.

Example: `padv.pipeline.GitLabOptions(ForceRunAllTasks=true)`

Data Types: `logical`

ExitInBatchMode — Exits MATLAB if MATLAB was run with the -batch startup option

1 (true) (default) | 0 (false)

Exits MATLAB if MATLAB was run with the -batch startup option, specified as a numeric or logical 0 (false) or 1 (true).

This property defines the `ExitInBatchMode` argument for the `runprocess` function in the generated pipeline file.

Example: `padv.pipeline.GitLabOptions(ExitInBatchMode=false)`

Data Types: `logical`

RerunFailedTasks — Treats all tasks which previously failed as being outdated

0 (false) (default) | 1 (true)

Treats all tasks which previously failed as being outdated, specified as a numeric or logical 1 (true) or 0 (false).

This property defines the `RerunFailedTasks` argument for the `runprocess` function in the generated pipeline file.

Example: `padv.pipeline.GitLabOptions(RerunFailedTasks=true)`

Data Types: `logical`

RerunErroredTasks — Treats all tasks which previously generated errors as outdated

0 (false) (default) | 1 (true)

Treats all tasks which previously generated errors as outdated, specified as a numeric or logical 1 (true) or 0 (false).

This property defines the `RerunErroredTasks` argument for the `runprocess` function in the generated pipeline file.

Example: `padv.pipeline.GitLabOptions(RerunErroredTasks=true)`

Data Types: `logical`

MatlabLaunchCmd — Command to start MATLAB program

"matlab" (default) | string

Command to start MATLAB program, specified as a string.

Use this property to specify how the pipeline starts the MATLAB program. This property defines how the script in the generated pipeline file launches MATLAB.

Example: `padv.pipeline.GitLabOptions(MatlabLaunchCmd = "matlab")`

Data Types: `string`

MatlabStartupOptions — Command-line startup options for MATLAB

"-nodesktop -logfile output.log" (default) | string

Command-line startup options for MATLAB, specified as a string.

Use this property to specify the command-line startup options that the pipeline uses when starting the MATLAB program. This property defines the command-line startup options that appear next to the `-batch` option and `MatlabLaunchCmd` value in the "script" section of the generated pipeline file. The pipeline starts MATLAB with the specified startup options.

By default, the support package launches MATLAB using the `-batch` option. If you need to run MATLAB without the `-batch` option, specify the property `AddBatchStartupOption` as `false`.

Note If you run MATLAB using the `-nodisplay` option, you should set up a virtual display server before you include the following built-in tasks in your process model:

- **Generate SDD Report**
- **Generate Simulink Web View**
- **Generate Model Comparison**

For information, see "Set Up Virtual Display for No-Display Machine" in the User's Guide.

Example: `padv.pipeline.GitLabOptions(MatlabStartupOptions = "-nodesktop -logfile mylogfile.log")`

Data Types: `string`

AddBatchStartupOption — Specify whether to open MATLAB using `-batch` startup option
`1` (true) (default) | `0` (false)

Specify whether to open MATLAB using `-batch` startup option, specified as a numeric or logical `0` (false) or `1` (true).

By default, the support package launches MATLAB in CI using the `-batch` startup option.

If you need to launch MATLAB with options that are not compatible with `-batch`, specify `AddBatchStartupOption` as `false`.

Example: `padv.pipeline.GitLabOptions(AddBatchStartupOption = false)`

Data Types: `logical`

GeneratedPipelineDirectory — Specify where the generated pipeline file generates
`fullfile("derived","pipeline")` (default) | `string`

Specify where the generated pipeline file generates, specified as a string.

This property defines the directory where the generated pipeline file generates.

By default, the generated pipeline file is named `"simulink_pipeline.yml"`. To change the name of the generated pipeline file, specify `GeneratedYMLFileName`.

Example: `padv.pipeline.GitLabOptions(GeneratedPipelineDirectory = fullfile("derived","pipeline","test"))`

Data Types: `string`

GenerateJUnitForProcess — Generate JUnit-style XML reports for process
`true` or `1` (default) | `false` or `0`

Generate JUnit-style XML reports for each task in the process, specified as a numeric or logical 1 (true) or 0 (false).

JUnit reports allow you see which tests failed in CI without having to examine the job logs.

If you generate JUnit reports, GitLab shows any test failures directly in the merge request and pipeline detail view. For more information on how GitLab displays JUnit results, see the GitLab documentation: https://docs.gitlab.com/ee/ci/testing/unit_test_reports.html#view-unit-test-reports-on-gitlab.

```
Example: padv.pipeline.GitLabOptions(GenerateJUnitForProcess = false)
```

Data Types: logical

GenerateReport — Generate Process Advisor build report

true or 1 (default) | false or 0

Generate Process Advisor build report, specified as a numeric or logical 1 (true) or 0 (false).

```
Example: padv.pipeline.GitLabOptions(GenerateReport = false)
```

Data Types: logical

ReportFormat — File format for generated report

"pdf" (default) | "html" | "html-file" | "docx"

File format for the generated report, specified as one of these values:

- "pdf" — PDF file
- "html" — HTML report, packaged as a zipped file that contains the HTML file, images, style sheet, and JavaScript files of the report
- "html-file" — HTML report
- "docx" — Microsoft Word document

```
Example: padv.pipeline.GitLabOptions(ReportFormat = "html-file")
```

ReportPath — Name and path of generated report

"ProcessAdvisorReport" (default) | string array

Name and path of generated report, specified as a string array.

By default, the report generates in the current working folder with the name "ProcessAdvisorReport".

```
Example: padv.pipeline.GitLabOptions(ReportPath = "myReport")
```

Data Types: string

StopOnStageFailure — Stop running pipeline after stage fails

0 (false) (default) | 1 (true)

Stop running pipeline after stage fails, specified as a numeric or logical 0 (false) or 1 (true).

By default, the pipeline continues to run, even if a stage in the pipeline fails.

```
Example: padv.pipeline.GitLabOptions(StopOnStageFailure = true)
```

Data Types: logical

CheckOutdatedResultsAfterMerge — Check for outdated results after merge

1 (true) (default) | 0 (false)

Check for outdated results after merge, specified as a numeric or logical 1 (true) or 0 (false).

When specified as true, the pipeline checks if task results are still up-to-date after merging artifact database files from parallel jobs. Outdated results are not expected if the merge is successful. If there are outdated results, there could be an issue with the merge.

Example: false

Data Types: logical

Examples**Specify GitLab Configuration Options When Generating Pipeline Configuration File**

Create a `padv.pipeline.GitLabOptions` object and change the options. When you generate a pipeline configuration file, the file uses the specified options.

This example shows how to use the pipeline generator API. For information on how to use the pipeline generator to integrate into a GitLab CI system, see "Integrate into GitLab".

Load a project. For this example, you can load a Process Advisor example project. In the MATLAB Command Window, enter:

```
processAdvisorExampleStart
```

Create a `padv.pipeline.GitLabOptions` object for generating a GitLab pipeline configuration file. Specify a GitLab CI/CD tag of `high_memory`, specify that the function `runprocess` should not automatically exit MATLAB after the pipeline finishes running, and a single stage pipeline architecture.

```
GitLabOptions = padv.pipeline.GitLabOptions(...
Tags = "high_memory",...
ExitInBatchMode = 0,...
PipelineArchitecture = padv.pipeline.Architecture.SingleStage);
```

Generate a GitLab pipeline configuration file by using the function `padv.pipeline.generatePipeline` with the specified options.

```
padv.pipeline.generatePipeline(GitLabOptions);
```

Note Calling `padv.pipeline.generatePipeline(GitLabOptions)` is equivalent to calling `padv.pipeline.generateGitLabPipeline(GitLabOptions)`.

By default, the generated pipeline file is named `simulink_pipeline.yml` and is saved in the **derived > pipeline** folder, relative to the project root. To change the name of the generated pipeline file, specify the argument `GeneratedYMLFileName` for `padv.pipeline.GitLabOptions`. To change where the pipeline file generates, specify the argument `GeneratedPipelineDirectory`.

For information on how to use the pipeline generator to integrate into a GitLab CI system, see "Integrate into GitLab" in the User's Guide.

padv.pipeline.JenkinsOptions

Options for generating Jenkins pipeline configuration file

Description

Use the `padv.pipeline.JenkinsOptions` object to represent the desired options for generating a Jenkins pipeline configuration file. To generate a Jenkins pipeline configuration file, use `padv.pipeline.JenkinsOptions` as an input argument to the `padv.pipeline.generatePipeline` function.

Note For information on how to use the pipeline generator to integrate into a Jenkins CI system, see "Integrate into Jenkins".

Note If you run MATLAB using the `-nodisplay` option or you use a machine that does not have a display (like many CI runners and Docker containers), you should set up a virtual display server before you include the following built-in tasks in your process model:

- **Generate SDD Report**
- **Generate Simulink Web View**
- **Generate Model Comparison**

For information, see "Set Up Virtual Display for No-Display Machine" in the User's Guide.

Creation

Syntax

```
options = padv.pipeline.JenkinsOptions  
options = padv.pipeline.JenkinsOptions(Name=Value)
```

Description

`options = padv.pipeline.JenkinsOptions` returns configuration options for generating a Jenkins pipeline configuration file.

`options = padv.pipeline.JenkinsOptions(Name=Value)` sets properties using one or more name-value arguments. For example, `padv.pipeline.JenkinsOptions(AgentLabel = "high_memory")` creates an object that specifies that a generated pipeline configuration file use an agent with the label `high_memory`.

Properties

AgentLabel — Which Jenkins agent executes pipeline tasks in Jenkins environment

"any" (default) | string | string array

Which Jenkins agent executes pipeline tasks in the Jenkins environment, specified as a string or string array.

Use this property to specify the Jenkins agent that executes all stages in the pipeline. Jenkins agents are typically either a machine or a container. For more information, see the "Glossary" in the Jenkins documentation: <https://www.jenkins.io/doc/book/glossary/#agent>.

Example: `options = padv.pipeline.JenkinsOptions(AgentLabel="high_memory")`

Data Types: `string`

EnableArtifactCollection — When to collect build artifacts

"always", 1, or true (default) | "never", 0, or false | "on_success" | "on_failure"

When to collect build artifacts, specified as:

- "never", 0, or false — Never collect artifacts
- "on_success" — Only collect artifacts when the pipeline succeeds
- "on_failure" — Only collect artifacts when the pipeline fails
- "always", 1, or true — Always collect artifacts

If you choose to collect artifacts, the child pipeline contains a job, `Collect_Artifacts`, that collects the build artifacts and attaches the artifacts to the `Collect_Artifacts` job.

This property uses the Jenkins Core Plugin to add an "archiveArtifacts" step in the generated Jenkinsfile that defines the Jenkins pipeline. Install the Jenkins Core Plugin before you specify `EnableArtifactCollection`. For more information, see the Jenkins documentation for "archiveArtifacts": <https://www.jenkins.io/doc/pipeline/steps/core/#archiveartifacts-archive-the-artifacts>.

Example: `padv.pipeline.JenkinsOptions(EnableArtifactCollection="on_failure")`

Data Types: `logical` | `string`

ArtifactZipFileName — Name of ZIP file for job artifacts

"padv_artifacts.zip" (default) | `string`

Name of ZIP file for job artifacts, specified as a string.

This property specifies the file name that appears next to the "artifacts" for the "archiveArtifacts" step in the generated Jenkinsfile that defines the Jenkins pipeline.

For more information, see the Jenkins documentation for "archiveArtifacts": <https://www.jenkins.io/doc/pipeline/steps/core/#archiveartifacts-archive-the-artifacts>.

Example: `padv.pipeline.JenkinsOptions(ArtifactZipFileName = "my_job_artifacts.zip")`

Data Types: `string`

SaveArtifactsOnSuccess — Setting to only archive artifacts for successful builds

1 (true) (default) | 0 (false)

Warning This property will be removed in a future release. Use the property `EnableArtifactCollection` instead.

Setting to only archive artifacts for successful builds, specified as a numeric or logical `0` (`false`) or `1` (`true`).

Use this property to specify whether Jenkins only saves build artifacts for successful builds. This property corresponds to the argument `"onlyIfSuccessful"` for the `"artifacts"` in the `"archiveArtifacts"` step in the Jenkinsfile that defines the pipeline.

For more information, see the Jenkins documentation for `"archiveArtifacts"`: <https://www.jenkins.io/doc/pipeline/steps/core/#archiveartifacts-archive-the-artifacts>.

Example: `padv.pipeline.JenkinsOptions(SaveArtifactsOnSuccess = false)`

Data Types: `logical`

GeneratedJenkinsFileName — File name of generated Jenkins pipeline file

`"simulink_pipeline"` (default) | `string`

File name of generated Jenkins pipeline file, specified as a string.

By default, the generated pipeline generates into the subfolder **derived > pipeline**, relative to the project root. To change where the pipeline file generates, specify `GeneratedPipelineDirectory`.

Example: `padv.pipeline.JenkinsOptions(GeneratedJenkinsFileName = "padv_generated_pipeline_file")`

Data Types: `string`

UseMatlabPlugin — Specify whether Jenkins uses MATLAB Plugin to launch MATLAB

`1` (`true`) (default) | `0` (`false`)

Specify whether Jenkins uses MATLAB Plugin to launch MATLAB, specified as a numeric or logical `0` (`false`) or `1` (`true`).

If the property `UseMatlabPlugin` is `true`, Jenkins uses the `"runMATLABCommand"` step to launch MATLAB and the pipeline generator ignores the properties `MatlabLaunchCmd` and `MatlabStartupOptions`. For more information, see the Jenkins documentation for `"runMATLABCommand"`: <https://www.jenkins.io/doc/pipeline/steps/matlab/#runmatlabcommand-run-matlab-commands-scripts-or-functions>

If the property `UseMatlabPlugin` is `false`, Jenkins uses the specified `ShellEnvironment` to launch MATLAB and uses the options specified by the properties `MatlabLaunchCmd` and `MatlabStartupOptions`.

Using the MATLAB Plugin for Jenkins is recommended. For more information, see <https://plugins.jenkins.io/matlab/>.

Example: `padv.pipeline.JenkinsOptions(UseMatlabPlugin = false)`

Data Types: `logical`

ShellEnvironment — Shell environment Jenkins uses to launch MATLAB

`"` (default) | `string`

Shell environment Jenkins uses to launch MATLAB, specified as one of these values:

- `"bat"` — Windows® batch script
- `"sh"` — Shell script

- "pwsh" — PowerShell Core script
- "powershell" — Windows PowerShell script
- "" — Automatically use "bat" or "sh" based on the platform where pipeline generation runs

If the property UseMatlabPlugin is true, Jenkins uses the "runMATLABCommand" step to launch MATLAB and the pipeline generator ignores the properties MatlabLaunchCmd and MatlabStartupOptions. For more information, see the Jenkins documentation for "runMATLABCommand": <https://www.jenkins.io/doc/pipeline/steps/matlab/#runmatlabcommand-run-matlab-commands-scripts-or-functions>

If the property UseMatlabPlugin is false, Jenkins uses the specified ShellEnvironment to launch MATLAB and uses the options specified by the properties MatlabLaunchCmd and MatlabStartupOptions.

Example: `padv.pipeline.JenkinsOptions(UseMatlabPlugin = false, ShellEnvironment = "bat")`

Data Types: string

PipelineArchitecture — Number of stages and grouping of tasks in CI pipeline

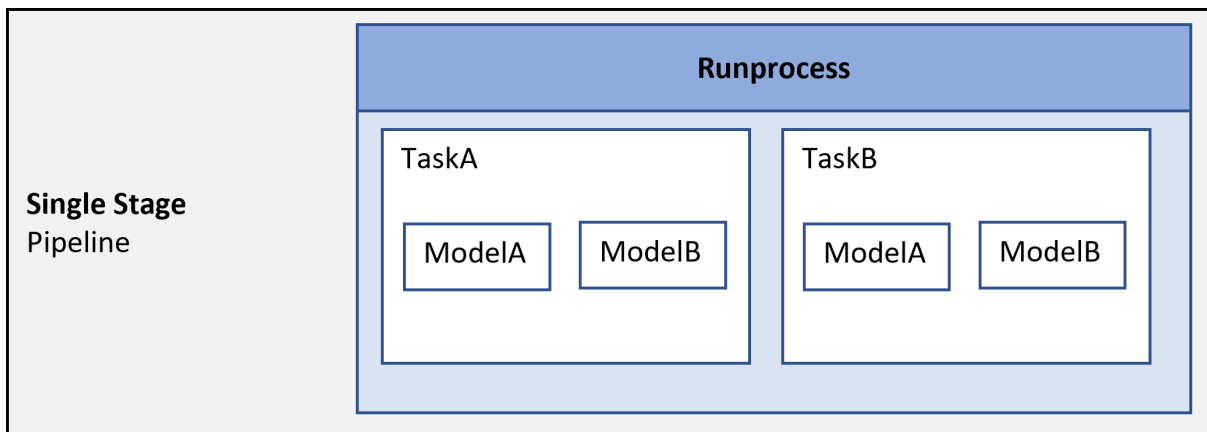
`padv.pipeline.Architecture.SingleStage` (default) |
`padv.pipeline.Architecture.SerialStages` |
`padv.pipeline.Architecture.SerialStagesGroupPerTask`

Number of stages and grouping of tasks in CI pipeline, specified as either:

- `padv.pipeline.Architecture.SingleStage` — Single stage runs all tasks

For example, a pipeline with one stage that runs each of the tasks in the process:

1 Runprocess

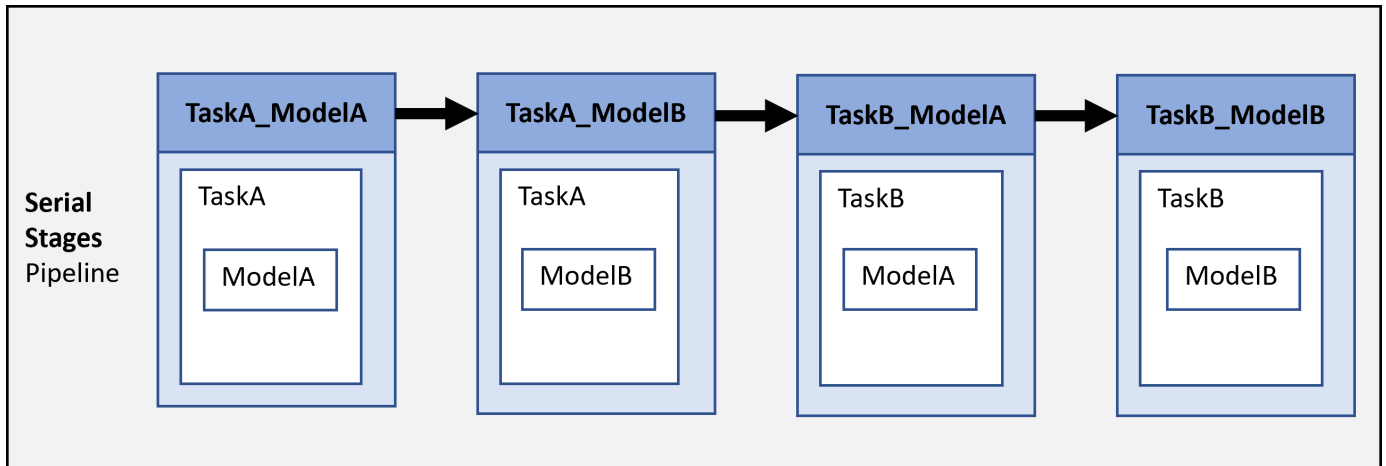


- `padv.pipeline.Architecture.SerialStages` — One stage for each task iteration

For example, a pipeline with four stages:

- 1 **TaskA_ModelA** — Runs a task TaskA on the model ModelA
- 2 **TaskA_ModelB** — Runs a task TaskA on the model ModelB
- 3 **TaskB_ModelA** — Runs a task TaskB on the model ModelA

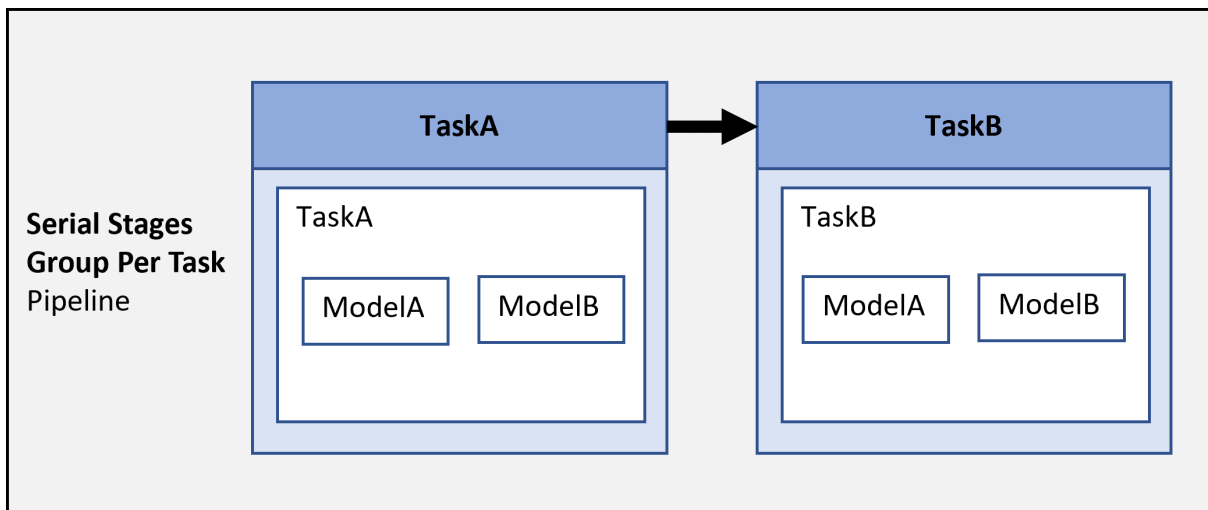
4 **TaskB_ModelB** — Runs a task TaskB on the model ModelB



- `padv.pipeline.Architecture.SerialStagesGroupPerTask` — One stage for each type of task

For example, a pipeline with two stages:

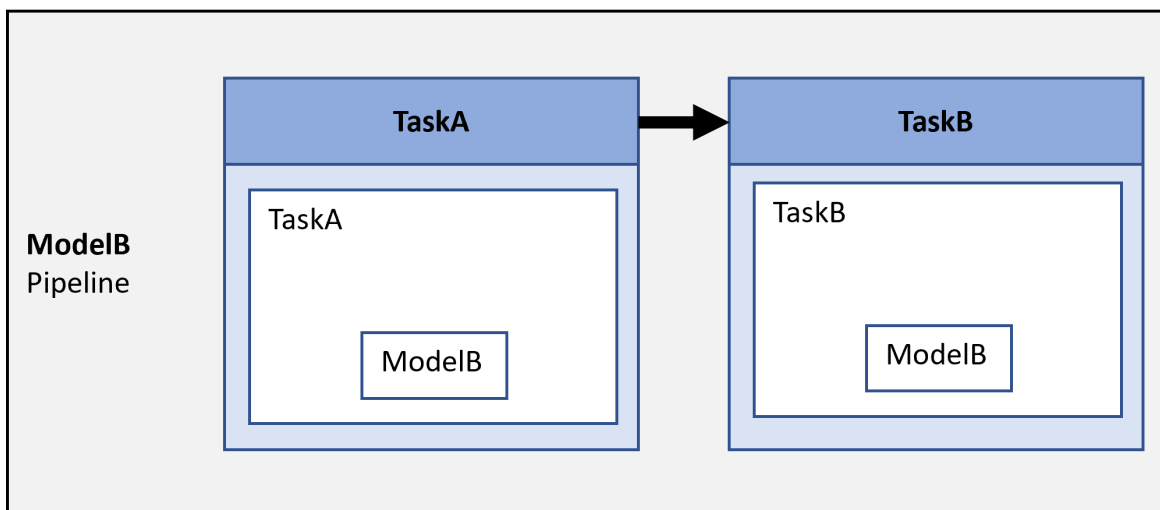
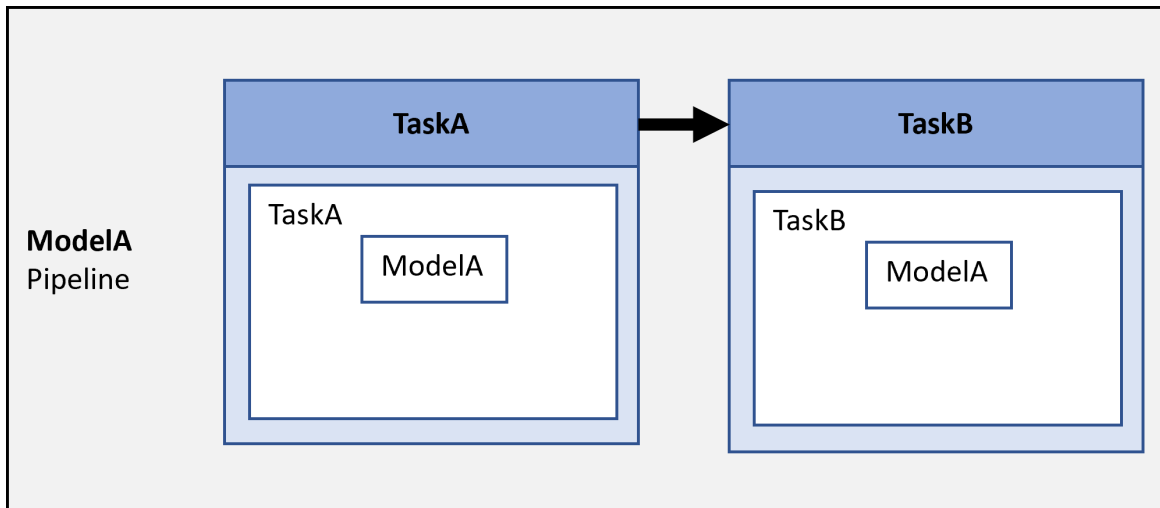
- 1 **TaskA** — Runs a task TaskA on each model in the project
- 2 **TaskB** — Runs a task TaskB on each model in the project



- `padv.pipeline.Architecture.IndependentModelPipelines` — Parallel, downstream pipelines for each model. Each pipeline independently runs the tasks associated with the model.

For example, a pipeline with parallel downstream pipelines:

- **ModelA** — Runs TaskA and TaskB on ModelA.
- **ModelB** — Runs TaskA and TaskB on ModelB.



For more information on pipeline architectures, see the "Customize Pipeline Architecture" section in "Integrate into Jenkins".

Example: `padv.pipeline.JenkinsOptions(PipelineArchitecture = padv.pipeline.Architecture.SerialStages)`

ForceRunAllTasks — Pipeline runs both up to date and outdated tasks

0 (false) (default) | 1 (true)

Pipeline runs both up to date and outdated tasks, specified as a numeric or logical 1 (true) or 0 (false).

The property defines the Force argument for the runprocess function in the generated pipeline file.

Example: `padv.pipeline.JenkinsOptions(ForceRunAllTasks=true)`

Data Types: logical

ExitInBatchMode — Exits MATLAB if MATLAB was run with the -batch startup option

1 (true) (default) | 0 (false)

Exits MATLAB if MATLAB was run with the `-batch` startup option, specified as a numeric or logical `0` (`false`) or `1` (`true`).

This property defines the `ExitInBatchMode` argument for the `runprocess` function in the generated pipeline file.

Example: `padv.pipeline.JenkinsOptions(ExitInBatchMode=false)`

Data Types: `logical`

RerunFailedTasks — Treats all tasks which previously failed as being outdated

`0` (`false`) (default) | `1` (`true`)

Treats all tasks which previously failed as being outdated, specified as a numeric or logical `1` (`true`) or `0` (`false`).

This property defines the `RerunFailedTasks` argument for the `runprocess` function in the generated pipeline file.

Example: `padv.pipeline.JenkinsOptions(RerunFailedTasks=true)`

Data Types: `logical`

RerunErroredTasks — Treats all tasks which previously generated errors as outdated

`0` (`false`) (default) | `1` (`true`)

Treats all tasks which previously generated errors as outdated, specified as a numeric or logical `1` (`true`) or `0` (`false`).

This property defines the `RerunErroredTasks` argument for the `runprocess` function in the generated pipeline file.

Example: `padv.pipeline.JenkinsOptions(RerunErroredTasks=true)`

Data Types: `logical`

MatlabLaunchCmd — Command to start MATLAB program

`"matlab"` (default) | `string`

Command to start MATLAB program, specified as a string.

Use this property to specify how the pipeline starts the MATLAB program. This property defines how the generated pipeline file launches MATLAB.

Example: `padv.pipeline.JenkinsOptions(MatlabLaunchCmd = "matlab")`

Data Types: `string`

MatlabStartupOptions — Command-line startup options for MATLAB

`"-nodesktop -logfile output.log"` (default) | `string`

Command-line startup options for MATLAB, specified as a string.

Use this property to specify the command-line startup options that the pipeline uses when starting the MATLAB program. This property defines the command-line startup options that appear next to the `-batch` option and `MatlabLaunchCmd` value in the `"script"` section of the generated pipeline file. The pipeline starts MATLAB with the specified startup options.

By default, the support package launches MATLAB using the `-batch` option. If you need to run MATLAB without the `-batch` option, specify the property `AddBatchStartupOption` as `false`.

Note If you run MATLAB using the `-nodisplay` option, you should set up a virtual display server before you include the following built-in tasks in your process model:

- **Generate SDD Report**
- **Generate Simulink Web View**
- **Generate Model Comparison**

For information, see "Set Up Virtual Display for No-Display Machine" in the User's Guide.

Example: `padv.pipeline.JenkinsOptions(MatlabStartupOptions = "-nodesktop -logfile mylogfile.log")`

Data Types: `string`

AddBatchStartupOption — Specify whether to open MATLAB using `-batch` startup option
`1 (true) (default) | 0 (false)`

Specify whether to open MATLAB using `-batch` startup option, specified as a numeric or logical `0 (false)` or `1 (true)`.

By default, the support package launches MATLAB in CI using the `-batch` startup option.

If you need to launch MATLAB with options that are not compatible with `-batch`, specify `AddBatchStartupOption` as `false`.

Example: `padv.pipeline.JenkinsOptions(AddBatchStartupOption = false)`

Data Types: `logical`

GeneratedPipelineDirectory — Specify where the generated pipeline file generates
`fullfile("derived","pipeline") (default) | string`

Specify where the generated pipeline file generates, specified as a string.

This property defines the directory where the generated pipeline file generates.

By default, the generated pipeline file is named `"simulink_pipeline"`. To change the name of the generated pipeline file, specify `GeneratedJenkinsFileName`.

Example: `padv.pipeline.JenkinsOptions(GeneratedPipelineDirectory = fullfile("derived","pipeline","test"))`

Data Types: `string`

GenerateJUnitForProcess — Generate JUnit-style XML reports for process
`true or 1 (default) | false or 0`

Generate JUnit-style XML reports for each task in the process, specified as a numeric or logical `1 (true)` or `0 (false)`.

JUnit reports allow you see which tests failed in CI without having to examine the job logs.

If you generate JUnit reports, Jenkins can show test failures and trends directly in the user interface. For more information on how Jenkins displays JUnit results, see the Jenkins documentation: <https://plugins.jenkins.io/junit/>.

Note You must have the JUnit plugin installed on your Jenkins controller to see JUnit results. For information, see <https://plugins.jenkins.io/junit/>.

Example: `padv.pipeline.JenkinsOptions(GenerateJUnitForProcess = false)`

Data Types: `logical`

GenerateReport — Generate Process Advisor build report

`true` or `1` (default) | `false` or `0`

Generate Process Advisor build report, specified as a numeric or logical `1` (`true`) or `0` (`false`).

Example: `padv.pipeline.JenkinsOptions(GenerateReport = false)`

Data Types: `logical`

ReportFormat — File format for generated report

`"pdf"` (default) | `"html"` | `"html-file"` | `"docx"`

File format for the generated report, specified as one of these values:

- `"pdf"` — PDF file
- `"html"` — HTML report, packaged as a zipped file that contains the HTML file, images, style sheet, and JavaScript files of the report
- `"html-file"` — HTML report
- `"docx"` — Microsoft Word document

Example: `padv.pipeline.JenkinsOptions(ReportFormat = "html-file")`

ReportPath — Name and path of generated report

`"ProcessAdvisorReport"` (default) | string array

Name and path of generated report, specified as a string array.

By default, the report generates in the current working folder with the name `"ProcessAdvisorReport"`.

Example: `padv.pipeline.JenkinsOptions(ReportFormat = "myReport")`

Data Types: `string`

StopOnStageFailure — Stop running pipeline after stage fails

`0` (`false`) (default) | `1` (`true`)

Stop running pipeline after stage fails, specified as a numeric or logical `0` (`false`) or `1` (`true`).

By default, the pipeline continues to run, even if a stage in the pipeline fails.

Example: `padv.pipeline.JenkinsOptions(StopOnStageFailure = true)`

Data Types: `logical`

CheckOutdatedResultsAfterMerge — Check for outdated results after merge

1 (true) (default) | 0 (false)

Check for outdated results after merge, specified as a numeric or logical 1 (true) or 0 (false).

When specified as true, the pipeline checks if task results are still up-to-date after merging artifact database files from parallel jobs. Outdated results are not expected if the merge is successful. If there are outdated results, there could be an issue with the merge.

Example: false

Data Types: logical

Examples**Specify Jenkins Configuration Options When Generating Pipeline Configuration File**

Create a `padv.pipeline.JenkinsOptions` object and change the options. When you generate a pipeline configuration file, the file uses the specified options.

This example shows how to use the pipeline generator API. For information on how to use the pipeline generator to integrate into a Jenkins CI system, see "Integrate into Jenkins".

Load a project. For this example, you can load a Process Advisor example project. In the MATLAB Command Window, enter:

```
processAdvisorExampleStart
```

Create a `padv.pipeline.JenkinsOptions` object for generating a Jenkins pipeline configuration file. Specify a Jenkins agent label of `high_memory`, specify that the function `runprocess` should not automatically exit MATLAB after the pipeline finishes running, and a single stage pipeline architecture.

```
JenkinsOptions = padv.pipeline.JenkinsOptions(...
AgentLabel = "high_memory",...
ExitInBatchMode = 0,...
PipelineArchitecture = padv.pipeline.Architecture.SingleStage);
```

Generate a Jenkins pipeline configuration file by using the function `padv.pipeline.generatePipeline` with the specified options.

```
padv.pipeline.generatePipeline(JenkinsOptions);
```

Note Calling `padv.pipeline.generatePipeline(JenkinsOptions)` is equivalent to calling `padv.pipeline.generateJenkinsPipeline(JenkinsOptions)`.

By default, the generated pipeline file is named `simulink_pipeline` and is saved in the **derived > pipeline** folder, relative to the project root. To change the name of the generated pipeline file, specify the argument `GeneratedJenkinsFileName` for `padv.pipeline.JenkinsOptions`. To change where the pipeline file generates, specify the argument `GeneratedPipelineDirectory`.

For information on how to use the pipeline generator to integrate into a Jenkins CI system, see "Integrate into Jenkins" in the User's Guide.

Report Generator API

After you run your tasks, you can use the report generator to create a report with the most recent task results. The report summarizes the task statuses, task results, and other information about the task execution.

For example, if you run the tasks in the default MBD pipeline, the report provides an overview of the:

- Model Advisor analysis, including the number of passing, warning, and failing checks
- Test results, organized by iteration
- Generated code files
- Coding standards checks

For an example, see "Prequalify Changes Before Submitting to Source Control" in the User's Guide PDF.

Functions

Create and Access Process Model

Function	Description
generateReport	Generate report with recent task results

generateReport

Generate report with recent task results

Syntax

```
generateReport(reportSettings)
generateReport( ____,Name,Value)
```

Description

`generateReport(reportSettings)` generates a report with the most recent task results.

After you run tasks using the Process Advisor app or `runprocess` function, you can use the `generateReport` function to generate a report of the task results.

Alternatively, you can use `runprocess` with the `GenerateReport` name-value argument specified as `true`: `runprocess(GenerateReport = true)`.

`generateReport(____,Name,Value)` specifies options using one or more name-value arguments.

For example, to generate a report in HTML format:

```
generateReport(padv.ProcessAdvisorReportGenerator(Format="html-file"))
```

Examples


Generate Report with Task Results

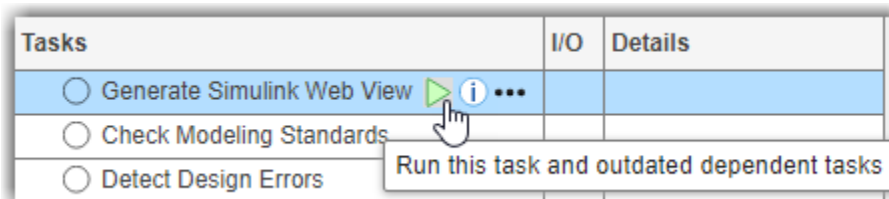
Run a task and generate a report with the task results.

Open the Process Advisor example project.

```
processAdvisorExampleStart
```

This command creates a copy of the Process Advisor example project and opens Process Advisor on the model `AHRS_Voter`.

Run a task. For this example, in Process Advisor, point to the task **Generate Simulink Web View** and click the run button .



Use the `generateReport` function to generate an HTML report with the task results.


```
generateReport(padv.ProcessAdvisorReportGenerator(Format="html-file"))
```

The report, `ProcessAdvisorReport.html`, generates in the current working folder.

Open and inspect the report. The report shows a summary of the task status, results, inputs, and outputs.

Input Arguments

reportSettings — Report generation settings

`padv.ProcessAdvisorReportGenerator` object

Report generation settings, specified as a `padv.ProcessAdvisorReportGenerator` object.

Example: `generateReport(padv.ProcessAdvisorReportGenerator)`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `generateReport(padv.ProcessAdvisorReportGenerator(Format="html-file"))`

Format — File format for generated report

"pdf" (default) | "html" | "html-file" | "docx"

File format for the generated report, specified as one of these values:

- "pdf" — PDF file
- "html" — HTML report, packaged as a zipped file that contains the HTML file, images, style sheet, and JavaScript files of the report
- "html-file" — HTML report
- "docx" — Microsoft Word document

Example: `generateReport(padv.ProcessAdvisorReportGenerator(Format="html-file"))`

OutputPath — Name and path of generated report

"ProcessAdvisorReport" (default) | string array

Name and path of generated report, specified as a string array.

By default, the report generates in the current working folder with the name "ProcessAdvisorReport".

Example: `generateReport(padv.ProcessAdvisorReportGenerator(OutputPath = "tools/myReport"))`

Data Types: string

Tips

- If you want to run tasks and generate a report in batch mode, you need to specify the `runprocess` argument `ExitInBatchMode` as `false` and use the `exitCode` returned by `runprocess` to exit:

```
[buildResult, exitCode] = runprocess(ExitInBatchMode=false);  
rptObj = padv.ProcessAdvisorReportGenerator();  
generateReport(rptObj);  
exit(exitCode);
```

Otherwise, the function `runprocess` automatically exits MATLAB before the report can generate.

Alternative Functionality

Alternatively, you can use `runprocess` with the `GenerateReport` name-value argument specified as `true`: `runprocess(GenerateReport = true)`.

Utilities

Classes

Specify Artifact Address for `padv.Artifact` Object

Class	Description
<code>padv.util.ArtifactAddress</code>	Address for artifact in project

Functions

Close Models Loaded by Task

Function	Description
<code>padv.util.closeModelsLoadedByTask</code>	Close models loaded by task

Get Current Project and Referenced Projects

Function	Description
<code>padv.util.getCurrentProject</code>	Get current project and persist project instance Note This function can be faster than the <code>currentProject</code> function because it creates a persistent variable for the current project instance.
<code>padv.util.getProjectReferences</code>	Get list of project references

Get Information From Artifact

Function	Description
<code>padv.util.getModelName</code>	Find name of model that contains artifact
<code>padv.util.getTestCaseID</code>	Find ID for test case that contains artifact

If your team generates code in parallel by generating an external code cache (see `GenerateExternalCodeCache` property for built-in task `padv.builtin.task.GenerateCode`), downstream tasks that depend on the generated code need to unpack the generated code target before running the task action. Built-in tasks like `padv.builtin.task.AnalyzeModelCode` unpack by using the utility function `padv.util.unpackExternalCodeCache`.

Reanalyze Project From Scratch

Function	Description
<code>padv.util.forceReanalyzeProject</code>	Reanalyze project and log analysis events
	Note You should only use the function <code>padv.util.forceReanalyzeProject</code> if there are unexpected project analysis issues. For general task and result cleanup, use <code>runprocess</code> instead.

Refresh Process Model

Function	Description
<code>padv.util.refreshProcessModel</code>	Refresh process model data

Save and Merge Artifact Database Files

Function	Description
<code>padv.util.mergeArtifactDatabases</code>	Merge artifact database files
<code>padv.util.saveArtifactDatabase</code>	Save copy of artifact database file

Unpack Generated Code Target

Function	Description
<code>padv.util.unpackExternalCodeCache</code>	Unpack code generation target from Simulink cache files

Process Advisor and the build system are able to detect changes to project files and identify outdated tasks by using the information in the artifact database file, located in `derived > artifacts.dmr`.

When your team works on multiple machines or runs tasks in parallel, you generate different versions of artifact database file. To create an artifact database file that includes the latest changes, you can save a base artifact database file and merge artifact database files by using the functions `padv.util.saveArtifactDatabase` and `padv.util.mergeArtifactDatabases`.

padv.util.ArtifactAddress

Address for artifact in project

Description

Use the `padv.util.ArtifactAddress` object to represent the address of an artifact in your project.

Creation

Syntax

```
addressObj = padv.util.ArtifactAddress(filePath)
addressObj = padv.util.ArtifactAddress( ____,Name=Value)
```

Description

`addressObj = padv.util.ArtifactAddress(filePath)` creates an artifact address by using the file path specified by `filePath`. You can access information inside the artifact address object by using the object functions listed below.

`addressObj = padv.util.ArtifactAddress(____,Name=Value)` creates an artifact address using the settings specified by one or more name-value arguments. For example, to create an artifact address that specifies the name of the project that contains the artifact, specify `OwningProjectName=projectName`.

Input Arguments

filePath — File path

string array

File path, specified as a string array.

Example: `padv.util.ArtifactAddress(fullfile("tools","sampleChecks.json"))`

Data Types: string

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, . . . ,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `padv.util.ArtifactAddress(filePath,OwningProjectName=projectName)`

OwningProjectName — Project that contains artifact

string array

Project that contains the artifact, specified as a string array.

You can retrieve the owning project name of an artifact address object by using the `getOwningProject` object function.

Example: "ProcessAdvisorExample"

Data Types: string

Track — Setting for tracking changes to artifact

true or 1 | false or 0

Setting for tracking changes to the artifact, specified as a numeric or logical 1 (true) or 0 (false).

For more information, see "Turn Off Change Tracking for Input Artifacts".

Example: false

Data Types: logical

Object Functions

Function	Description
<code>getFileAddress</code>	Get address of file on disk. <code>getFileAddress(addressObj)</code>
<code>getKey</code>	Get unique address of artifact. <code>getKey(addressObj)</code>
<code>getOwningProject</code>	Get name of project that contains the artifact. <code>getOwningProject(addressObj)</code>
<code>isFileArtifact</code>	Determine if input is file. <code>isFileArtifact(addressObj)</code>
<code>isSubFileArtifact</code>	Determine if input is subfile. A subfile is a part of a larger file. For example, a subsystem is a subfile of a model file. <code>isSubFileArtifact(addressObj)</code>

Examples

Specify Address for Artifact

Create artifact address for file in project.

```
addressObj = padv.util.ArtifactAddress(...
fullfile("tools", "sampleChecks.json"));
```

Use artifact address to create `padv.Artifact` object.

```
paArtifact = padv.Artifact("other_file",addressObj)
```

Specify Which Project Contains Artifact

Specify the name of the project that contains the artifact.

```
projectName = "My Reference Project";
```

Specify that the project contains the artifact.

```
addressObj = padv.util.ArtifactAddress(fullfile("tools","sampleChecks.json"),...  
OwningProjectName=projectName)
```

You can view which project contains the artifact by using the `getOwningProject` function.

```
getOwningProject(addressObj)
```

```
ans =
```

```
    "My Reference Project"
```

padv.util.closeModelsLoadedByTask

Close models loaded by task

Syntax

```
padv.util.closeModelsLoadedByTask(PreviouslyLoadedModels = modelList)
```

Description

`padv.util.closeModelsLoadedByTask(PreviouslyLoadedModels = modelList)` closes models that were loaded by the current task. The function determines which models the task loaded by comparing the current list of loaded models to a list of previously loaded models, `modelList`. The function uses `close_system(model, 0)` to close the models without saving.

Use this function inside the run function of a custom task to close all models loaded by the task. Note that the function does not close models that are open in the Simulink Editor.

Examples

Close Models Loaded by Task

Find which models were already loaded and then use the function

`padv.util.closeModelsLoadedByTask` to close only models loaded by the current task.

Inside the run function for your custom task, use the function `get_param` to find and save a list of the previously loaded models. Then, after your task performs its action and specifies the task results, close the models loaded by the task. For example, the run function in your custom task might look like:

```
function taskResult=run(obj, input)
    % Before the task loads models, save a list of the models that are already loaded.
    loadedModels = get_param(Simulink.allBlockDiagrams(), 'Name');

    % <load models for this task>
    % <specify task results>

    % Close models that were loaded by this task.
    padv.util.closeModelsLoadedByTask(PreviouslyLoadedModels=loadedModels);
end
```

Input Arguments

modelList — List of previously loaded models

List of previously loaded models, specified as an array of model names.

You can use the function `get_param` to find the currently loaded models:

```
loadedModels = get_param(Simulink.allBlockDiagrams(), 'Name');
```


Example: {'modelA'; 'modelB'; 'modelC'}

padv.util.forceReanalyzeProject

Reanalyze project and log analysis events

Syntax

```
padv.util.forceReanalyzeProject()
```

Description

`padv.util.forceReanalyzeProject()` forces a reanalysis of the current project by creating backups of the existing artifact database (`artifacts.dmr`), clearing the existing project analysis, and reanalyzing the project. The function also logs project analysis events, which can help with troubleshooting persistent project analysis issues. Note that when you run the function, the function closes and reopens the project.

The function creates backup files and detailed logs in the `derived` folder in the project and creates a ZIP file containing these artifacts for further analysis. The files include:

- `artifacts_no_update.dmr.bak` — Backup of the `artifacts.dmr` file before update
- `artifacts_update.dmr.bak` — Backup of the `artifacts.dmr` file after update
- `artifacts_new.dmr.bak` — Backup of the `artifacts.dmr` file after reanalysis
- `dt_Event_Log.txt` — Event log file
- `detailed_logs.txt` — Detailed log file
- `logs.zip` — ZIP file containing the above files

Note You should only use the function `padv.util.forceReanalyzeProject` if there are unexpected project analysis issues. When you clear the existing project analysis file, you might permanently lose important information, including the UUIDs that the digital thread assigned to artifacts in your project. Reanalyzing a project might take some time to complete. The `artifacts.dmr` file might be used by other project users and if you use other tools that use the digital thread, you might need to re-run the metrics in those tools.

For general task and result cleanup, use `runprocess` instead. The `runprocess` function has name-value arguments, `Clean` and `DeleteOutputs`, that you can use to clean task results and delete task outputs. For information, see `runprocess` in this PDF.

padv.util.getCurrentProject

Get current project and persist project instance

Syntax

```
cp = padv.util.getCurrentProject()
```

Description

`cp = padv.util.getCurrentProject()` gets the currently open project, and returns a project object, `cp`. You can use this function to get the current project in your code, for example, in custom queries. This function can be faster than the `currentProject` function because `cp` is a persistent variable.

Examples

Get Current Project

Get the current project, represented by a `matlab.project.Project` object.

Open the Process Advisor example project.

```
processAdvisorExampleStart
```

Get the current project.

```
cp = padv.util.getCurrentProject()
```

Output Arguments

cp – Current project

`matlab.project.Project`

Current project, returned as a `matlab.project.Project` object. `cp` is a persistent variable that can remain in memory between calls to the function.

If you do not have a project open, then the function returns an empty array.

padv.util.getModelName

Namespace: padv.util

Find name of model that contains artifact

Syntax

```
modelName = padv.util.getModelName(artifact)
```

Description

`modelName = padv.util.getModelName(artifact)` returns the name of the model that contains artifact.

Input Arguments

artifact — Artifact information

`padv.Artifact` object

Artifact information, specified as a `padv.Artifact` object.

You can create a `padv.Artifact` object either by:

- Running a built-in query. When you run a built-in query, the query returns either a `padv.Artifact` object or an array of `padv.Artifact` objects.
- Using the `padv.Artifact` class.

Example:

```
padv.Artifact("sl_model_file", padv.util.ArtifactAddress(fullfile("02_Models",  
"AHRS_Voter", "specification", "AHRS_Voter.slx"))))
```

Output Arguments

modelName — Name of model that contains artifact

string

Name of model that contains artifact, returned as a string.

padv.util.getProjectReferences

Get list of project references

Syntax

```
prjReferences = padv.util.getProjectReferences()  
prjReferences = padv.util.getProjectReferences("reset")
```

Description

`prjReferences = padv.util.getProjectReferences()` gets a list of the project references for the current project. The function caches the list.

`prjReferences = padv.util.getProjectReferences("reset")` resets the cached list of project references.

Examples

Get List of Project References

Get a list of the project references for the current project.

Open the Process Advisor example for project references.

```
processAdvisorProjectReferenceExampleStart
```

Get the list of project references for the current project.

```
prjReferences = padv.util.getProjectReferences()
```

Output Arguments

prjReferences — Project references

ProjectReference object | array of ProjectReference objects

Project references, returned as a ProjectReference object or an array of ProjectReference objects.

padv.util.getTestCaseID

Find ID for test case that contains artifact

Syntax

```
testCaseID = padv.util.getTestCaseID(artifact)
```

Description

`testCaseID = padv.util.getTestCaseID(artifact)` returns the ID for the test case that contains `artifact`.

Examples

Find Test Case ID Associated with Artifact

Find the test case ID for a test case by using `padv.util.getTestCaseID`.

Open the Process Advisor example project. In the MATLAB Command Window, enter:

```
processAdvisorExampleStart
```

Create a query that can find the test cases in the project. Since test cases are part of a larger test file, test cases are subfile artifacts and you must specify `FilterSubFileArtifacts` as `false` to stop the query from filtering out the test cases.

```
q = padv.builtin.query.FindArtifacts(ArtifactType = "sl_test_case", ...  
FilterSubFileArtifacts = false);
```

Find the test cases in the project by running the query. The query returns the as an array of `padv.Artifact` objects.

```
testCaseArtifacts = run(q);
```

Find the test case ID for one of the test cases returned by the query.

```
id = padv.util.getTestCaseID(testCaseArtifacts(1))
```

Input Arguments

artifact — Artifact information

`padv.Artifact` object

Artifact information, specified as a `padv.Artifact` object.

You can create a `padv.Artifact` object either by:

- Running a built-in query. When you run a built-in query, the query returns either a `padv.Artifact` object or an array of `padv.Artifact` objects.

- Using the `padv.Artifact` class.

Example:

```
padv.Artifact("sl_model_file",padv.util.ArtifactAddress(fullfile("02_Models",  
"AHRS_Voter","specification","AHRS_Voter.slx"))))
```

Output Arguments

testCaseID — ID for test case that contains artifact

string

ID for the test case that contains the artifact, returned as a string.

Version History

padv.util.mergeArtifactDatabases

Merge artifact database files

Syntax

```
padv.util.mergeArtifactDatabases(Base = baseFile, Branches = filesToMerge,
Merged = mergedFile)
padv.util.mergeArtifactDatabases( ____, CheckOutdatedResults = false)
```

Description

`padv.util.mergeArtifactDatabases(Base = baseFile, Branches = filesToMerge, Merged = mergedFile)` merges the artifact database files, `filesToMerge`, with the common ancestor artifact database file, `baseFile`, to create a merged artifact database file `mergedFile`.

You can use this function to merge artifact database files from different feature branches or CI pipeline jobs. The function requires an open project.

`padv.util.mergeArtifactDatabases(____, CheckOutdatedResults = false)` merges without validating that task results are still up-to-date after the merge. Outdated results are not expected if the merge is successful. If there are outdated results, there could be an issue with the merge. By default, `CheckOutdatedResults` is `true`.

Note Only supported in R2023b Update 5 and later releases.

Examples

Merge Project Analysis from Different Feature Branches

Process Advisor and the build system are able to detect changes to project files and identify outdated tasks by using the information in the artifact database file `artifacts.dmr`. When your team works on a project with multiple feature branches, you might need to merge different versions of `artifacts.dmr` into a single file that contains the latest project analysis. To create the file, you need to save a copy of the base artifact database file and then merge the `artifacts.dmr` files from each branch.

When your team members clone the project from source control, have them download the latest derived files, including the `artifacts.dmr` file that contains the latest analysis of the project. By default, digital thread stores the artifact database file inside the `derived` folder in the project root.

You can use a database or repository management tool to handle derived files effectively.

To resolve conflicts between the artifact database files from the different feature branches, you need to create a base artifact database file. Use the most recent `artifacts.dmr` file from the derived files as the base because that file represents the latest shared state of project analysis across the feature branches.

Create a copy of the artifact database file inside the `derived` folder and name the file `base.dmr`.


```
padv.util.saveArtifactDatabase(fullfile("derived", "base.dmr"))
```

As each team member works on their separate branches, the digital thread updates the `artifacts.dmr` file in their copy of the project to reflect their changes.

After a team member makes the changes on their branch, use the function `padv.util.saveArtifactDatabase` in each branch to save a copy of the artifact database file from that branch. For example, you might have artifact database files like `featureA.dmr` and `featureB.dmr`.

Merge the artifact database files into a new `artifacts.dmr` file by using the function `padv.util.mergeArtifactDatabases`. The base artifact database file is `base.dmr` and the artifact database files from the branches are `featureA.dmr` and `featureB.dmr`.

```
padv.util.mergeArtifactDatabases(...
Base = fullfile("derived", "base.dmr"), ...
Branches = [fullfile("derived", "featureA.dmr"), fullfile("derived", "featureB.dmr")], ...
Merged = fullfile("derived", "artifacts.dmr"))
```

This section describes how to merge artifact database files from separate feature branches, but you can also use these functions to merge artifact database files from jobs in CI and tasks that you run in parallel. Starting in R2023b Update 5, GitHub and Jenkins pipelines that you generate by using the function `padv.pipeline.generatePipeline` automatically merge artifact database files.

Input Arguments

baseFile — Path and name of base artifact database file

string

Path and name of base artifact database file, specified as a string.

The base artifact database file is the common ancestor of the artifact database files that you want to merge. The path must be relative to the project root or an absolute path.

To create a common ancestor, you can save a copy of an artifact database file by using the function `padv.util.saveArtifactDatabase`.

Example: `fullfile("derived", "base.dmr")`

Data Types: string

filesToMerge — Paths and names of artifact database files to merge

string array

Paths and names of artifact database files that you want to merge, specified as a string array.

Example: `[fullfile("derived", "modelA.dmr"), fullfile("derived", "modelB.dmr")]`

Data Types: string

mergedFile — Path and name of merged artifact database file

string

Path and name of merged artifact database file, specified as a string.

The path must be relative to the project root or an absolute path.

Example: `fullfile("derived", "artifacts.dmr")`

Data Types: `string`

Version History

Introduced in R2023b

padv.util.refreshProcessModel

Refresh process model data

Syntax

```
padv.util.refreshProcessModel()
```

Description

`padv.util.refreshProcessModel()` refreshes the process model. Use this function if you need to manually refresh the process model data.

Examples

Refresh Process Model

Make a change to a project and programmatically refresh the process model data.

Open the example project for Process Advisor.

```
processAdvisorExampleStart
```

The `AHRS_Voter` model opens.

Make a change to the `AHRS_Voter` model and re-save the model.

The warning banner in Process Advisor shows that the process model data needs to be refreshed.

Programmatically refresh the process model data by using `padv.util.refreshProcessModel`.

```
padv.util.refreshProcessModel
```

padv.util.saveArtifactDatabase

Save copy of artifact database file

Syntax

```
padv.util.saveArtifactDatabase(destination)
```

Description

`padv.util.saveArtifactDatabase(destination)` saves a copy of the artifact database file in the destination specified by `destination`.

The artifact database file, `artifacts.dmr`, is saved in the `derived` folder in the project root. This file tracks the project artifacts and their dependencies. Manually copying this file can lead to inconsistencies or incorrect behavior due to pending artifact changes.

You can use this function to create base artifact database files and save copies of artifact database files from different feature branches or CI pipeline jobs.

The function requires an open project.

Note Only supported in R2023b Update 5 and later releases.

Examples

Merge Project Analysis from Different Feature Branches

Process Advisor and the build system are able to detect changes to project files and identify outdated tasks by using the information in the artifact database file `artifacts.dmr`. When your team works on a project with multiple feature branches, you might need to merge different versions of `artifacts.dmr` into a single file that contains the latest project analysis. To create the file, you need to save a copy of the base artifact database file and then merge the `artifacts.dmr` files from each branch.

When your team members clone the project from source control, have them download the latest derived files, including the `artifacts.dmr` file that contains the latest analysis of the project. By default, digital thread stores the artifact database file inside the `derived` folder in the project root.

You can use a database or repository management tool to handle derived files effectively.

To resolve conflicts between the artifact database files from the different feature branches, you need to create a base artifact database file. Use the most recent `artifacts.dmr` file from the derived files as the base because that file represents the latest shared state of project analysis across the feature branches.

Create a copy of the artifact database file inside the `derived` folder and name the file `base.dmr`.

```
padv.util.saveArtifactDatabase(fullfile("derived", "base.dmr"))
```

As each team member works on their separate branches, the digital thread updates the `artifacts.dmr` file in their copy of the project to reflect their changes.

After a team member makes the changes on their branch, use the function `padv.util.saveArtifactDatabase` in each branch to save a copy of the artifact database file from that branch. For example, you might have artifact database files like `featureA.dmr` and `featureB.dmr`.

Merge the artifact database files into a new `artifacts.dmr` file by using the function `padv.util.mergeArtifactDatabases`. The base artifact database file is `base.dmr` and the artifact database files from the branches are `featureA.dmr` and `featureB.dmr`.

```
padv.util.mergeArtifactDatabases(...
Base = fullfile("derived", "base.dmr"), ...
Branches = [fullfile("derived", "featureA.dmr"), fullfile("derived", "featureB.dmr)], ...
Merged = fullfile("derived", "artifacts.dmr"))
```

This section describes how to merge artifact database files from separate feature branches, but you can also use these functions to merge artifact database files from jobs in CI and tasks that you run in parallel. Starting in R2023b Update 5, GitHub and Jenkins pipelines that you generate by using the function `padv.pipeline.generatePipeline` automatically merge artifact database files.

Input Arguments

destination — File destination

string

File destination for copied artifact database file, specified as a string.

The path must be relative to the project root or an absolute path and must include the `.dmr` extension.

Example: `fullfile("derived", "base.dmr")`

Data Types: string

Version History

Introduced in R2023b

padv.util.unpackExternalCodeCache

Unpack code generation target from Simulink cache files

Syntax

```
padv.util.unpackExternalCodeCache(cacheFiles)
```

Description

`padv.util.unpackExternalCodeCache(cacheFiles)` unpacks the code generation target from the Simulink cache files, `cacheFiles`.

An external code cache allows your team to generate code in parallel while maintaining up-to-date task results. For information on parallel code generation, see the `GenerateExternalCodeCache` property for the built-in task `padv.builtin.task.GenerateCode`.

If your team generates code in parallel by generating an external code cache, downstream tasks that depend on the generated code need to unpack the generated code target before running the task action. Built-in tasks that depend on generated code, like `padv.builtin.task.AnalyzeModelCode`, unpack the code generation target by using the utility function `padv.util.unpackExternalCodeCache`.

Examples

Unpack Code Generation Target

Generate and unpack code generation target.

Open the parallel code generation example.

```
processAdvisorParallelExampleStart
```

Generate code by running a code generation task iteration. For example, run the code generation task on the reference model `OuterLoop_Control`.

```
runprocess(Tasks = "padv.builtin.task.GenerateCode", ...  
           FilterArtifact = fullfile("02_Models", "OuterLoop_Control", ...  
                                     "specification", "OuterLoop_Control.slx"));
```

Find the external code cache file by using the built-in query.

```
q = padv.builtin.query.FindExternalCodeCache;  
artifactsArray = run(q);
```

Unpack the cache file.

```
padv.util.unpackExternalCodeCache(artifactsArray);
```

Input Arguments

cacheFiles — Address for external code cache files

array of `padv.Artifact` objects | cell array of character vectors | string array

Absolute or relative address for external code cache files, specified as either an array of `padv.Artifact` objects, a cell array of character vectors, or a string array.

The built-in code generation task, `padv.builtin.task.GenerateCode`, generates these cache files when you specify the task property `GenerateExternalCodeCache` as `true`.

The files must be:

- `.slxc.bk` files
- compatible with the `slxcunpack` function
- inside the project root folder

Process Advisor Example Projects

The support package includes example projects that you can use to try the Process Advisor app and build system. If you use GitHub, GitLab, or Jenkins, you can use the examples for those specific CI platforms to see example pipeline configuration files and example Dockerfiles.

Example projects:

- `processAdvisorExampleStart`
- `processAdvisorGitHubExampleStart`
- `processAdvisorGitLabExampleStart`
- `processAdvisorJenkinsExampleStart`
- `processAdvisorProjectReferenceExampleStart`

processAdvisorExampleStart

Set up Process Advisor example project

Syntax

```
processAdvisorExampleStart  
processAdvisorExampleStart(Name=Value)
```

Description

`processAdvisorExampleStart` sets up a Process Advisor example project. The function creates a new copy of the Process Advisor example project and automatically opens the Process Advisor app on the model `AHRS_Voter`.

`processAdvisorExampleStart(Name=Value)` sets up a Process Advisor example project using the specified options.

Input Arguments

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `processAdvisorExampleStart(ProjectFolder = "exampleProject")`

CI — Add pipeline configuration file for specific CI platform

`"" (default) | "github" | "gitlab" | "jenkins"`

Add pipeline configuration file for a specific CI platform, specified as:

- `"github"` — for GitHub
- `"gitlab"` — for GitLab (same as calling `processAdvisorGitLabExampleStart`)
- `"jenkins"` — for Jenkins (same as calling `processAdvisorJenkinsExampleStart`)

By default, the function does not add any pipeline configuration files to the example project.

To configure the pipeline configuration file to use automatic pipeline generation, use the argument `PipelineGen`.

Example: `processAdvisorExampleStart(CI="jenkins")`

Data Types: `string`

PipelineGen — Configure pipeline configuration file to use automatic pipeline generation

`true or 1 (default) | false or 0`

Configure the pipeline configuration file to use automatic pipeline generation, specified as a numeric or logical `0` (`false`) or `1` (`true`).

Example: `processAdvisorExampleStart(CI = "github", PipelineGen = false)`

Data Types: `logical`

IncludeDockerFile — Add example Dockerfile to project

`true` or `1` (default) | `false` or `0`

Add an example Dockerfile to the project, specified as a numeric or logical `0` (`false`) or `1` (`true`).

By default, the function adds an example Dockerfile named `Dockerfile` to the project root. You can use the example Dockerfile to create a Docker image that includes MATLAB, other MathWorks® products, and the CI/CD Automation for Simulink Check™ support package.

For more information on Dockerfiles, see "Create Docker Container for Support Package" in the User's Guide PDF.

Example: `processAdvisorExampleStart(IncludeDockerFile = false)`

Data Types: `logical`

ProjectFolder — Folder to download project into

`""` (default) | `string`

Folder to download project into, specified as a string.

By default, the function does not create a parent folder for the project.

Example: `processAdvisorExampleStart(ProjectFolder = "exampleProject")`

Data Types: `string`

processAdvisorGitHubExampleStart

Set up Process Advisor example for GitHub

Syntax

```
processAdvisorGitHubExampleStart
```

Description

`processAdvisorGitHubExampleStart` sets up Process Advisor example for GitHub (same as `processAdvisorExampleStart(CI = "github", PipelineGen = false)`).

The example includes a pipeline configuration file that can automatically generate a pipeline for GitHub.

processAdvisorGitLabExampleStart

Set up Process Advisor example for GitLab

Syntax

```
processAdvisorGitLabExampleStart
```

Description

`processAdvisorGitLabExampleStart` sets up Process Advisor example for GitLab (same as `processAdvisorExampleStart(CI="gitlab")`).

The example includes a pipeline configuration file that can automatically generate a pipeline for GitLab.

processAdvisorJenkinsExampleStart

Set up Process Advisor example for Jenkins

Syntax

```
processAdvisorJenkinsExampleStart
```

Description

`processAdvisorJenkinsExampleStart` sets up Process Advisor example for Jenkins (same as `processAdvisorExampleStart(CI="jenkins")`).

The example includes a pipeline configuration file that can automatically generate a pipeline for GitLab. You need to update the example `Jenkinsfile` to specify the bin directory for your MATLAB installation and the Git branch, credentials, and URL for your repository.

processAdvisorProjectReferenceExampleStart

Set up Process Advisor example that uses project references

Syntax

```
processAdvisorProjectReferenceExampleStart
```

Description

`processAdvisorProjectReferenceExampleStart` sets up a Process Advisor example project that uses project references.

Artifact Types

The build system uses artifact types to identify and categorize the different file types and modeling constructs in your project.

You can use an artifact type to find specific types of artifacts in your project:

```
% Find model files in the project by using the artifact type "sl_model_file"
q = padv.builtin.query.FindArtifacts(ArtifactType="sl_model_file");
results = run(q);
results.Address
```

You can also use an artifact type to create a `padv.Artifact` object that represents a specific artifact and run tasks associated with that artifact:

```
% specify the relative path to the model AHRV_Voter
model = padv.Artifact("sl_model_file",...
padv.util.ArtifactAddress(...
fullfile("02_Models","AHRV_Voter","specification","AHRV_Voter.slx")));

% run only the tasks for the AHRV_Voter model
runprocess(FilterArtifact = model)
```

The following table lists the valid artifact types.

Artifact Type	Description
"harness_info_file"	Harness info file
"m_class"	MATLAB class
"m_file"	MATLAB file
"m_func"	MATLAB function
"m_method"	MATLAB class method
"m_property"	MATLAB class property
"ma_config_file"	Model Advisor configuration file
"ma_justification_file"	Model Advisor justification file
"other_file"	Other file
"padv_output_file"	Process Advisor output file
"sf_chart"	Stateflow® chart
"sf_graphical_fcn"	Stateflow graphical function
"sf_group"	Stateflow group
"sf_state"	Stateflow state
"sf_state_transition_chart"	Stateflow state transition chart
"sf_truth_table"	Stateflow truth table
"sl_block_diagram"	Block diagram
"sl_data_dictionary_file"	Data dictionary file

Artifact Type	Description
"sl_embedded_matlab_fcn"	MATLAB function
"sl_harness_block_diagram"	Harness block diagram
"sl_harness_file"	Test harness file
"sl_library_file"	Library file
"sl_model_file"	Simulink model file
"sl_protected_model_file"	Protected Simulink model file
"sl_req_table"	Requirements Table
"sl_subsystem"	Subsystem
"sl_subsystem_file"	Subsystem file
"sl_test_case"	Simulink Test™ case
"sl_test_case_result"	Simulink Test case result
"sl_test_file"	Simulink Test file
"sl_test_iteration"	Simulink Test iteration
"sl_test_iteration_result"	Simulink Test iteration result
"sl_test_report_file"	Simulink Test result report
"sl_test_result_file"	Simulink Test result file
"sl_test_resultset"	Simulink Test result set
"sl_test_seq"	Test Sequence
"sl_test_suite"	Simulink Test suite
"sl_test_suite_result"	Simulink Test suite result
"zc_block_diagram"	System Composer™ architecture
"zc_component"	System Composer architecture component
"zc_file"	System Composer architecture file

Tokens

The default process model and built-in task source code use the following tokens as placeholders for dynamic path resolution of artifacts, directories and other information relevant to the process:

Token	Description
<code>\$INPUTARTIFACT\$</code>	Input artifact for task
<code>\$ITERATIONARTIFACT\$</code>	Current artifact that the task is acting on
<code>\$PWD\$</code>	Current working directory
<code>\$TIMESTAMP\$</code>	Current date and time in the format 'yyyy_mm_dd_HH_MM_ss'
<code>\$PROJECTROOT\$</code>	Root folder of project
<code>\$TASKNAME\$</code>	Task name or title
<code>\$DEFAULTOUTPUTDIR\$</code>	Default output directory for the process model
<code>\$ROOTITERATIONARTIFACT\$</code>	Root-level artifact for the iteration artifact

You can use these tokens in your process model, but note that:

- The output directory of a task cannot be specified as `$PROJECTROOT$`.
- The tokens `PWD` and `$TIMESTAMP$` are not supported by the pipeline generator.

Built-In Task Library

The support package CI/CD Automation for Simulink Check contains several built-in tasks that you can use when you define your process. You can reconfigure the tasks in the process model to change the task behavior. After you install the support package, you can view the source code files for the built-in tasks. In the MATLAB Command Window, enter:

```
cd(fullfile(matlabshared.supportpkg.getSupportPackageRoot,...
"toolbox","padv","build_service","ml","+padv","+builtin","+task"))
```

The built-in tasks include tasks for generating model reports, performing model analysis, running tests, generating code, and analyzing code:

Goal	Task Title	Task Instance	Requires License	Requires Display*
Model Reports	Generate SDD Report	padv.builtin.task.GenerateSDDReport	Simulink Report Generator™	✓
	Generate Simulink Web View	padv.builtin.task.GenerateSimulinkWebView		✓
	Generate Model Comparison	padv.builtin.task.GenerateModelComparison	Simulink	✓
Model Analysis	Check Modeling Standards	padv.builtin.task.RunModelStandards	Simulink Check	
	Detect Design Errors	padv.builtin.task.DetectDesignErrors	Simulink Design Verifier™	
Testing and Coverage	Merge Test Results	padv.builtin.task.MergeTestResults	Simulink Test	
	Run Tests	padv.builtin.task.RunTestsPerModel		
	Run Tests	padv.builtin.task.RunTestsPerTestCase		
Code Generation	Generate Code	padv.builtin.task.GenerateCode	Embedded Coder®	
Code Analysis	Check Coding Standards or Prove Code Quality	padv.builtin.task.AnalyzeModelCode	Polyspace® Bug Finder™ or Polyspace Code Prover™	
	Inspect Code	padv.builtin.task.RunCodeInspection	Simulink Code Inspector™	

*Built-in tasks that require a display might generate an error if there is no display available. If you run MATLAB using the `-nodisplay` option or you use a machine that does not have a display (like many CI runners and Docker containers), you should set up a virtual display server on that machine before you run the tasks. For information, see "Set Up Virtual Display for No-Display Machine" in the User's Guide.

Reference pages for the built-in task are listed alphabetically on the following pages:

- "Check Coding Standards or Prove Code Quality" on page 10-3
- "Check Modeling Standards" on page 10-11
- "Detect Design Errors" on page 10-17
- "Generate Code" on page 10-20
- "Generate Model Comparison" on page 10-23
- "Generate SDD Report" on page 10-26
- "Generate Simulink Web View" on page 10-30
- "Inspect Code" on page 10-33
- "Merge Test Results" on page 10-35
- "Run Tests (per model)" on page 10-40
- "Run Tests (per test case)" on page 10-44

Check Coding Standards or Prove Code Quality

An instance of this built-in task can be configured to either:

- **Check Coding Standards** (default) — Quickly analyze generated model code for many types of run-time defects, coding standards, and code metrics by using Polyspace Bug Finder.
- **Prove Code Quality** — Check *every* operation in your code for a set of possible run-time errors and try to prove the absence of the error for all execution paths by using Polyspace Code Prover. For information, see "Advanced - Polyspace Code Prover Option" and "Perform Code Prover Verification" in this PDF.

Note You can use both Bug Finder and Code Prover in your software development workflow. For information on how to include both a Bug Finder task and a Code Prover task in your process model, see "Check Coding Standards and Prove Code Quality" in this PDF.

For information on the differences between Bug Finder and Code Prover, see <https://www.mathworks.com/help/bugfinder/gs/use-bug-finder-and-code-prover.html>.

This task runs on the generated model code, iterating over either each model in the project or the project itself. If a model does not have generated code, the task skips the model and displays a warning message.

Optionally, you can have the task automatically upload results to Polyspace Access™ so that your team can review the results in the Polyspace Access web interface. For information, see "Advanced - Polyspace Access Configuration Options" and "Upload to Polyspace Access" in this PDF.

Task Instance	Task Title in Process Advisor
<code>padv.builtin.task.AnalyzeModelCode</code>	Check Coding Standards or Prove Code Quality

Note Starting in R2023b, this task is not supported on macOS (Apple silicon).

Prerequisites

- This task requires that your Polyspace installation is integrated with MATLAB and Simulink. If you have not already integrated your installation, use the function `polyspacesetup`. For information, see <https://www.mathworks.com/help/bugfinder/ug/integrate-polyspace-with-matlab-and-simulink.html>.
- If you start MATLAB with the `-batch` option, the task requires a Polyspace server product. The required server product depends on the task configuration:
 - **Check Coding Standards** (default) — Requires the Polyspace Bug Finder Server™ product.
 - **Prove Code Quality** — Requires the Polyspace Code Prover Server product.

Add Task to Process

Use the `addTask` function to add the task to the process model. The following code uses the `exist` function to check if Polyspace is installed and integrated before attempting to add the task to the process model:

```
psTask = addTask(pm, padv.builtin.task.AnalyzeModelCode);
```

If you want to have one task instance that uses Bug Finder and another task instance that uses Code Prover, see "Check Coding Standards and Prove Code Quality" in this PDF.

Reconfigure Task

You can change how a task performs an action by setting the properties of the task object.

For `padv.builtin.task.AnalyzeModelCode` task objects, the properties include:

Polyspace Options

Property	Description
<code>TreatAsRefModel</code>	By default, the task automatically identifies whether a model is a top model or a reference model before analyzing the model code. But you can specify <code>TreatAsRefModel</code> as <code>true</code> or <code>false</code> if you want to override the behavior and only analyze reference model code or top model code. Default: ""
<code>ResultDir</code>	Directory where build system stores results from analyzing model code Default: <code>fullfile('\$DEFAULTOUTPUTDIR\$', '\$ITERATIONARTIFACT\$', 'ps_results')</code>
<code>Reports</code>	Reports output by the task Default: ["BugFinderSummary" "CodingStandards"]
<code>ReportPath</code>	Path to reports output by the task Default: <code>string(fullfile('\$DEFAULTOUTPUTDIR\$', '\$ITERATIONARTIFACT\$', 'ps_results'))</code>
<code>ReportFormat</code>	Format of output reports Default: "PDF"
<code>ReportNames</code>	Names of output reports Default: ["\$ITERATIONARTIFACT \$_BugFinderSummary" "\$ITERATIONARTIFACT\$_CodingStandards"]

Advanced - Polyspace Code Prover Option

Property	Description
VerificationMode	<p>Polyspace mode, specified as either:</p> <ul style="list-style-type: none"> "BugFinder" — Perform Bug Finder analysis. "CodeProver" — Perform Code Prover verification. For information, see "Perform Code Prover Verification" in this PDF. <p>Default: "BugFinder"</p>

Advanced - Polyspace Analysis Options

Property	Description
Batch	<p>Option to run analysis on server (-batch)</p> <p>Default: false</p>
Scheduler	<p>Specify cluster or job scheduler (-scheduler)</p> <p>Default: string.empty</p>

Advanced - Polyspace Project Options

Property	Description
SavePsPrjFileAfterAnalysis	<p>Save Polyspace project file after analyzing model code</p> <p>Default: 1</p>
PsPrjFileName	<p>Polyspace project file path</p> <p>Default: "\$ITERATIONARTIFACT \$_BugFinder"</p>

Advanced - Polyspace Access Configuration Options

Property	Description
PsAccessEnable	<p>Enable task to upload Bug Finder analysis results to Polyspace Access, specified as a numeric or logical 1 (true) or 0 (false).</p> <p>Note If you specify PsAccessEnable as true, you must also specify values for the other Polyspace Access Configuration Options. For information, see "Upload to Polyspace Access".</p> <p>Default: false</p>
PsAccessHostName	<p>Polyspace Access machine host name, specified as a string. You can find the host name in the URL of the Polyspace Access interface, for example, <code>https://hostname:port/metrics/index.html</code>.</p> <p>Default: ""</p>
PsAccessPortNumber	<p>Polyspace Access port, specified as a string. You can find the port number in the URL of the Polyspace Access interface, for example, <code>https://hostname:portNumber/metrics/index.html</code>.</p> <p>Default: "9443"</p>
PsAccessProtocol	<p>HTTP protocol used to access Polyspace Access, specified as either "http" or "https".</p> <p>Default: "https"</p>
PsAccessCredentialsFile	<p>Full path to text file where you store your login credentials for Polyspace Access, specified as a string.</p> <p>A credentials file is useful if you do not want to store your credentials in your process model. For information on how to create a credentials file, see https://www.mathworks.com/help/bugfinder/ref/polyspaceaccess.html#mw_b91d7771-6fdf-4deb-8bf2-3e67252cce00.</p> <p>Alternatively, you can specify an API key (PsAccessApiKey) or a username and password (PsAccessUserName and PsAccessEncryptedPassword) to pass your credentials to Polyspace Access.</p> <p>Default: string.empty</p>

Property	Description
PsAccessApiKey	<p>API key for Polyspace Access, specified as a string.</p> <p>For information on API keys and how to assign an API key to a user, see the <code>login options</code>: https://www.mathworks.com/help/bugfinder/ref/polyspaceaccess.html#mw_595ad91b-5f80-4a87-b6e9-fecf45ce663c.</p> <p>Alternatively, you can use a credentials file (<code>PsAccessCredentialsFile</code>) or a username and password (<code>PsAccessUserName</code> and <code>PsAccessEncryptedPassword</code>) to pass your credentials to Polyspace Access.</p> <p>Default: <code>string.empty</code></p>
PsAccessUserName	<p>Username for Polyspace Access, specified as a string.</p> <p>For information on login credentials, see the <code>login options</code>: https://www.mathworks.com/help/bugfinder/ref/polyspaceaccess.html#mw_595ad91b-5f80-4a87-b6e9-fecf45ce663c.</p> <p>Alternatively, you can use a credentials file (<code>PsAccessCredentialsFile</code>) or an API key (<code>PsAccessApiKey</code>) to pass your credentials to Polyspace Access.</p> <p>Default: <code>" "</code></p>
PsAccessEncryptedPassword	<p>Password for Polyspace Access, specified as a string.</p> <p>For information on login credentials, see the <code>login options</code>: https://www.mathworks.com/help/bugfinder/ref/polyspaceaccess.html#mw_595ad91b-5f80-4a87-b6e9-fecf45ce663c.</p> <p>Alternatively, you can use a credentials file (<code>PsAccessCredentialsFile</code>) or an API key (<code>PsAccessApiKey</code>) to pass your credentials to Polyspace Access.</p> <p>Default: <code>" "</code></p>

Property	Description
PsAccessParentFolder	<p>Path of parent folder in Polyspace Access explorer under which you store uploaded results, specified as a string.</p> <p>For more information, see the upload options: https://www.mathworks.com/help/bugfinder/ref/polyspaceaccess.html#mw_80702b90-6802-4aad-9447-291610be4807</p> <p>Default: ""</p>
PsAccessResultsName	<p>Name of uploaded results in Polyspace Access explorer, specified as a string.</p> <p>For more information, see the upload options: https://www.mathworks.com/help/bugfinder/ref/polyspaceaccess.html#mw_80702b90-6802-4aad-9447-291610be4807</p> <p>Default: ""</p>

Perform Code Prover Verification

If you have a Polyspace Code Prover license, you can reconfigure the task to check every operation in your code for a set of possible run-time errors and try to prove the absence of the error for all execution paths.

To reconfigure the task, specify the `VerificationMode` property as `"CodeProver"`. For example:

```
psTask = pm.addTask(padv.builtin.task.AnalyzeModelCode);
psTask.Title = "Prove Code Quality";
psTask.VerificationMode = "CodeProver";
```

This code specifies a value for the `Title` property since the default task title is `"Check Coding Standards"`. You can use the other task properties to specify the report templates and other task settings.

Check Coding Standards and Prove Code Quality

You can use both Bug Finder and Code Prover in your software development workflow.

Both Polyspace Bug Finder and Polyspace Code Prover detect run-time errors through static analysis.

- Bug Finder quickly analyzes your code and detects many types of defects.
- Code Prover checks every operation in your code for a set of possible run-time errors and try to prove the absence of the error for all execution paths.

To include both a Bug Finder task and a Code Prover task in your process model, you must add two separate instances of the built-in task `padv.builtin.task.AnalyzeModelCode` to the process model. Each instance needs a unique value for the `Name` property. Use the `VerificationMode` property to specify whether the task uses Bug Finder (default) or Code Prover (`"CodeProver"`). You can use the other task properties to specify the report templates and other task settings.

For example:

```

%% Check Coding Standards with Polyspace Bug Finder
psbfTask = pm.addTask(padv.builtin.task.AnalyzeModelCode());
% Report Options
psbfTask.ResultDir = fullfile(defaultResultPath,"bug_finder");
psbfTask.ReportPath = fullfile(defaultResultPath,"bug_finder");

%% Prove Code Quality with Polyspace Code Prover
pscpTask = pm.addTask(padv.builtin.task.AnalyzeModelCode(Name="ProveCodeQuality"));
pscpTask.Title = "Prove Code Quality";
pscpTask.VerificationMode = "CodeProver";
% Report Options
pscpTask.ResultDir = string(fullfile(defaultResultPath,"code_prover"));
pscpTask.Reports = ["Developer", "CallHierarchy", "VariableAccess"];
pscpTask.ReportPath = string(fullfile(defaultResultPath,"code_prover"));
pscpTask.ReportNames = [...
    "$ITERATIONARTIFACT$_Developer", ...
    "$ITERATIONARTIFACT$_CallHierarchy", ...
    "$ITERATIONARTIFACT$_VariableAccess"];

```

Note that this code specifies different result directories and report paths for each task to prevent the task outputs from overwriting each other.

For information on:

- Differences between Bug Finder and Code Prover, see: <https://www.mathworks.com/help/bugfinder/gs/use-bug-finder-and-code-prover.html>
- How Bug Finder and Code Prover fit into a software development workflow, see: <https://www.mathworks.com/help/bugfinder/gs/workflow-using-both-polyspace-bug-finder-and-polyspace-code-prover.html>

Upload to Polyspace Access

If you have a Polyspace Access license, you can reconfigure this task to automatically upload results to Polyspace Access for your team to review.

Before you reconfigure the task, make sure that you have performed this one-time setup: https://www.mathworks.com/help/bugfinder/gs/run-bug-finder-on-server.html#mw_c7a5eb97-7327-4f99-9717-77773d462d8b

To reconfigure the task, update your process model to specify the property `PsAccessEnable` as `true` and to specify values for these properties:

- `PsAccessHostName`
- `PsAccessPortNumber`
- `PsAccessProtocol`
- `PsAccessParentFolder`
- And one of the following sets of credentials:
 - `PsAccessCredentialsFile`
 - `PsAccessApiKey`
 - `PsAccessUserName` and `PsAccessEncryptedPassword`

For example:

```

%% Check coding standards
if includeGenerateCodeTask && includeAnalyzeModelCode
    psTask = pm.addTask(padv.builtin.task.AnalyzeModelCode());
    psTask.addInputQueries(padv.builtin.query.FindFileWithAddress( ...
        Type = "ps_prj_file",...
        Path = fullfile("tools","CodingRulesOnly_config.psprj")));
    psTask.ResultDir = string(fullfile("$DEFAULTOUTPUTDIR$", ...
        "$ITERATIONARTIFACT$", "coding_standards"));
    psTask.Reports = "CodingStandards";
    psTask.ReportPath = string(fullfile("$DEFAULTOUTPUTDIR$", ...
        "$ITERATIONARTIFACT$", "coding_standards"));
    psTask.ReportNames = "$ITERATIONARTIFACT$_CodingStandards";
    psTask.ReportFormat = "PDF";

    % Polyspace Access configuration options
    psTask.PsAccessEnable = true;
    psTask.PsAccessHostName = "my-polyspace-access";
    psTask.PsAccessPortNumber = "9443";
    psTask.PsAccessProtocol = "https";
    psTask.PsAccessCredentialsFile = "C:\Users\username\myCredentials.txt";
    psTask.PsAccessParentFolder = "public/myProject";
    psTask.PsAccessResultsName = "$ITERATIONARTIFACT$_CodingStandards";

end

```

This code uses a credentials file, `myCredentials.txt`, to pass credentials to Polyspace Access, but you can also use an API key or a username and password. For information on how to generate and maintain credentials for Polyspace Access, see https://www.mathworks.com/help/bugfinder/ref/polyspaceaccess.html#mw_595ad91b-5f80-4a87-b6e9-fecf45ce663c.

For information on these properties, see the "Advanced - Polyspace Access Configuration Options" in the previous section.

For information on Polyspace Access, see:

- <https://www.mathworks.com/help/bugfinder/gs/send-polyspace-analysis-from-desktop-to-remote-server.html>
- <https://www.mathworks.com/help/bugfinder/gs/run-bug-finder-on-server.html>

Source Code

To view the source code for this built-in task, in the MATLAB Command Window, enter:

```
open padv.builtin.task.AnalyzeModelCode
```

Check Modeling Standards

This task uses the Model Advisor to check your models for modeling conditions and configuration settings that cause inaccurate or inefficient simulation of the system that the model represents. Running model standards checking can also help you verify compliance with industry standards and guidelines.

You can configure this task to specify which model standards the task runs. For example, you can specify a Model Advisor configuration file or list of check identifiers to include in the Model Advisor analysis. If you do not specify which model standards to run, the task runs a subset of high-integrity systems checks by default.

Task Instance	Task Title in Process Advisor
<code>padv.builtin.task.RunModelStandards</code>	Check Modeling Standards

Add Task to Process

Use the `addTask` function to add the task to the process model:

```
maTask = addTask(pm,padv.builtin.task.RunModelStandards);
```

Reconfigure Task

You can change certain task behaviors by setting the property values for the task object or by adding inputs to the task.

Change Property Values

You can change certain task behaviors by setting the properties of the task object. For example, if you want to specify a list of Model Advisor checks for the task to run, you can modify the `CheckIDList` property of the task object `maTask`:

```
maTask.CheckIDList = {'mathworks.jmaab.db_0032',...  
'mathworks.jmaab.jc_0281'};
```

Note If you want the task to use a Model Advisor configuration file or Model Advisor justification file, you do not need to change any property values, but you do need to add those files as inputs to the task. When you provide a file as an input to the task, the task can use the file, recognize changes to the file, and update the task status as needed. For information, see the sections "Use Model Advisor Configuration File" and "Use Model Advisor Justification File".

For `padv.builtin.task.RunModelStandards` task objects, the properties include:

Property	Description
CheckIDList	<p>List of unique identifiers for the Model Advisor checks, specified as a character vector, or cell array of character vectors. For example, <code>{'mathworks.jmaab.db_0032','mathworks.jmaab.jc_0281'}</code>.</p> <hr/> <p>Note If you specify <code>CheckIDList</code> and add a Model Advisor configuration file as an input for the task, the task runs Model Advisor using the Model Advisor configuration file and ignores the list of check IDs.</p> <hr/> <p>Default: <missing></p>
DisplayResults	<p>Report display setting for the Model Advisor, specified as either:</p> <ul style="list-style-type: none"> • "Summary" — Display summary of the system results in the Command Window • "Details" — Display a summary of the system results and the pass and fail results for each check on each system • "None" — Display no information in the Command Window <p>Default: "Summary"</p>
ExtensiveAnalysis	<p>Extensive analysis setting for the Model Advisor, specified as either:</p> <ul style="list-style-type: none"> • "off" — Model Advisor only runs checks in your configuration that do not trigger extensive analysis • "on" — Model Advisor runs each check in your Model Advisor configuration file, including checks that trigger extensive analysis with tools like Simulink Design Verifier <p>Default: "on"</p>
Force	<p>Force delete <code>modeladvisor/system</code> folders, specified as either:</p> <ul style="list-style-type: none"> • "off" — Prompt you before removing existing <code>modeladvisor/system</code> folders • "on" — Automatically removes existing <code>modeladvisor/system</code> folders <p>Default: "on"</p>

Property	Description
ParallelMode	Parallel execution setting for the Model Advisor, specified as "off" or "on". Default: "off"
ReportFormat	Format of the generated report, specified as either: <ul style="list-style-type: none"> "html" — HTML format "docx" — Microsoft Word document format Default: "html"
ReportName	Prefix for the Model Advisor report file name, specified as a character vector. An underscore and the model name, "_modelName", are appended to the report file name. Default: "\$ITERATIONARTIFACT \$ModelAdvisor"
ReportPath	Folder for the generated report, specified as a character vector. Default: string(fullfile('\$DEFAULTOUTPUTDIR\$', '\$ITERATIONARTIFACT\$', 'model_standards'))
ShowExclusions	Exclusion display setting for the report, specified as either: <ul style="list-style-type: none"> "off" — Report does not list Model Advisor check exclusions "on" — Report lists Model Advisor check exclusions Default: "on"
TempDir	Temporary working folder setting for the Model Advisor, specified as either: <ul style="list-style-type: none"> "off" — Run Model Advisor in the current working folder "on" — Run Model Advisor from a temporary working folder to avoid concurrency issues when running using a parallel pool Default: "off"

The task uses these properties to specify input arguments for the function `ModelAdvisor.run`. For more information on the arguments, see the Simulink Check documentation for `ModelAdvisor.run`: <https://www.mathworks.com/help/slcheck/ref/modeladvisor.run.html>.

Use Model Advisor Configuration File

By default, the **Check Modeling Standards** task runs a subset of high-integrity checks. If you want the task to run the Model Advisor checks specified by the Model Advisor configuration file, you can add the configuration file as an input to the task. For example, in the process model, you can use the `addInputQueries` function to specify an input query that finds the Model Advisor configuration file and use the built-in query `padv.builtin.query.FindFileWithAddress` as the input query to find the Model Advisor configuration file:

- The first argument, 'ma_config_file', specifies that the file is a Model Advisor configuration file.
- The second argument specifies the path to the Model Advisor configuration file. In this example, the configuration file is a file, `sampleChecks.json`, in the `tools` folder in the project.

```
%% Check modeling standards
% Tools required: Model Advisor
if includeModelStandardsTask
    maTask = pm.addTask(padv.builtin.task.RunModelStandards());

    % Specify which Model Advisor configuration to run
    maTask.addInputQueries(padv.builtin.query.FindFileWithAddress( ...
        Type = "ma_config_file",...
        Path = fullfile("tools","sampleChecks.json")));

end
```

Note If you provide both a list of check IDs (`CheckIDList`) and a Model Advisor configuration file for the task, the task runs Model Advisor using the Model Advisor configuration file and ignores the list of check IDs.

Use Model Advisor Justification File

Starting in R2023a, if you want the task to use your Model Advisor justification files when checking modeling standards, you can reconfigure the task to add the justification files as inputs. Add the built-in query `padv.builtin.query.FindMAJustificationFileForModel` as an input query for the task and specify the folder, `JustificationFolder`, that contains the justification files.

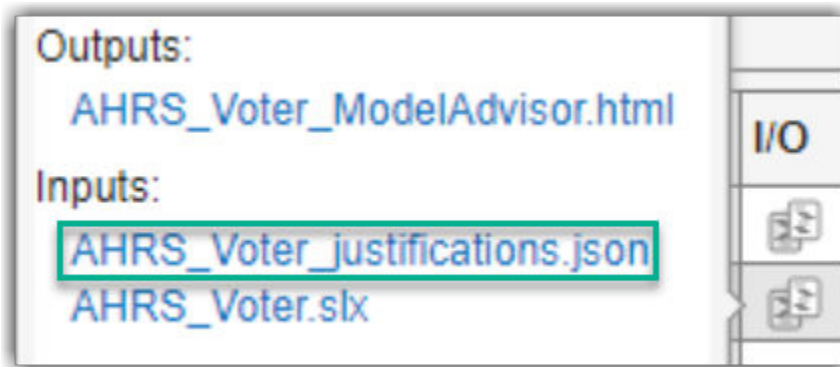
For example, if your justification files are in the directory `Justifications/ModelAdvisor` relative to your project root, use the function `addInputQueries` to add those justification files as inputs to the task:

```
%% Check modeling standards
% Tools required: Model Advisor
if includeModelStandardsTask
    maTask = pm.addTask(padv.builtin.task.RunModelStandards());

    % Find and use justification files
    maTask.addInputQueries(...
        padv.builtin.query.FindMAJustificationFileForModel(...
            JustificationFolder=fullfile("Justifications","ModelAdvisor")));

end
```

The justification file appears as an input in the **I/O** column in Process Advisor.



Create and Configure Multiple Instances of Check Modeling Standards

You can add multiple instances of a task to your process model to run different task configurations.

For example, you can have one instance of the built-in task `padv.builtin.task.RunModelStandards` that runs a specific Model Advisor configuration for a subset of models and another Model Advisor configuration for other models.

To create multiple instances of a task, you need to specify unique values for the Name properties of each task instance. By default, the task name is the name of the task class.

```
% Tasks need unique names
maTaskA = pm.addTask(padv.builtin.task.RunModelStandards(...
    Name = "maTaskA"));
maTaskB = pm.addTask(padv.builtin.task.RunModelStandards(...
    Name = "maTaskB"));
```

Use the other task properties to configure the task as needed. For example, you can specify which models the task runs on, which Model Advisor configuration file the task uses, and where the reports generate.

```
% Can specify unique title for task that appears in Process Advisor
maTaskA.Title = "Check Modeling Standards - A";
maTaskB.Title = "Check Modeling Standards - B";

% Can specify different Model Advisor configurations
maTaskA.addInputQueries(padv.builtin.query.FindFileWithAddress( ...
    Type='ma_config_file', Path=fullfile('configs','configA.json')));
maTaskB.addInputQueries(padv.builtin.query.FindFileWithAddress( ...
    Type='ma_config_file', Path=fullfile('configs','configB.json')));

% Can run on different sets of models
maTaskA.IterationQuery = padv.builtin.query.FindModels(...
    IncludePath = "control");
maTaskB.IterationQuery = padv.builtin.query.FindModelsWithLabel(...
    "ProjectLabelCategory","ProjectLabel");

% Specify unique locations for Model Advisor reports
maTaskA.ReportPath = fullfile( ...
    defaultResultPath,'model_standards_A_results');
maTaskB.ReportPath = fullfile( ...
    defaultResultPath,'model_standards_B_results');
```

Source Code

To view the source code for this built-in task, in the MATLAB Command Window, enter:

open `padv.builtin.task.RunModelStandards`

Detect Design Errors

This task uses Simulink Design Verifier to statically detect run-time errors and dead logic and to derive design ranges on your model. Design error detection can identify dead logic, integer overflow, division by zero, and violations of design properties and assertions. By default, this task outputs a design error detection report and data file.

Task Instance	Task Title in Process Advisor
<code>padv.builtin.task.DetectDesignErrors</code>	Detect Design Errors

Add Task to Process

Use the `addTask` function to add the task to the process model:

```
dedObj = addTask(pm, padv.builtin.task.DetectDesignErrors);
```

Reconfigure Task

You can change how a task performs an action by setting the properties of the task object.

For example, you can set the properties of the task object to change the analysis options:

```
dedObj.DetectDeadLogic = "on";
```

For `padv.builtin.task.DetectDesignErrors` task objects, the properties include:

Property	Description
<code>DataFileName</code>	Folder and file name for the MAT-file that contains the data generated during the analysis, specified as a character array. The data is stored in an <code>sldvData</code> structure. Default: "\$ITERATIONARTIFACT\$_sldvdata"
<code>DesignMinMaxCheck</code>	Check that the intermediate and output signals in models are within the range of specified minimum and maximum constraints, specified as "on" or "off". Default: "off"
<code>DetectActiveLogic</code>	Analyze models for active logic, specified as "on" or "off". Note that this parameter is enabled only if <code>DetectDeadLogic</code> is "on". Default: "off"
<code>DetectBlockInputRangeViolations</code>	Analyze models for block input range violations, specified as "on" or "off". Default: "off"

Property	Description
DetectDeadLogic	Analyze models for dead logic, specified as "on" or "off". Default: "off"
DetectDivisionByZero	Analyze models for division-by-zero errors, specified as "on" or "off". Default: "on"
DetectDSMAccessViolations	Analyze models for data store access violations, specified as "on" or "off". Default: "off"
DetectHISMViolationsHisl_0002	Check the usage of rem and reciprocal operations that cause non-finite results, specified as "on" or "off". Default: "on"
DetectHISMViolationsHisl_0003	Check the usage of Square Root (Sqrt) operations with inputs that can be negative, specified as "on" or "off". Default: "on"
DetectHISMViolationsHisl_0004	Check the usage of log and log10 operations that cause non-finite results, specified as "on" or "off". Default: "on"
DetectHISMViolationsHisl_0028	Check the usage of Reciprocal Square Root (rSqrt) blocks with inputs that can go zero or negative, specified as "on" or "off". Default: "on"
DetectInfNaN	Analyze models for non-finite and NaN floating-point values, specified as "on" or "off". Default: "off"
DetectIntegerOverflow	Analyze models for integer and fixed-point data overflow errors, specified as "on" or "off". Default: "on"
DetectOutOfBounds	Analyze models for out of bounds array access errors, specified as "on" or "off". Default: "on"
DetectSubnormal	Analyze models for subnormal floating-point values, specified as "on" or "off". Default: "off"

Property	Description
DisplayReport	After analysis, display the report that Simulink Design Verifier generates, specified as "on" or "off". Default: "off"
MaxProcessTime	Maximum time (in seconds) that Simulink Design Verifier spends analyzing a model, specified as a double. Default: 300
Options	Options for the generated report, specified as "summary", "objectives", "objects", or a combination of these options in an array. Default: ["summary" "objectives"]
ReportFormat	Format of the generated report, specified as either: <ul style="list-style-type: none"> "HTML" — HTML format "PDF" — PDF format Default: "HTML"
ReportFilePath	Folder and file name for the analysis report, specified as a character array. Default: fullfile('\$DEFAULTOUTPUTDIR\$', '\$ITERATIONARTIFACT\$', 'design_error_detections', '\$ITERATIONARTIFACT\$_Design_Error_Detection_Report')
ShowUI	Display messages in the log window, specified as a numeric or logical 1 (true) or 0 (false). When ShowUI is specified as 0, messages appear in the MATLAB Command Window. Default: 0

The task uses these properties to create a design verification options object by using the function `sldvoptions` and generate a report by using the function `sldvreport`. For more information on the options, see the Simulink Design Verifier documentation for `sldvoptions` and `sldvreport`.

Source Code

To view the source code for this built-in task, in the MATLAB Command Window, enter:

```
open padv.builtin.task.DetectDesignErrors
```

Generate Code

This task uses Embedded Coder to generate code. The task returns the generated code report as an output file.

Task Instance	Task Title in Process Advisor
<code>padv.builtin.task.GenerateCode</code>	Generate Code

You can use this task to generate code, iterating over either each model in the project or the project.

Note This task generates code but does not build executable files.

Add Task to Process

Use the `addTask` function to add the task to the process model:

```
addTask(pm,padv.builtin.task.GenerateCode);
```

Reconfigure Task

You can change how a task performs an action by setting the properties of the task object.

For `padv.builtin.task.GenerateCode` task objects, the properties include:

General Behavior

Property	Description
TreatAsRefModel	<p>By default, the task automatically identifies whether a model is a top model or a reference model before generating code. But you can specify <code>TreatAsRefModel</code> as <code>true</code> or <code>false</code> if you want to override the behavior and only generate reference model code or top model code.</p> <p>Default: []</p>
GenerateCodeOnly	<p>By default, the task generates code only and does not build an executable file.</p> <p>Default: 1</p>
ObfuscateCode	<p>Generate obfuscated C code, specified as a numeric or logical 1 (<code>true</code>) or 0 (<code>false</code>).</p> <p>Default: 0</p>
UpdateThisModelReferenceTarget	<p>Conditional rebuild option for model reference build, specified as either:</p> <ul style="list-style-type: none"> • "Force" • "IfOutOfDateOrStructuralChange" • "IfOutOfDate" <p>Default: "IfOutOfDateOrStructuralChange"</p>
ForceTopModelBuild	<p>Force top model of model hierarchy to build, specified as a numeric or logical 1 (<code>true</code>) or 0 (<code>false</code>).</p> <p>Default: 0</p>

Parallel Code Generation

Property	Description
IncludeModelReferenceSimulationTargets	Build model reference simulation targets, specified as a numeric or logical 1 (true) or 0 (false). Default: false
GenerateExternalCodeCache	Setting to collect only SLXC files as task outputs, specified as a numeric or logical 1 (true) or 0 (false). Default: false
ExternalCodeCacheDirectory	Location to save SLXC file, specified as a string. Default: fullfile('\$DEFAULTOUTPUTDIR', '\$ITERATIONARTIFACT\$', 'external_code_cache')
TrackAllGeneratedCode	Track all code files, not just model.c and model.h files, specified as a numeric or logical 1 (true) or 0 (false). Default: false

The task uses these properties to generate code by using the function `slbuild`. For more information on the `slbuild` arguments, see the documentation for `slbuild`.

Source Code

To view the source code for this built-in task, in the MATLAB Command Window, enter:

```
open padv.builtin.task.GenerateCode
```

Generate Model Comparison

This task uses the Comparison Tool to compare models in the project to their ancestors in Git and publishes a comparison report. The task compares your version of the model to either the latest or previous version on the main branch in Git:

- If you make a change to a model and run the task, the task compares your version of the model to either the head of the current branch or latest version on the main branch in Git.
- If you do not make any changes to a model and run the task, the task compares the model to the previous version available on the main branch in Git.

You can use the task properties to specify different report options, filtering options, and the name of the Git branch used for the comparison.

Task Instance	Task Title in Process Advisor
<code>padv.builtin.task.GenerateModelComparison</code>	Generate Model Comparison

Prerequisites

- To find and compare model ancestors, this task requires that you use Git source control for your project. For information on how to add a project to Git source control, see <https://www.mathworks.com/help/simulink/ug/add-a-project-to-source-control.html>.
- If you run MATLAB using the `-nodisplay` option or you use a machine that does not have a display (like many CI runners and Docker containers), you should set up a virtual display server before you include this task in your process model. For information, see "Set Up Virtual Display for No-Display Machine" in the User's Guide.

Add Task to Process

Use the `addTask` function to add the task to the process model:

```
modelCompareTask = addTask(pm, padv.builtin.task.GenerateModelComparison());
```

Reconfigure Task

You can change how a task performs an action by setting the properties of the task object.

For `padv.builtin.task.GenerateModelComparison` task objects, the properties include:

Report Options

Property	Description
ReportName	Names of generated comparison report, specified as a string. Default: "\$ITERATIONARTIFACT \$ _Model_Comparison"
ReportPath	Path to generated comparison report, specified as a string. Default: fullfile('\$DEFAULTOUTPUTDIR\$', '\$ITERATIONARTIFACT \$', 'model_comparison')
ReportFormat	Format of generated comparison report, specified as either "DOCX", "HTML", or "PDF". Default: "HTML"







Advanced Options

Property	Description
Filter	Setting for filtering model comparison report, specified as either: <ul style="list-style-type: none"> "unfiltered" — Removes all filtering from the comparison. "default" — Default filtering for the comparison, which hides any non-functional changes. Default: "default"
MainBranch	Name of Git branch used for comparison, specified as a string. Default: "main"

The task uses these properties to specify input arguments for the function `visdiff`. For information on `visdiff`, see <https://www.mathworks.com/help/matlab/ref/visdiff.html>.

Launch Tool Action

In Process Advisor, when you point to the task and click ... > **Compare to Ancestor**, you can open the Model Comparison tool.

Tasks	I/O	Details
▼ <input checked="" type="checkbox"/> Generate Model Comparison		<input checked="" type="checkbox"/> 5
<input checked="" type="checkbox"/> AHRS_Voter.slx		<input checked="" type="checkbox"/> 1
<input checked="" type="checkbox"/> Actuator_Control.slx		<input checked="" type="checkbox"/> 1
<input checked="" type="checkbox"/> Flight_Control.slx		<input checked="" type="checkbox"/> 1
<input checked="" type="checkbox"/> InnerLoop_Control.slx		<input checked="" type="checkbox"/> 1
<input checked="" type="checkbox"/> OuterLoop_Control.slx		<input checked="" type="checkbox"/> 1
▼ <input type="checkbox"/> Generate SDD		
<input type="checkbox"/> AHRS_Voter.slx		
<input type="checkbox"/> Actuator_Control.slx		
<input type="checkbox"/> Flight_Control.slx		

Source Code

To view the source code for this built-in task, in the MATLAB Command Window, enter:

open `padv.builtin.task.GenerateModelComparison`

Generate SDD Report

This task uses Simulink Report Generator to generate a System Design Description (SDD) report from a predefined template. The System Design Description report provides a summary or detailed information about a system design represented by a model.

Task Instance	Task Title in Process Advisor
<code>padv.builtin.task.GenerateSDDReport</code>	Generate SDD Report

Prerequisites

- If you run MATLAB using the `-nodisplay` option or you use a machine that does not have a display (like many CI runners and Docker containers), you should set up a virtual display server before you include this task in your process model. For information, see "Set Up Virtual Display for No-Display Machine" in the User's Guide.

Add Task to Process

Use the `addTask` function to add the task to the process model:

```
addTask(pm,padv.builtin.task.GenerateSDDReport);
```

Reconfigure Task

You can change how a task performs an action by setting the properties of the task object.

For `padv.builtin.task.GenerateSDDReport` task objects, the properties include:

Property	Description
<code>DisplayReport</code>	Open the generated report, specified as a numeric or logical 1 (true) or 0 (false). Default: 0
<code>IncludeCustomLibraries</code>	Include custom libraries in the design description, specified as a numeric or logical 1 (true) or 0 (false). Default: 0
<code>IncludeDetails</code>	Include design details, like block parameters, in the design description, specified as a numeric or logical 1 (true) or 0 (false). Default: 1
<code>IncludeGlossary</code>	Include a glossary in the design description, specified as a numeric or logical 1 (true) or 0 (false). Default: 1

Property	Description
IncludeLookupTables	<p>Include lookup tables in the design description, specified as a numeric or logical 1 (true) or 0 (false).</p> <p>Default: 1</p>
IncludeModelRefs	<p>Include model references in the design description, specified as a numeric or logical 1 (true) or 0 (false).</p> <p>Default: 0</p>
IncludeRequirementsLinks	<p>Include requirement links in the design description, specified as a numeric or logical 1 (true) or 0 (false).</p> <p>Default: 1</p>
IncrOutputName	<p>Increment the report name to avoid overwriting an existing report, specified as a numeric or logical 1 (true) or 0 (false).</p> <p>Default: 0</p>
LegalNotice	<p>Legal notice that appears on the report, specified as a string array.</p> <p>Default: "For Internal Distribution Only"</p>
PackageType	<p>File package type index of the generated HTML report, specified as either:</p> <ul style="list-style-type: none"> • 1 — Zipped. Package report files in a single compressed file that has the report name, with a .zip extension. • 2 — Unzipped. Generate the report files in a subfolder of the current folder. The subfolder has the report name. • 3 — Both zipped and unzipped. Package the report files as both zipped and unzipped. <p>Note that this parameter is enabled when ReportFormat is "html".</p> <p>Default: 1</p>

Property	Description
ReportFormat	<p>Output format for the generated report, specified as either:</p> <ul style="list-style-type: none"> "html" — HTML format. You can use the property <code>PackageType</code> to specify whether report files are zipped, unzipped, or produce both zipped and unzipped files. "pdf" — PDF format "docx" — Microsoft Word document format <p>Default: "html"</p>
ReportName	<p>File name for the generated report, specified as a string array.</p> <p>Default: "\$ITERATIONARTIFACT\$_SDD"</p>
ReportPath	<p>Path to the generated report, specified as a string array.</p> <p>Default: <code>string(fullfile('\$DEFAULTOUTPUTDIR\$', '\$ITERATIONARTIFACT\$', 'system_design_description'))</code></p>
ReportTitle	<p>Title of the report, specified as a string.</p> <p>Default: ""</p>
TitleImagePath	<p>Path of image that appears on report title page, specified as a string.</p> <p>Default: ""</p>
Subtitle	<p>Subtitle of the report, specified as a string.</p> <p>Default: "Design Description"</p>
TimeFormat	<p>Format of the data and time that the report generated, specified as a string.</p> <p>Default: ""</p>
UseStatusWindow	<p>Display report generation status messages in separate window, specified as a numeric or logical 1 (true) or 0 (false).</p> <p>Default: 0</p>

The task uses these properties to specify the report options for an SDD object. For information on the System Design Description options, see <https://www.mathworks.com/help/rptgenext/ug/system-design-description-dialog-box.html>.

Source Code

To view the source code for this built-in task, in the MATLAB Command Window, enter:

open `padv.builtin.task.GenerateSDDReport`

Generate Simulink Web View

This task uses the Simulink Report Generator to create a Web view for your models.

Task Instance	Task Title in Process Advisor
<code>padv.builtin.task.GenerateSimulinkWebView</code>	Generate Simulink Web View

Prerequisites

- If you run MATLAB using the `-nodisplay` option or you use a machine that does not have a display (like many CI runners and Docker containers), you should set up a virtual display server before you include this task in your process model. For information, see "Set Up Virtual Display for No-Display Machine" in the User's Guide.

Add Task to Process

Use the `addTask` function to add the task to the process model:

```
addTask(pm,padv.builtin.task.GenerateSimulinkWebView);
```

Reconfigure Task

You can change how a task performs an action by setting the properties of the task object.

For `padv.builtin.task.GenerateSimulinkWebView` task objects, the properties include:

Property	Description
<code>FollowLinks</code>	Follow links into library blocks, specified as either: <ul style="list-style-type: none"> • 0 — Does not allow you to follow links into library blocks in a web view • 1 — Allows you to follow links into library blocks in a web view Default: 1
<code>FollowModelReference</code>	Access referenced models in a web view, specified as either: <ul style="list-style-type: none"> • 0 — Does not allow you to access referenced models in a web view • 1 — Allows you to access referenced models in a web view Default: 1
<code>IncludeNotes</code>	Include user notes, specified as a numeric or logical 1 (true) or 0 (false). Default: 1

Property	Description
IncrementalExport	Starting in R2022b, export models incrementally, specified as a numeric or logical 1 (true) or 0 (false). Default: 0
LookUnderMasks	Export the ability to interact with masked blocks, specified as either "None" or "All". Default: "All"
PackagingType	Type of web view output package, specified as "zipped", "unzipped", or "both". Default: "unzipped"
RecurseFolder	Export models in subfolders, specified as a numeric or logical 1 (true) or 0 (false). Default: 0
ReportName	File name for the generated report, specified as a string array. Default: "\$ITERATIONARTIFACT\$_webview"
ReportPath	Path to the generated report, specified as a string array. Default: string(fullfile('\$DEFAULTOUTPUTDIR\$', '\$ITERATIONARTIFACT\$', 'webview'))
SearchScope	Systems to export, relative to the system_name system, specified as "All", "CurrentAndBelow", "CurrentAndAbove", or "Current". Default: "All"
ShowProgressBar	Display the status bar when you export a web view, specified as a numeric or logical 1 (true) or 0 (false). Default: 0
ViewFile	Display the web view in a web browser when you export the web view, specified as a numeric or logical 1 (true) or 0 (false). Default: 0

The task uses these properties to specify the input arguments for the function `slwebview`. For information on the arguments, see the documentation for `slwebview`: <https://www.mathworks.com/help/rptgenext/ug/slwebview.html>.

Source Code

To view the source code for this built-in task, in the MATLAB Command Window, enter:

```
open padv.builtin.task.GenerateSimulinkWebView
```

Inspect Code

This task uses the Simulink Code Inspector to detect unintended functionality in your models by establishing model-to-code and code-to-model traceability. The results of this task can help you to satisfy code-review objectives in DO-178 and other high-integrity standards.

Task Instance	Task Title in Process Advisor
<code>padv.builtin.task.RunCodeInspection</code>	Inspect Code

This task runs on the generated model code, iterating over either each model in the project or the project itself.

Note This task is not supported on macOS.

Add Task to Process

Use the `addTask` function to add the task to the process model and use the `IsTopModel` property to specify that the task should inspect reference model code:

```
addTask(pm,padv.builtin.task.RunCodeInspection);
```

Reconfigure Task

You can change how a task performs an action by setting the properties of the task object.

For `padv.builtin.task.RunCodeInspection` task objects, the properties include:

Property	Description
<code>IsTopModel</code>	By default, the task automatically identifies whether a model is a top model or a reference model before generating code. But you can specify <code>IsTopModel</code> as <code>true</code> or <code>false</code> if you want to override the behavior and only generate top model code or reference model code. Default: <code>[]</code>
<code>ReportFolder</code>	Path to the generated report, specified as a string array. The task uses this property to specify the report folder for code inspection. Default: <code>string(fullfile('\$DEFAULTOUTPUTDIR\$', '\$ITERATIONARTIFACT\$', 'code_inspection'))</code>

The task uses these properties to create a code inspection object (`slci.Configuration`).

Source Code

To view the source code for this built-in task, in the MATLAB Command Window, enter:

open `padv.builtin.task.RunCodeInspection`

Merge Test Results

This task uses Simulink Test and Simulink Coverage™ to generate the following artifacts for a model:

- a consolidated test results report
- a merged model coverage report for normal mode simulation results
- a merged code coverage report for software-in-the-loop (SIL) mode results
- a merged code coverage report for processor-in-the-loop (PIL) mode results

Task Instance	Task Title in Process Advisor
<code>padv.builtin.task.MergeTestResults</code>	Merge Test Results

Prerequisites

- You can use the built-in task `padv.builtin.task.RunTestsPerTestCase` to run your test cases. This task only supports merging coverage results from normal simulation mode results. The merging of coverage results from software-in-the-loop (SIL), processor-in-the-loop (PIL), and other modes is not supported.

Add Task to Process

Use the `addTask` function to add the task to the process model:

```
addTask(pm,padv.builtin.task.MergeTestResults);
```

Reconfigure Task

You can change how a task performs an action by setting the properties of the task object.

For `padv.builtin.task.MergeTestResults` task objects, the properties include:

Property	Description
<code>CovAllTestInMdlSumm</code>	Include each test in the model summary, specified as a numeric or logical 1 (<code>true</code>) or 0 (<code>false</code>). Default: 0
<code>CovBarGrInMdlSumm</code>	Produce bar graphs in the model summary, specified as a numeric or logical 1 (<code>true</code>) or 0 (<code>false</code>). Default: 1
<code>CovComplexInBlkTable</code>	Include cyclomatic complexity numbers in block details, specified as a numeric or logical 1 (<code>true</code>) or 0 (<code>false</code>). Default: 1

Property	Description
CovComplexInSumm	Include cyclomatic complexity numbers in summary, specified as a numeric or logical 1 (true) or 0 (false). Default: 1
CovElimFullCov	Exclude fully covered model objects from report, specified as a numeric or logical 1 (true) or 0 (false). Default: 0
CovElimFullCovDetails	Exclude fully covered model object details from report, specified as a numeric or logical 1 (true) or 0 (false). Default: 1
CovFiltExecMetric	Filter Execution metric from report, specified as a numeric or logical 1 (true) or 0 (false). Default: 0
CovFiltSFEvent	Filter Stateflow events from report, specified as a numeric or logical 1 (true) or 0 (false). Default: 0
CovGenerateWebViewReport	Generate web view report, specified as a numeric or logical 1 (true) or 0 (false). Default: 0
CovHitCntInMdlSumm	Display hit/count ratio in the model summary, specified as a numeric or logical 1 (true) or 0 (false). Default: 0
CovReportName	Name of generated model coverage report, specified as a string. Default: "\$ITERATIONARTIFACT \$_ModelCoverage_Report.html"
CovReportNameSIL	Name of generated software-in-the-loop (SIL) code coverage report, specified as a string. Default: "\$ITERATIONARTIFACT \$_SIL_CodeCoverage_Report.html"
CovReportNamePIL	Name of generated processor-in-the-loop (PIL) code coverage report, specified as a string. Default: "\$ITERATIONARTIFACT \$_PIL_CodeCoverage_Report.html"

Property	Description
CovShowReport	<p>Show coverage report, specified as a numeric or logical 1 (true) or 0 (false).</p> <p>Default: 0</p>
CovTwoColorBarGraphs	<p>Use two-color bar graphs, specified as a numeric or logical 1 (true) or 0 (false).</p> <p>Default: 1</p>
Author	<p>Name of the report author, specified as a string array.</p> <p>Default: "Process Advisor"</p>
IncludeComparisonSignalPlots	<p>Include the signal comparison plots defined under baseline criteria, equivalence criteria, or assessments using the verify operator in the test case, specified as a numeric or logical 1 (true) or 0 (false).</p> <p>Default: 0</p>
IncludeCoverageResult	<p>Include coverage metrics that are collected at test execution, specified as a numeric or logical 1 (true) or 0 (false).</p> <p>Default: 1</p>
IncludeErrorMessage	<p>Include error messages from the test case simulations, specified as a numeric or logical 1 (true) or 0 (false).</p> <p>Default: 1</p>
IncludeMATLABFigures	<p>Include the figures opened from a callback script, custom criteria, or by the model in the report, specified as a numeric or logical 1 (true) or 0 (false).</p> <p>Default: 0</p>
IncludeMLVersion	<p>Include the version of MATLAB used to run the test cases, specified as a numeric or logical 1 (true) or 0 (false).</p> <p>Default: 1</p>
IncludeSimulationMetadata	<p>Include simulation metadata for each test case or iteration, specified as a numeric or logical 1 (true) or 0 (false).</p> <p>Default: 0</p>

Property	Description
IncludeSimulationSignalPlots	Include the simulation output plots of each signal, specified as a numeric or logical 1 (true) or 0 (false). Default: 0
IncludeTestRequirement	Include the test requirement link defined under Requirements in the test case, specified as a numeric or logical 1 (true) or 0 (false). Default: 1
IncludeTestResults	Include all or a subset of test results in the report, specified as either: <ul style="list-style-type: none"> • 0 — Passed and failed results • 1 — Only passed results • 2 — Only failed results Default: 0
LaunchReport	Open the generated report, specified as a numeric or logical 1 (true) or 0 (false). Default: 0
LoadSimulationSignalData	Task loads simulation signal data when loading test results, specified as a numeric or logical 1 (true) or 0 (false). Default: 0
NumPlotColumnsPerPage	Number of columns of plots to include on report pages, specified as an integer 1, 2, 3, or 4. Default: 1
NumPlotRowsPerPage	Number of rows of plots to include on report pages, specified as an integer 1, 2, 3, or 4. Default: 2
ReportFormat	Output format for the generated report, specified as either: <ul style="list-style-type: none"> • "pdf" — PDF format • "docx" — Microsoft Word document format • "zip" — Zipped file Default: "pdf"

Property	Description
ReportPath	Path to the generated report, specified as a string array. Default: <code>fullfile('\$DEFAULTOUTPUTDIR\$', '\$ITERATIONARTIFACT\$', 'test_results')</code>
ReportName	File name for the generated report, specified as a string array. Default: <code>"\$ITERATIONARTIFACT\$_Test_Report"</code>
ReportTitle	Title of the report, specified as a string. Default: <code>"\$ITERATIONARTIFACT\$ TEST REPORT"</code>

Source Code

To view the source code for this built-in task, in the MATLAB Command Window, enter:

```
open padv.builtin.task.MergeTestResults
```

Run Tests (per model)

This task uses Simulink Test to run the test cases associated with your models. The task runs each test cases for each model. Process Advisor shows the name of each model under the **Run Tests** task. Certain tests might generate code.

Task Instance	Task Title in Process Advisor
<code>padv.builtin.task.RunTestsPerModel</code>	Run Tests

Note Since this task runs each test case individually, the task only executes test-case level callbacks. The task does not execute test-file level callbacks or test-suite level callbacks.

Add Task to Process

Use the `addTask` function to add the task to the process model:

```
addTask(pm, padv.builtin.task.RunTestsPerModel);
```

Reconfigure Task

You can change how a task performs an action by setting the properties of the task object.

For `padv.builtin.task.RunTestsPerModel` task objects, the properties include:

Property	Description
Author	Name of the report author, specified as a string array. Default: "Process Advisor"
IncludeComparisonSignalPlots	Include the signal comparison plots defined under baseline criteria, equivalence criteria, or assessments using the <code>verify</code> operator in the test case, specified as a numeric or logical 1 (true) or 0 (false). Default: 0
IncludeCoverageResult	Include coverage metrics that are collected at test execution, specified as a numeric or logical 1 (true) or 0 (false). Default: 1
IncludeErrorMessage	Include error messages from the test case simulations, specified as a numeric or logical 1 (true) or 0 (false). Default: 1

Property	Description
IncludeMATLABFigures	<p>Include the figures opened from a callback script, custom criteria, or by the model in the report, specified as a numeric or logical 1 (true) or 0 (false).</p> <p>Default: 0</p>
IncludeMLVersion	<p>Include the version of MATLAB used to run the test cases, specified as a numeric or logical 1 (true) or 0 (false).</p> <p>Default: 1</p>
IncludeSimulationMetadata	<p>Include simulation metadata for each test case or iteration, specified as a numeric or logical 1 (true) or 0 (false).</p> <p>Default: 0</p>
IncludeSimulationSignalPlots	<p>Include the simulation output plots of each signal, specified as a numeric or logical 1 (true) or 0 (false).</p> <p>Default: 0</p>
IncludeTestRequirement	<p>Include the test requirement link defined under Requirements in the test case, specified as a numeric or logical 1 (true) or 0 (false).</p> <p>Default: 1</p>
IncludeTestResults	<p>Include all or a subset of test results in the report, specified as either:</p> <ul style="list-style-type: none"> • 0 — Passed and failed results • 1 — Only passed results • 2 — Only failed results <p>Default: 0</p>
LaunchReport	<p>Open the generated report, specified as a numeric or logical 1 (true) or 0 (false).</p> <p>Default: 0</p>
NumPlotColumnsPerPage	<p>Number of columns of plots to include on report pages, specified as an integer 1, 2, 3, or 4.</p> <p>Default: 1</p>
NumPlotRowsPerPage	<p>Number of rows of plots to include on report pages, specified as an integer 1, 2, 3, or 4.</p> <p>Default: 2</p>

Property	Description
ReportFormat	<p>Output format for the generated report, specified as either:</p> <ul style="list-style-type: none"> • "pdf" — PDF format • "docx" — Microsoft Word document format • "zip" — Zipped file <p>Default: "pdf"</p>
ReportPath	<p>Path to the generated report, specified as a string array.</p> <p>Default: <code>fullfile('\$DEFAULTOUTPUTDIR\$', '\$ITERATIONARTIFACT\$', 'test_results')</code></p>
ReportName	<p>File name for the generated report, specified as a string array.</p> <p>Default: "\$ITERATIONARTIFACT\$_Test"</p>
ReportTitle	<p>Title of the report, specified as a string.</p> <p>Default: "\$ITERATIONARTIFACT\$ REPORT"</p>
ResultFileName	<p>Name of test result file, specified as a string array.</p> <p>Default: "\$ITERATIONARTIFACT\$_ResultFile"</p>
ResultFilePath	<p>Path to test result file, specified as a string array.</p> <p>Default: <code>fullfile('\$DEFAULTOUTPUTDIR\$', '\$ITERATIONARTIFACT\$', 'test_results')</code></p>
SaveResultsAfterRun	<p>Save the test results to a file after execution, specified as a numeric or logical 1 (true) or 0 (false).</p> <p>Default: 1</p>

Property	Description
SimulationMode	<p data-bbox="865 300 1036 331"><i>Since R2023a</i></p> <p data-bbox="865 359 1471 485">Simulation mode for running tests, specified as "Normal", "Accelerator", "Rapid Accelerator", "Software-in-the-Loop", or "Processor-in-the-Loop".</p> <p data-bbox="865 512 1463 606">By default, the property is empty (""), which means the built-in task uses the simulation mode that you define in the test itself.</p> <p data-bbox="865 634 1471 791">If you specify a value other than "", the built-in task overrides the simulation mode set in the Test Manager. You do not need to update the test parameters or settings to run the test in the new mode.</p> <p data-bbox="865 819 1008 850">Default: ""</p>

Source Code

To view the source code for this built-in task, in the MATLAB Command Window, enter:

```
open padv.builtin.task.RunTestsPerModel
```

Run Tests (per test case)

This task uses Simulink Test to run the test cases associated with your models. The task runs each test case individually. Process Advisor shows the name of each test case under the **Run Tests** task. Certain tests might generate code.

Task Instance	Task Title in Process Advisor
<code>padv.builtin.task.RunTestsPerTestCase</code>	Run Tests

Note Since this task runs each test case individually, the task only executes test-case level callbacks. The task does not execute test-file level callbacks or test-suite level callbacks.

To generate a consolidated test results report and a merged coverage report for your model, you can use the built-in task `padv.builtin.task.MergeTestResults`.

Add Task to Process

Use the `addTask` function to add the task to the process model:

```
addTask(pm,padv.builtin.task.RunTestsPerTestCase);
```

Reconfigure Task

You can change how a task performs an action by setting the properties of the task object.

For `padv.builtin.task.RunTestsPerTestCase` task objects, the properties include:

Property	Description
<code>ResultFileName</code>	Name of test result file, specified as a string array. Default: "\$ITERATIONARTIFACT \$_ResultFile"

Property	Description
SimulationMode	<p data-bbox="865 300 1036 327"><i>Since R2023a</i></p> <p data-bbox="865 359 1468 485">Simulation mode for running tests, specified as "Normal", "Accelerator", "Rapid Accelerator", "Software-in-the-Loop", or "Processor-in-the-Loop".</p> <p data-bbox="865 516 1468 604">By default, the property is empty (""), which means the built-in task uses the simulation mode that you define in the test itself.</p> <p data-bbox="865 636 1468 789">If you specify a value other than "", the built-in task overrides the simulation mode set in the Test Manager. You do not need to update the test parameters or settings to run the test in the new mode.</p> <p data-bbox="865 821 1008 848">Default: ""</p>

If you want the task to only run on test cases that have a specific test tag, specify the `IterationQuery` using the built-in query `padv.builtin.query.FindTestCasesForModel` and specify the test tag using the `Tags` argument. For example, to have the task only run on test cases that were tagged with the test tag `FeatureA`:

```
addTask(pm,padv.builtin.task.RunTestsPerTestCase,...
IterationQuery = padv.builtin.query.FindTestCasesForModel(Tags="FeatureA"));
```

Source Code

To view the source code for this built-in task, in the MATLAB Command Window, enter:

```
open padv.builtin.task.RunTestsPerTestCase
```


Built-In Query Library

The support package CI/CD Automation for Simulink Check contains several built-in queries that can find specific sets of artifacts in your project. You can use the queries when you define your process, but note that you can only use certain queries as an input query (`InputQueries`) or iteration query (`IterationQuery`) for a task. The built-in queries include:

Query	Returns	Iteration Query	Input Query
<code>padv.builtin.query.FindArtifacts</code>	Artifacts that meet specified criteria	✓	✓*
<code>padv.builtin.query.FindCodeForModel</code>	Generated code files and <code>buildInfo.mat</code> for a model	✓	✓
<code>padv.builtin.query.FindExternalCodeCache</code>	External code cache files in project		✓
<code>padv.builtin.query.FindFilesWithLabel</code>	Files with specific project label	✓	
<code>padv.builtin.query.FindFileWithAddress</code>	File at the specified address	✓	✓
<code>padv.builtin.query.FindMAJustificationFileForModel</code>	Find Model Advisor justification files	✓	✓
<code>padv.builtin.query.FindModels</code>	Models	✓	✓*
<code>padv.builtin.query.FindModelsWithLabel</code>	Models with specific project label	✓	
<code>padv.builtin.query.FindModelsWithTestCases</code>	Models associated with a test case	✓	
<code>padv.builtin.query.FindProjectFile</code>	Project file	✓	✓
<code>padv.builtin.query.FindRefModels</code>	Referenced models	✓	
<code>padv.builtin.query.FindRequirements</code>	Requirement sets	✓	✓*
<code>padv.builtin.query.FindRequirementsForModel</code>	Requirements associated with model	✓	✓
<code>padv.builtin.query.FindTestCasesForModel</code>	Test cases associated with model	✓	✓
<code>padv.builtin.query.FindTopModels</code>	Top models	✓	✓
<code>padv.builtin.query.GetDependentArtifacts</code>	Dependent artifacts for artifact		✓

Query	Returns	Iteration Query	Input Query
<code>padv.builtin.query.GetIterationArtifact</code>	Artifact that the task is iterating over		✓
<code>padv.builtin.query.GetOutputsOfDependentTask</code>	Outputs from immediate predecessor task		✓

*You cannot use the query as an input query if you specify the query input argument `InProject` as `true`.

Reference pages for the built-in task are listed alphabetically on the following pages.

Tip You can access help for the built-in queries from the MATLAB Command Window. For example, this code returns help information for the built-in query `padv.builtin.query.FindArtifacts`:

```
help padv.builtin.query.FindArtifacts
```

padv.builtin.query.FindArtifacts

This query returns each of the artifacts in project that meet the criteria specified by the optional input arguments.

Syntax

`q = padv.builtin.query.FindArtifacts()` finds all artifacts in the project.

`q = padv.builtin.query.FindArtifacts(Name, Value)` find artifacts that meet the criteria specified by one or more name-value arguments. For example, to find artifacts that include "HLR" in the full file path, specify `IncludePath="HLR"`.

Input Arguments

Name-Value Arguments

- **Name** — Unique identifier for query, specified as character vector or string. Example: `"CustomQueryForArtifacts"`
- **ArtifactType** — Type of artifact, specified as a string or a cell array of character vectors. For a list of valid artifact types, see the chapter "Artifact Types" in this PDF. Example: `{"sl_model_file", "m_file"}`
- **IncludeLabel** — Find artifacts that have a specific project label, specified as a cell array where the first entry is the project label category and the second entry is the project label name. Example: `{"Classification", "Design"}`
- **ExcludeLabel** — Exclude artifacts that have a specific project label, specified as a cell array where the first entry is the project label category and the second entry is the project label name. Example: `{"Classification", "Design"}`
- **IncludePath** — Find artifacts where the path contains specific text, specified as a character vector or string. Example: `"HLR"`
- **ExcludePath** — Exclude artifacts where the path contains specific text, specified as a character vector. Example: `"HLR"`
- **InProject** — Include only artifacts that have been added to the project, specified as a numeric or logical 1 (true) or 0 (false). Example: `true`

Note If you specify `InProject` as `true`, you can no longer use the query as an input query.

- **FilterSubFileArtifacts** — Filter out sub-file artifacts from query results, specified as a numeric or logical 1 (true) or 0 (false). A sub-file is a part of a larger file. For example, a subsystem is a sub-file of a model file. Example: `false`

Methods

run	<p>Return artifacts from query</p> <p>The run method inside this built-in query runs on a query object <code>obj</code> and returns artifacts <code>artifacts</code>. If you inherit from this built-in query, make sure to use the same method signature inside your custom query:</p> <pre>function artifacts = run(obj,~) ... end</pre>
-----	--

Use in Process Model

You can use this query in your process model to find artifacts that your tasks can iterate over (`IterationQuery`) or use as inputs (`InputQueries`).

For example, suppose that you have a custom task, `MyCustomTask`, that you add to your process model. You can use the built-in query `padv.builtin.query.FindArtifacts` to find specific types of artifacts. To find the data dictionaries in the project, you specify the `ArtifactType` argument as `"sl_data_dictionary_file"`.

```
taskObj = addTask(pm, "MyCustomTask",...
    IterationQuery = padv.builtin.query.FindArtifacts(...
    ArtifactType = "sl_data_dictionary_file"),...
    InputQueries = padv.builtin.query.GetIterationArtifact);
```

In this example, specifying `InputQueries` as `padv.builtin.query.GetIterationArtifact` allows the task to use the artifacts returned by `IterationQuery` as inputs to the task.

Test Outside Process Model

Although you typically use queries inside your process model, you can run queries outside of your process model to confirm which artifacts the query returns.

For example:

- 1 Open a project. For this example, you can open the Process Advisor example project.


```
processAdvisorExampleStart
```
- 2 Create an instance of the query. You can use the arguments of the query to filter the query results. For example, you can use the `IncludeLabel` argument to have the query only return artifacts that use the `Design` project label from the `Classification` project label category.

```
q = padv.builtin.query.FindArtifacts(...
    IncludeLabel = {'Classification', 'Design'});
```

- 3 Run the query and inspect the array of artifacts that the query returns.

```
run(q)
ans =
```

1×26 Artifact array with properties:

Type
Parent
ArtifactAddress

padv.builtin.query.FindCodeForModel

This query returns only the generated code files and `buildInfo.mat` for a model.

Syntax

`q = padv.builtin.query.FindCodeForModel()` finds the generated code files and `buildInfo.mat` for a model.

`q = padv.builtin.query.FindCodeForModel(Name, Value)` finds artifacts that meet the criteria specified by one or more name-value arguments.

Input Arguments

Name-Value Arguments

- **IncludeLabel** — Find artifacts that have a specific project label, specified as a cell array where the first entry is the project label category and the second entry is the project label name. Example: `{"Classification", "Design"}`
- **ExcludeLabel** — Exclude artifacts that have a specific project label, specified as a cell array where the first entry is the project label category and the second entry is the project label name. Example: `{"Classification", "Design"}`
- **IncludePath** — Find artifacts where the path contains specific text, specified as a character vector or string. Example: `"HLR"`
- **ExcludePath** — Exclude artifacts where the path contains specific text, specified as a character vector. Example: `"HLR"`

Methods

run	<p>Return artifacts from query</p> <p>The <code>run</code> method inside this built-in query runs on a query object <code>obj</code> and returns artifacts that are associated with the artifact <code>iterationArtifact</code>. If you inherit from this built-in query, make sure to use the same method signature inside your custom query:</p> <pre>function artifacts = run(obj, iterationArtifact) ... end</pre>
-----	--

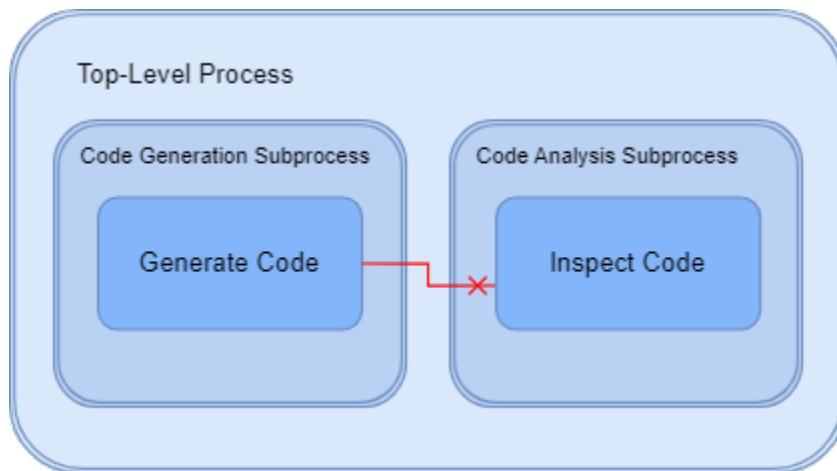
Use in Process Model

You can use this query in your process model to find artifacts that your tasks can iterate over (`IterationQuery`) or use as inputs (`InputQueries`).

For example, suppose that you create one subprocess to contain your code generation tasks and another subprocess to contain your code analysis tasks:

```
spCodeGen = pm.addSubprocess("Code Generation Tasks");
spCodeAnalysis = pm.addSubprocess("Code Analysis Tasks");
```

Your code analysis tasks need access to the generated code, but the tasks themselves cannot directly depend on the code generation task because that relationship would cross the subprocess boundary.



To pass the generated code from your code generation subprocess to your code analysis subprocess, you can:

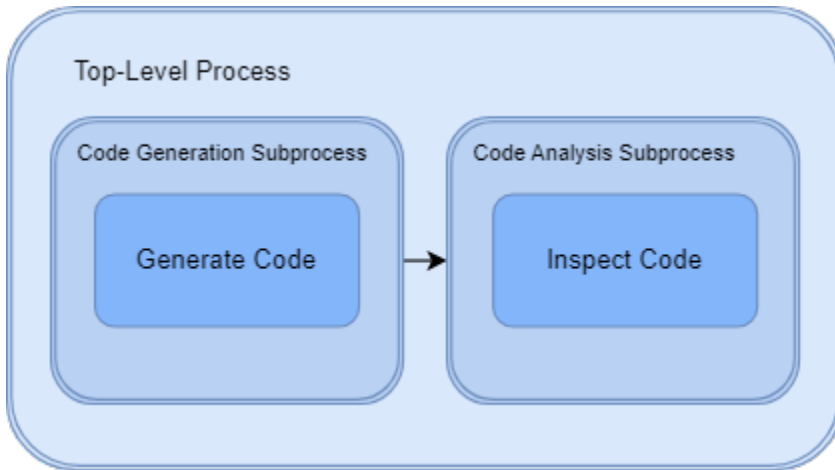
- Update any code analysis tasks to find and use the generated model code as an input to the task using the built-in query `padv.builtin.query.FindCodeForModel`
- Specify that the code analysis subprocess depends on the code generation subprocess

% Update Code Analysis Tasks to find and use model code as an input to the task

```
psbfTask = spCodeAnalysis.addTask(padv.builtin.task.AnalyzeModelCode(...
    InputQueries=padv.builtin.query.FindCodeFolderForModel));
pscptTask = spCodeAnalysis.addTask(padv.builtin.task.AnalyzeModelCode(...
    Name="ProveCodeQuality", InputQueries=padv.builtin.query.FindCodeFolderForModel));
slciTask = spCodeAnalysis.addTask(padv.builtin.task.RunCodeInspection(...
    InputQueries=padv.builtin.query.FindCodeForModel));
```

% Code Analysis Subprocess depends on Code Generation Subprocess

```
spCodeAnalysis.dependsOn(spCodeGen);
```



padv.builtin.query.FindExternalCodeCache

This query returns the external code cache files (.slxc.bk) in the project.

Syntax

`q = padv.builtin.query.FindExternalCodeCache()` finds the external code cache files (.slxc.bk) in the project. You can use this query to find external code cache files that you generate using the built-in task `padv.builtin.task.GenerateCode`. The built-in task generates an external code cache when you specify the task property `GenerateExternalCodeCache` as `true`.

`q = padv.builtin.query.FindArtifacts(Name = queryName)` finds the files and specifies a new name, `queryName`, for the query object.

Input Arguments

Name-Value Arguments

- **Name** — Unique identifier for query, specified as character vector or string. Example: "CustomQueryForArtifacts"

Methods

run	<p>Return artifacts from query</p> <p>The run method inside this built-in query runs on a query object <code>obj</code> and returns artifacts <code>artifacts</code>. If you inherit from this built-in query, make sure to use the same method signature inside your custom query:</p> <pre>function artifacts = run(obj,~) ... end</pre>
-----	--

Use in Task Definition

You can use this query in your task definition to find and unpack external code cache files.

For example, if your team generates code in parallel by generating an external code cache, downstream tasks that depend on the generated code need to unpack the generated code target before performing the main task action. If you have a custom task that depends on that generated code, you can find the external code cache files by using the built-in query `padv.builtin.query.FindExternalCodeCache` and unpack the code generation target by using the utility function `padv.util.unpackExternalCodeCache`. For example, you might use:

```
% Before main task action, access the generated code
% by finding and unpacking the external code cache
q = padv.builtin.query.FindExternalCodeCache;
artifactsArray = run(q);
    if ~isempty(artifactsArray)
```

```
        padv.util.unpackExternalCodeCache(artifactsArray)
    end
```

```
% <definition for main task action that uses the generated code>
```

For information about parallel code generation and external code caches, see the documentation for the `GenerateExternalCodeCache` property for the built-in task `padv.builtin.task.GenerateCode`. The external code cache allows your team to generate code in parallel while maintaining up-to-date task results.

Test Query from Command Window

Although you typically use queries inside a process model or task definition, you can run queries directly from the MATLAB Command Window to confirm which artifacts the query returns.

For example:

Open the parallel code generation example.

```
processAdvisorParallelExampleStart
```

Generate code by running a code generation task iteration. For example, run the code generation task on the reference model `OuterLoop_Control`.

```
runprocess(Tasks = "padv.builtin.task.GenerateCode", ...
    FilterArtifact = fullfile("02_Models","OuterLoop_Control", ...
    "specification","OuterLoop_Control.slx"));
```

Find the external code cache file by using the built-in query.

```
q = padv.builtin.query.FindExternalCodeCache;
artifactsArray = run(q);
```

Unpack the cache file.

```
padv.util.unpackExternalCodeCache(artifactsArray);
```

padv.builtin.query.FindFilesWithLabel

This query returns files in the project that use the specified project label.

Syntax

`q = padv.builtin.query.FindFilesWithLabel(categoryName, labelName)` finds files that use the project label `labelName` from the project label category `categoryName`.

`q = padv.builtin.query.FindFilesWithLabel(____, Name, Value)` find files that use the project label `labelName` from the project label category `categoryName` and meet the criteria specified by one or more name-value arguments. For example, to only return artifacts that include "HLR" in the full file path, specify `IncludePath="HLR"`.

Input Arguments

- **categoryName** — Name of project label category, specified as a character vector or string. Example: "ModelLabels"
- **labelName** — Project label name, specified as character vector or string. Example: "RunModelAdvisor"

Name-Value Arguments

- **Name** — Unique identifier for query, specified as character vector or string. Example: "CustomQueryForArtifacts"
- **IncludeLabel** — Find artifacts that have a specific project label, specified as a cell array where the first entry is the project label category and the second entry is the project label name. Example: {"Classification", "Design"}
- **ExcludeLabel** — Exclude artifacts that have a specific project label, specified as a cell array where the first entry is the project label category and the second entry is the project label name. Example: {"Classification", "Design"}
- **IncludePath** — Find artifacts where the path contains specific text, specified as a character vector or string. Example: "HLR"
- **ExcludePath** — Exclude artifacts where the path contains specific text, specified as a character vector. Example: "HLR"
- **InProject** — Include only artifacts that have been added to the project, specified as a numeric or logical 1 (true) or 0 (false). Example: true

Methods

run	<p>Return artifacts from query</p> <p>The run method inside this built-in query runs on a query object <code>obj</code> and returns artifacts <code>artifacts</code>. If you inherit from this built-in query, make sure to use the same method signature inside your custom query:</p> <pre>function artifacts = run(obj,~) ... end</pre>
-----	--

Use in Process Model

You can use this query in your process model to find artifacts for your task to iterate over (`IterationQuery`).

For example, suppose that you want the built-in task `padv.builtin.task.RunModelStandards` to only run for models that use the project label `RunModelAdvisor` from the project label category `ModelLabels`. You can change the `IterationQuery` for the task to specify a different set of artifacts for the task to run on. You can use the built-in query `padv.builtin.query.FindFilesWithLabel` to find the models that use that project label. Specify the first input argument as the project label category and the second argument as the project label name.

```
maTask = pm.addTask(padv.builtin.task.RunModelStandards());
maTask.IterationQuery = ...
    padv.builtin.query.FindFilesWithLabel("ModelLabels", "RunModelAdvisor");
```

Note You cannot use this query as an input query (`InputQueries`).

padv.builtin.query.FindFileWithAddress

This query returns the file at the specified address in the project.

Syntax

`q = padv.builtin.query.FindFileWithAddress(Type = ArtifactType, Path = FilePath)` finds a file, of type `ArtifactType`, at the address specified by `FilePath`.

To find multiple files, specify `ArtifactType` and `FilePath` as vectors of the same length.

`q = padv.builtin.query.FindFileWithAddress(____, Name=Value)` finds and returns a file using the settings specified by one or more name-value arguments. For example, if you do not want the build system to track changes to the returned file, specify `TrackArtifacts=false`.

Input Arguments

- **ArtifactType** — Type of artifact, specified as a string or string array. For a list of valid artifact types, see the chapter "Artifact Types" in this PDF.

Examples:

- "sl_model_file"
- ["sl_model_file", "m_file"]

- **FilePath** — Path to file, specified as a character vector or string.

Examples:

- `fullfile("02_Models", "AHRS_Voter", "specification", "AHRS_Voter.slx")`
- `[fullfile("myFiles", "myModel.slx"), fullfile("myFiles", "myScript.m")]`

Name-Value Arguments

- **ValidateFileExistence** — Validate that the file exists before attempting to return the file in the query results, specified as a numeric or logical 1 (`true`) or 0 (`false`). Default: `true`
- **TrackArtifacts** — Setting that controls whether the build system tracks changes to the returned file, specified as a numeric or logical 1 (`true`) or 0 (`false`). Default: `true`

For more information, see "Turn Off Change Tracking for Input Artifacts".

Note If you specify `TrackArtifacts=false`, you can no longer use the query as an iteration query. The build system needs to track changes iteration artifacts to identify the iterations for the task.

Methods

run	<p>Return artifacts from query</p> <p>The run method inside this built-in query runs on a query object <code>obj</code> and returns artifacts <code>artifacts</code>. If you inherit from this built-in query, make sure to use the same method signature inside your custom query:</p> <pre>function artifacts = run(obj,~) ... end</pre>
-----	--

Use in Process Model

By default, you can use this query in your process model to find artifacts that your tasks can iterate over (`IterationQuery`) or use as inputs (`InputQueries`). However, if you specify `TrackArtifacts=false`, you can no longer use this query as an iteration query because the build system needs to track changes iteration artifacts to identify the iterations for the task.

Find Single File

For example, by default, the **Check Modeling Standards** task runs a subset of high-integrity checks. But suppose that you want the task to run the Model Advisor checks specified by the Model Advisor configuration file `sampleChecks.json` instead. In the process model, you can use the `addInputQueries` function to specify an input query that finds the Model Advisor configuration file. You can use the built-in query `padv.builtin.query.FindFileWithAddress` as an input query to find the Model Advisor configuration file:

- The first argument, `"ma_config_file"`, specifies that the artifact type of the file is a Model Advisor configuration file.
- The second argument specifies the path to the Model Advisor configuration file.

```
%% Checking model standards on a model
if includeModelStandardsTask
    maTask = pm.addTask(padv.builtin.task.RunModelStandards());
    maTask.ReportPath = fullfile( ...
        defaultResultPath, 'model_standards_results');

    % Specify which Model Advisor configuration file to run
    maTask.addInputQueries(padv.builtin.query.FindFileWithAddress( ...
        Type = "ma_config_file", ...
        Path = fullfile("tools", "sampleChecks.json")));

end
```

Find Multiple Files

To find multiple files, specify the artifact type (`Type`) and the file path (`Path`) using vectors of the same length. For example:

```
padv.builtin.query.FindFileWithAddress(...
    Type=["ma_config_file", ...
        "sl_model_file"], ...
```



```
Path=[fullfile("tools","sampleChecks.json"),...
      fullfile("02_Models","AHRV_Voter","specification","AHRV_Voter.slx")])
```

If you only specify one value for Type, the query uses the same artifact type for each specified file specified by Path.

```
padv.builtin.query.FindFileWithAddress(...
Type="ma_config_file",...
Path=[fullfile("tools","sampleChecks.json"), fullfile("tools","myCustomChecks.json")])
```

Test Outside Process Model

Although you typically use queries inside your process model, you can run queries outside of your process model to confirm which artifacts the query returns.

For example:

- 1 Open a project. For this example, you can open the Process Advisor example project.

```
processAdvisorExampleStart
```

- 2 Create an instance of the query. For example, create a query that finds a file with the artifact type Model Advisor configuration file (ma_config_file) at the file path specified by fullfile("tools","sampleChecks.json").

```
q = padv.builtin.query.FindFileWithAddress( ...
    Type = "ma_config_file",...
    Path = fullfile("tools","sampleChecks.json"))
```

- 3 Run the query.

```
run(q)
```

The query returns the specified artifact.

```
ans =
    "tools\sampleChecks.json"
```

padv.builtin.query.FindMAJustificationFileForModel

Starting in R2023a, this query returns the Model Advisor justification file associated with the current model.

Syntax

`q = padv.builtin.query.FindMAJustificationFileForModel(JustificationFolder = relativePathToFolder)` finds the Model Advisor justification file associated with the current model by searching for the file within the specified folder `relativePathToFolder`. The query expects that the current iteration artifact is a model and that the Model Advisor justification filename is the model name followed by `_justifications.json`. The query returns the justification file as a `padv.Artifact` object of type `ma_justification_file`.

`q = padv.builtin.query.FindMAJustificationFileForModel(___, Name = queryName)` finds the Model Advisor justification file and specifies a new name, `queryName`, for the query object.

Note This query is only supported in R2023a and later releases.

Input Arguments

- **relativePathToFolder** — Relative path to folder that contains justification files (`.json`) for models in the project, specified as a character vector or string. Example: `fullfile("Justifications","ModelAdvisor")`
- **queryName** — Unique identifier for query, specified as character vector or string. Example: `"CustomFindJustificationFile"`

Use in Process Model

You can use this query in your process model to provide the justification files as inputs for the built-in task `padv.builtin.task.RunModelStandards` (InputQueries) or to find justification files that your tasks can iterate over (IterationQuery).

Use Justifications When Checking Modeling Standards

If you want the built-in task `padv.builtin.task.RunModelStandards` to use your Model Advisor justification files when checking modeling standards, you can reconfigure the task to add the justification files as inputs. Add the built-in query `padv.builtin.query.FindMAJustificationFileForModel` as an input query for the task and specify the folder, `JustificationFolder`, that contains the justification files. For example, if your justification files are in the directory `Justifications/ModelAdvisor` relative to your project root, use the function `addInputQueries` to add those justification files as inputs to the task:

```
%% Check modeling standards
% Tools required: Model Advisor
if includeModelStandardsTask
    maTask = pm.addTask(padv.builtin.task.RunModelStandards());

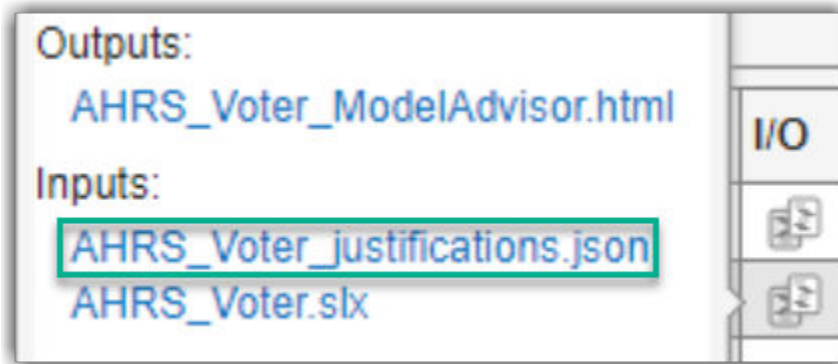
    % Find and use justification files
```

```

    maTask.addInputQueries(...
        padv.builtin.query.FindMAJustificationFileForModel(...
            JustificationFolder=fullfile("Justifications","ModelAdvisor"));
    end

```

The justification file appears as an input in the **I/O** column in Process Advisor.



Iterate over Justification Files in Folder

If you want a task to iterate over the justification files for the models in the project, you can use this query as the `IterationQuery` for a task. For example:

```

myTask = pm.addTask("MyCustomTask",...
    IterationQuery = padv.builtin.query.FindMAJustificationFileForModel(...
        JustificationFolder = fullfile("Justifications","ModelAdvisor"));

```

padv.builtin.query.FindModels

This query returns each of the models in project that meet the criteria specified by the optional input arguments.

Syntax

`q = padv.builtin.query.FindModels()` finds all models in the project. The models include Simulink models and System Composer models.

`q = padv.builtin.query.FindModels(Name, Value)` find models that meet the criteria specified by one or more name-value arguments. For example, to find models that include `Control` in the full file path, specify `IncludePath="Control"`.

Input Arguments

Name-Value Arguments

- **Name** — Unique identifier for query, specified as character vector or string. Example: `"CustomQuery"`
- **IncludeLabel** — Find artifacts that have a specific project label, specified as a cell array where the first entry is the project label category and the second entry is the project label name. Example: `{"Classification", "Design"}`
- **ExcludeLabel** — Exclude artifacts that have a specific project label, specified as a cell array where the first entry is the project label category and the second entry is the project label name. Example: `{"Classification", "Design"}`
- **IncludePath** — Find artifacts where the path contains specific text, specified as a character vector or string. Example: `"Control"`
- **ExcludePath** — Exclude artifacts where the path contains specific text, specified as a character vector. Example: `"Control"`
- **InProject** — Include only artifacts that have been added to the project, specified as a numeric or logical 1 (`true`) or 0 (`false`). Example: `true`

Note If you specify `InProject` as `true`, you can no longer use the query as an input query.

Methods

run	<p>Return artifacts from query</p> <p>The run method inside this built-in query runs on a query object <code>obj</code> and returns artifacts <code>artifacts</code>. If you inherit from this built-in query, make sure to use the same method signature inside your custom query:</p> <pre>function artifacts = run(obj,~) ... end</pre>
-----	--

Use in Process Model

You can use this query in your process model to find artifacts that your tasks can iterate over (IterationQuery) or use as inputs (InputQueries).

For example, suppose that you only want to run the **Check Modeling Standards** task for models that have `Control` in their file path. By default, the **Check Modeling Standards** task uses the built-in query `padv.builtin.query.FindModels` as the `IterationQuery`. In the process model, you can change the `IterationQuery` for the task to:

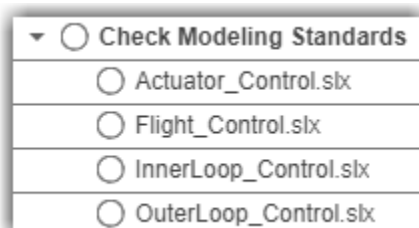
- 1 Use the built-in query `padv.builtin.query.FindModels` to find the models in the project.
- 2 Specify the `IncludePath` argument of the query to filter out any models that do not have `Control` in the file path.

```
%% Checking model standards on a model
if includeModelStandardsTask
    maTask = pm.addTask(padv.builtin.task.RunModelStandards());
    maTask.ReportPath = fullfile( ...
        defaultResultPath, 'model_standards_results');

    % Specify which set of artifacts to run for
    maTask.IterationQuery = ...
        padv.builtin.query.FindModels(IncludePath = "Control")

end
```

For the Process Advisor example project, the model `AHRS_Voter.slx` no longer appears under the task title in Process Advisor because `AHRS_Voter.slx` does not include `Control` in the path.



Test Outside Process Model

Although you typically use queries inside your process model, you can run queries outside of your process model to confirm which artifacts the query returns.

For example:

- 1 Open a project. For this example, you can open the Process Advisor example project.

```
processAdvisorExampleStart
```

- 2 Create an instance of the query. You can use the arguments of the query to filter the query results. For example, you can use the `IncludeLabel` argument to have the query only return artifacts that use the `Design` project label from the `Classification` project label category.

```
q = padv.builtin.query.FindModels(...  
IncludeLabel = {"Classification", "Design"});
```

- 3 Run the query and inspect the array of artifacts that the query returns.

```
run(q)
```

```
ans =
```

```
1x5 Artifact array with properties:
```

```
Type  
Parent  
ArtifactAddress
```

padv.builtin.query.FindModelsWithLabel

This query returns each of the models in project that use the specified project label.

Syntax

`q = padv.builtin.query.FindModelsWithLabel(categoryName, labelName)` finds models that use the project label `labelName` from the project label category `categoryName`.

Input Arguments

- **categoryName** — Name of project label category, specified as a character vector or string. Example: "ModelLabels"
- **labelName** — Project label name, specified as character vector or string. Example: "RunModelAdvisor"

Name-Value Arguments

- **Name** — Unique identifier for query, specified as character vector or string. Example: "CustomQueryForArtifacts"
- **IncludeLabel** — Find artifacts that have a specific project label, specified as a cell array where the first entry is the project label category and the second entry is the project label name. Example: {"Classification", "Design"}
- **ExcludeLabel** — Exclude artifacts that have a specific project label, specified as a cell array where the first entry is the project label category and the second entry is the project label name. Example: {"Classification", "Design"}
- **IncludePath** — Find artifacts where the path contains specific text, specified as a character vector or string. Example: "HLR"
- **ExcludePath** — Exclude artifacts where the path contains specific text, specified as a character vector. Example: "HLR"
- **InProject** — Include only artifacts that have been added to the project, specified as a numeric or logical 1 (true) or 0 (false). Example: true

Methods

run	<p>Return artifacts from query</p> <p>The run method inside this built-in query runs on a query object <code>obj</code> and returns artifacts <code>artifacts</code>. If you inherit from this built-in query, make sure to use the same method signature inside your custom query:</p> <pre>function artifacts = run(obj,~) ... end</pre>
-----	--

Use in Process Model

You can use this query in your process model to find artifacts for your task to iterate over (`IterationQuery`).

For example, suppose that you want the built-in task `padv.builtin.task.RunModelStandards` to only run for models that use the project label `RunModelAdvisor` from the project label category `ModelLabels`. You can change the `IterationQuery` for the task to specify a different set of artifacts for the task to run on. You can use the built-in query `padv.builtin.query.FindModelsWithLabel` to find the models that use that project label. Specify the first input argument as the project label category and the second argument as the project label name.

```
maTask = pm.addTask(padv.builtin.task.RunModelStandards());
maTask.IterationQuery = ...
    padv.builtin.query.FindModelsWithLabel("ModelLabels", "RunModelAdvisor");
```

Note You cannot use this query as an input query (`InputQueries`).

padv.builtin.query.FindModelsWithTestCases

This query returns each of the models in the project that are associated with a test case. You can use the optional name-value arguments to filter the results.

Syntax

`q = padv.builtin.query.FindModelsWithTestCases()` finds all models that are associated with a test case.

`q = padv.builtin.query.FindModelsWithTestCases(Name, Value)` find models that are associated with a test case and meet the criteria specified by one or more name-value arguments. For example, to find models that are associated with test cases and include `Control` in the full file path, specify `IncludePath="Control"`.

Input Arguments

Name-Value Arguments

- **Name** — Unique identifier for query, specified as character vector or string. Example: `"CustomQuery"`
- **IncludeLabel** — Find artifacts that have a specific project label, specified as a cell array where the first entry is the project label category and the second entry is the project label name. Example: `{"Classification", "Design"}`
- **ExcludeLabel** — Exclude artifacts that have a specific project label, specified as a cell array where the first entry is the project label category and the second entry is the project label name. Example: `{"Classification", "Design"}`
- **IncludePath** — Find artifacts where the path contains specific text, specified as a character vector or string. Example: `"Control"`
- **ExcludePath** — Exclude artifacts where the path contains specific text, specified as a character vector. Example: `"Control"`

Methods

run	<p>Return artifacts from query</p> <p>The <code>run</code> method inside this built-in query runs on a query object <code>obj</code> and returns artifacts associated with the artifact <code>iterationArtifact</code>. If you inherit from this built-in query, make sure to use the same method signature inside your custom query:</p> <pre>function artifacts = run(obj, iterationArtifact) ... end</pre>
-----	---

Use in Process Model

You can use this query in your process model to find artifacts for your task to iterate over (IterationQuery).

For example, suppose that you only want to run the **Merge Test Results** task for certain models that do not have `Control` in the file path. By default, the **Merge Test Results** task uses the built-in query `padv.builtin.query.FindModelsWithTestCases` as the `IterationQuery`. In the process model, you can change the `IterationQuery` for the task to:

- 1 Use the built-in query `padv.builtin.query.FindModelsWithTestCases` to find the models that are associated with a test case.
- 2 Specify the `ExcludePath` argument of the query to filter out any models that have `Control` in the file path.

```
mergeTestTask = pm.addTask(padv.builtin.task.MergeTestResults());  
mergeTestTask.IterationQuery = padv.builtin.query.FindModelsWithTestCases(...  
    ExcludePath = "Control");
```

Note You cannot use this query as an input query (InputQueries).

padv.builtin.query.FindProjectFile

This query returns the project file.

Syntax

`q = padv.builtin.query.FindProjectFile()` finds the project file.

Methods

run	<p>Return artifacts from query</p> <p>The run method inside this built-in query runs on a query object <code>obj</code> and returns artifacts <code>artifacts</code>. If you inherit from this built-in query, make sure to use the same method signature inside your custom query:</p> <pre>function artifacts = run(obj,~) ... end</pre>
-----	--

Use in Process Model

You can use this query in your process model to find artifacts that your tasks can iterate over (`IterationQuery`) or use as inputs (`InputQueries`).

For example, suppose that you have a custom task, `MyCustomTask`, that you want to run once for the project. You can use the built-in query `padv.builtin.query.FindProjectFile` to find the project file and specify the query as the `IterationQuery` for the custom task.

```
taskObj = addTask(pm, "MyCustomTask", ...
    IterationQuery = padv.builtin.query.FindProjectFile);
```

padv.builtin.query.FindRefModels

This query returns each of the referenced models in the project. You can use optional name-value arguments to filter the results.

Syntax

`q = padv.builtin.query.FindRefModels()` finds all reference models in the project.

`q = padv.builtin.query.FindRefModels(Name, Value)` find reference models that meet the criteria specified by one or more name-value arguments. For example, to find reference models that include `Control` in the full file path, specify `IncludePath="Control"`.

Input Arguments

Name-Value Arguments

- **Name** — Unique identifier for query, specified as character vector or string. Example: `"CustomQuery"`
- **IncludeLabel** — Find artifacts that have a specific project label, specified as a cell array where the first entry is the project label category and the second entry is the project label name. Example: `{"Classification", "Design"}`
- **ExcludeLabel** — Exclude artifacts that have a specific project label, specified as a cell array where the first entry is the project label category and the second entry is the project label name. Example: `{"Classification", "Design"}`
- **IncludePath** — Find artifacts where the path contains specific text, specified as a character vector or string. Example: `"Control"`
- **ExcludePath** — Exclude artifacts where the path contains specific text, specified as a character vector. Example: `"Control"`

Methods

run	<p>Return artifacts from query</p> <p>The <code>run</code> method inside this built-in query runs on a query object <code>obj</code> and returns artifacts <code>artifacts</code>. If you inherit from this built-in query, make sure to use the same method signature inside your custom query:</p> <pre>function artifacts = run(obj,~) ... end</pre>
-----	---

Use in Process Model

You can use this query in your process model to find artifacts for your task to iterate over (`IterationQuery`).

For example, suppose that you want the built-in task `padv.builtin.task.RunModelStandards` to only run on reference models in the project. You can change the `IterationQuery` for the task to specify a different set of artifacts for the task to run on. You can use the built-in query `padv.builtin.query.FindRefModels` to find the reference models.

```
maTask = pm.addTask(padv.builtin.task.RunModelStandards());
maTask.IterationQuery = ...
    padv.builtin.query.FindRefModels;
```

Note You cannot use this query as an input query (`InputQueries`).

padv.builtin.query.FindRequirements

This query returns each of the requirement sets (`.slreqx`) within the project. You can use optional name-value arguments to filter the results.

Syntax

`q = padv.builtin.query.FindRequirements()` finds all requirement sets in the project.

`q = padv.builtin.query.FindRequirements(Name, Value)` finds requirement sets that meet the criteria specified by one or more name-value arguments. For example, to find requirement sets that include `System` in the full file path, specify `IncludePath="System"`.

Input Arguments

Name-Value Arguments

- **Name** — Unique identifier for query, specified as character vector or string. Example: `"CustomQuery"`
- **IncludeLabel** — Find artifacts that have a specific project label, specified as a cell array where the first entry is the project label category and the second entry is the project label name. Example: `{"Level", "System"}`
- **ExcludeLabel** — Exclude artifacts that have a specific project label, specified as a cell array where the first entry is the project label category and the second entry is the project label name. Example: `{"Level", "System"}`
- **IncludePath** — Find artifacts where the path contains specific text, specified as a character vector or string. Example: `"System"`
- **ExcludePath** — Exclude artifacts where the path contains specific text, specified as a character vector. Example: `"System"`
- **InProject** — Include only artifacts that have been added to the project, specified as a numeric or logical 1 (`true`) or 0 (`false`). Example: `true`

Note If you specify `InProject` as `true`, you can no longer use the query as an input query.

Methods

run	<p>Return artifacts from query</p> <p>The run method inside this built-in query runs on a query object <code>obj</code> and returns artifacts <code>artifacts</code>. If you inherit from this built-in query, make sure to use the same method signature inside your custom query:</p> <pre>function artifacts = run(obj,~) ... end</pre>
-----	--

Use in Process Model

You can use this query in your process model to find artifacts that your tasks can iterate over (`IterationQuery`) or use as inputs (`InputQueries`).

For example, suppose that you have a custom task, `MyCustomTask`, that you add to your process model. You can use the built-in query `padv.builtin.query.FindRequirements` to find requirement sets in the project. If you specify `padv.builtin.query.FindRequirements` as the `IterationQuery` for the task, the task runs once for each requirement set in the project.

```
taskObj = addTask(pm, "MyCustomTask", ...
    IterationQuery = padv.builtin.query.FindRequirements, ...
    InputQueries = padv.builtin.query.GetIterationArtifact);
```

In this example, specifying `InputQueries` as `padv.builtin.query.GetIterationArtifact` allows the task to use the artifacts returned by `IterationQuery` as inputs to the task.

In Process Advisor, the requirement sets appear in the **Tasks** column.

padv.builtin.query.FindRequirementsForModel

This query returns each of the requirements associated with a model. You can use optional name-value arguments to filter the results.

Syntax

`q = padv.builtin.query.FindRequirementsForModel()` finds all requirements associated with models in the project.

`q = padv.builtin.query.FindRequirementsForModel(Name, Value)` find requirements that are associated with a model in the project and meet the criteria specified by one or more name-value arguments. For example, to find requirements that include `System` in the full file path, specify `IncludePath="System"`.

Input Arguments

Name-Value Arguments

- **Name** — Unique identifier for query, specified as character vector or string. Example: `"CustomQuery"`
- **IncludeLabel** — Find artifacts that have a specific project label, specified as a cell array where the first entry is the project label category and the second entry is the project label name. Example: `{"Level", "System"}`
- **ExcludeLabel** — Exclude artifacts that have a specific project label, specified as a cell array where the first entry is the project label category and the second entry is the project label name. Example: `{"Level", "System"}`
- **IncludePath** — Find artifacts where the path contains specific text, specified as a character vector or string. Example: `"System"`
- **ExcludePath** — Exclude artifacts where the path contains specific text, specified as a character vector. Example: `"System"`

Methods

run	<p>Return artifacts from query</p> <p>The run method inside this built-in query runs on a query object <code>obj</code> and returns artifacts that are associated with the artifact <code>iterationArtifact</code>. If you inherit from this built-in query, make sure to use the same method signature inside your custom query:</p> <pre>function artifacts = run(obj, iterationArtifact) ... end</pre>
-----	---

padv.builtin.query.FindTestCasesForModel

This query returns each of the test cases associated with a model. You can use optional name-value arguments to filter the results.

Note The query also finds test cases associated with subsystem references. A subsystem reference allows you to save the contents of a subsystem in a separate file and reference it using a Subsystem Reference block.

Syntax

`q = padv.builtin.query.FindTestCasesForModel()` finds test cases associated with a model.

`q = padv.builtin.query.FindTestCasesForModel(Name, Value)` finds test cases that are associated with a model and meet the criteria specified by one or more name-value arguments. For example, to find test cases that include HLR in the full file path, specify `IncludePath="HLR"`.

Input Arguments

Name-Value Arguments

- **Name** — Unique identifier for query, specified as character vector or string. Example: "CustomQuery"
- **IncludeLabel** — Find artifacts that have a specific project label, specified as a cell array where the first entry is the project label category and the second entry is the project label name. Example: {"Level", "HLR"}
- **ExcludeLabel** — Exclude artifacts that have a specific project label, specified as a cell array where the first entry is the project label category and the second entry is the project label name. Example: {"Level", "HLR"}
- **IncludePath** — Find artifacts where the path contains specific text, specified as a character vector or string. Example: "HLR"
- **ExcludePath** — Exclude artifacts where the path contains specific text, specified as a character vector. Example: "HLR"
- **Tags** — Only include test cases that use a specific test case tag or tags. Example: {"tag1", "tag2"}

Methods

run	<p>Return artifacts from query</p> <p>The run method inside this built-in query runs on a query object obj and returns artifacts artifacts that are associated with the artifact iterationArtifact. If you inherit from this built-in query, make sure to use the same method signature inside your custom query:</p> <pre>function artifacts = run(obj,iterationArtifact) ... end</pre>
-----	--

Use in Process Model

You can use this query in your process model to find artifacts that your tasks can iterate over (IterationQuery) or use as inputs (InputQueries).

For example, suppose that you want the **Run Tests** task to only run on test cases that use the specific test case tag TagA. You can use the built-in query padv.builtin.query.FindTestCasesForModel to find the test cases and the Tags input argument to have the query only return test cases that use the specified test case tag.

```
milTask = pm.addTask(padv.builtin.task.RunTestsPerTestCase());
milTask.IterationQuery = padv.builtin.query.FindTestCasesForModel(...
    Tags = "TagA");
```

If you need to include multiple instances of a task, you need to specify different Name values for each task.

```
% Run Tests for TagA
milTaskA = addTask(pm,padv.builtin.task.RunTestsPerTestCase(...
    Name = "RunTestsForTagA"));
milTaskA.Title = "Run Tests for TagA";
milTaskA.IterationQuery = padv.builtin.query.FindTestCasesForModel(...
    Tags = "TagA");

% Run Tests for TagB
milTaskB = pm.addTask(padv.builtin.task.RunTestsPerTestCase(...
    Name = "RunTestsForTagB"));
milTaskB.Title = "Run Tests for TagB";
milTaskB.IterationQuery = padv.builtin.query.FindTestCasesForModel(...
    Tags = "TagB");
```

padv.builtin.query.FindTopModels

This query returns each of the top models in the project. You can use optional name-value arguments to filter the results.

Syntax

`q = padv.builtin.query.FindTopModels()` finds all top models in the project.

`q = padv.builtin.query.FindTopModels(Name, Value)` find top models that meet the criteria specified by one or more name-value arguments. For example, to find top models that include `Control` in the full file path, specify `IncludePath="Control"`.

Input Arguments

Name-Value Arguments

- **Name** — Unique identifier for query, specified as character vector or string. Example: `"CustomQuery"`
- **IncludeLabel** — Find artifacts that have a specific project label, specified as a cell array where the first entry is the project label category and the second entry is the project label name. Example: `{"Classification", "Design"}`
- **ExcludeLabel** — Exclude artifacts that have a specific project label, specified as a cell array where the first entry is the project label category and the second entry is the project label name. Example: `{"Classification", "Design"}`
- **IncludePath** — Find artifacts where the path contains specific text, specified as a character vector or string. Example: `"Control"`
- **ExcludePath** — Exclude artifacts where the path contains specific text, specified as a character vector. Example: `"Control"`

Methods

run	<p>Return artifacts from query</p> <p>The <code>run</code> method inside this built-in query runs on a query object <code>obj</code> and returns artifacts <code>artifacts</code>. If you inherit from this built-in query, make sure to use the same method signature inside your custom query:</p> <pre>function artifacts = run(obj,~) ... end</pre>
-----	---

Use in Process Model

You can use this query in your process model to find artifacts that your tasks can iterate over (`IterationQuery`) or use as inputs (`InputQueries`).

For example, suppose that you want the built-in task `padv.builtin.task.RunModelStandards` to only run on top models in the project. By default, the **Check Modeling Standards** task uses the built-in query `padv.builtin.query.FindModels` as the `IterationQuery`. In the process model, you can change the `IterationQuery` for the task to:

- 1 Use the built-in query `padv.builtin.query.FindTopModels` to find the top models in the project.
- 2 Specify the `IncludePath` argument of the query to only include top models that have `Control` in the file path.

```
maTask = pm.addTask(padv.builtin.task.RunModelStandards());  
maTask.IterationQuery = ...  
    padv.builtin.query.FindTopModels(IncludePath = "Control");
```

For the Process Advisor example project, the model `Flight_Control.slx` appears under the task title in Process Advisor.

```
▼ ○ Check Modeling Standards  
    ○ Flight_Control.slx
```

padv.builtin.query.GetDependentArtifacts

This query returns the dependent artifacts for a given artifact.

Syntax

`q = padv.builtin.query.GetDependentArtifacts()` gets the dependent artifacts for a given artifact.

Methods

run	<p>Return artifacts from query</p> <p>The run method inside this built-in query runs on a query object <code>obj</code> and returns artifacts that are associated with the artifact <code>iterationArtifact</code>. If you inherit from this built-in query, make sure to use the same method signature inside your custom query:</p> <pre>function artifacts = run(obj,iterationArtifact) ... end</pre>
-----	--

Use in Task

You can use this query in your custom tasks to find artifacts that your tasks can use as inputs (InputQueries).

For example, the query `padv.builtin.query.GetDependentArtifacts` is often used as the `InputDependencyQuery` for a task. If you specify `padv.builtin.query.GetDependentArtifacts` as the `InputDependencyQuery` for a task, the query analyzes each input and finds any additional file dependencies.

```
classdef MyCustomTask < padv.Task
    methods
        function obj = MyCustomTask(options)
            arguments
                options.Name = "MyCustomTask";
                options.IterationQuery = "padv.builtin.query.FindModels";
                options.InputQueries = "padv.builtin.query.GetIterationArtifact";
                % For each input, find dependencies that can affect whether
                % task results are up-to-date
                options.InputDependencyQuery = padv.builtin.query.GetDependentArtifacts;
            end

            obj@padv.Task(options.Name,...
                IterationQuery=options.IterationQuery,...
                InputQueries=options.InputQueries,...
                InputDependencyQuery=options.InputDependencyQuery);
        end
        function taskResult = run(obj,input)
            taskResult = padv.TaskResult;
        end
    end
end
```

```
        taskResult.Status = padv.TaskStatus.Pass;  
    end  
end
```

When you run a task, the build system runs the `InputDependencyQuery` to find any additional dependencies that can affect whether task results are up-to-date.

Note You cannot use this query as an iteration query (`IterationQuery`).

padv.builtin.query.GetIterationArtifact

This query returns the artifact that the task is iterating over.

Syntax

`q = padv.builtin.query.GetIterationArtifact()` gets the artifact that the task is iterating over.

Methods

run	<p>Return artifacts from query</p> <p>The run method inside this built-in query returns the iteration artifact <code>iterationArtifact</code>. If you inherit from this built-in query, make sure to use the same method signature inside your custom query:</p> <pre>function artifact = run(~,iterationArtifact) artifact = iterationArtifact; end</pre>
-----	--

Use in Task

You can use this query in your custom tasks to find artifacts that your tasks can use as inputs (InputQueries).

For example, the query `padv.builtin.query.GetIterationArtifact` is often used as one of the input queries (InputQueries) for a task. If your `IterationQuery` is `padv.builtin.query.FindModels` and you specify `padv.builtin.query.GetIterationArtifact` as an input query for a task, the task considers the models in the project as inputs to the task.

```
classdef MyCustomTask < padv.Task
    methods
        function obj = MyCustomTask(options)
            arguments
                options.Name = "MyCustomTask";
                options.IterationQuery = "padv.builtin.query.FindModels";
                options.InputQueries = "padv.builtin.query.GetIterationArtifact";
            end

            obj@padv.Task(options.Name,...
                IterationQuery=options.IterationQuery,...
                InputQueries=options.InputQueries,...
                InputDependencyQuery=options.InputDependencyQuery);
        end
        function taskResult = run(obj,input)
            taskResult = padv.TaskResult;
            taskResult.Status = padv.TaskStatus.Pass;
        end
    end
end
```


When you run a task, the build system runs the `InputQueries` to find the inputs to the task.

Note You cannot use this query as an iteration query (`IterationQuery`).

padv.builtin.query.GetOutputsOfDependentTask

This query returns the outputs from the predecessor task.

Syntax

`q = padv.builtin.query.GetOutputsOfDependentTask()` gets the outputs from the predecessor task. You must define the predecessor task by using the function `dependsOn` on the task objects.

`q = padv.builtin.query.GetOutputsOfDependentTask(Task=taskName)` gets the outputs from the predecessor task specified by `taskName`.

`q = padv.builtin.query.GetOutputsOfDependentTask(Name = queryName, Task=taskName)` gets the outputs from the predecessor task specified by `taskName`. The query object gets the name specified by `queryName`. If you do not specify a query name, the query automatically generates a unique name based on the name of the predecessor task.

Input Arguments

Name-Value Arguments

- **Name** — Unique identifier for query, specified as character vector or string. Example: "CustomQuery"
- **Task** — Task name, specified as a character vector or string. Example: "padv.builtin.task.RunModelStandards"

Methods

run	<p>Return artifacts from query</p> <p>The run method inside this built-in query runs on a query object <code>obj</code> and returns artifacts <code>artifacts</code>. If you inherit from this built-in query, make sure to use the same method signature inside your custom query:</p> <pre>function artifacts = run(obj,~) ... end</pre>
-----	--

Use in Task

You can use this query in your custom tasks to find artifacts that your tasks can use as inputs (InputQueries).

For example, the query `padv.builtin.query.GetOutputsOfDependentTask` is often used as one of the input queries (InputQueries) for a task. If you open the source code for the **Merge Test Results** task, you can see that the task uses the built-in query `padv.builtin.query.GetOutputsOfDependentTask` as an input query.

```
open padv.builtin.task.MergeTestResults
```

```
...
options.InputQueries = [padv.builtin.query.GetIterationArtifact,...
    padv.builtin.query.GetOutputsOfDependentTask(Task="padv.builtin.task.RunTestsPerTestCase")];
options.InputDependencyQuery = padv.builtin.query.GetDependentArtifacts;
...
```

When you run the **Merge Test Results** task, the build system runs this input query, which passes the outputs of the **Run Tests** task as inputs to the **Merge Test Results** task.

Note Note that since the **Merge Test Results** task depends on data from the **Run Tests** task, the default process model uses the `dependsOn` function to explicitly specify the dependency relationship between the tasks.

```
if includeTestsPerTestCaseTask && includeMergeTestResultsTask
    mergeTestTask.dependsOn(milTask, "WhenStatus", {'Pass', 'Fail'});
end
```
