



ANSYS ACT Developer's Guide



ANSYS, Inc.
Southpointe
2600 ANSYS Drive
Canonsburg, PA 15317
ansysinfo@ansys.com
<http://www.ansys.com>
(T) 724-746-3304
(F) 724-514-9494

Release 18.2
August 2017

ANSYS, Inc. and
ANSYS Europe,
Ltd. are UL
registered ISO
9001: 2008
companies.

Copyright and Trademark Information

© 2017 ANSYS, Inc. Unauthorized use, distribution or duplication is prohibited.

ANSYS, ANSYS Workbench, AUTODYN, CFX, FLUENT and any and all ANSYS, Inc. brand, product, service and feature names, logos and slogans are registered trademarks or trademarks of ANSYS, Inc. or its subsidiaries located in the United States or other countries. ICEM CFD is a trademark used by ANSYS, Inc. under license. CFX is a trademark of Sony Corporation in Japan. All other brand, product, service and feature names or trademarks are the property of their respective owners. FLEXIm and FLEXnet are trademarks of Flexera Software LLC.

Disclaimer Notice

THIS ANSYS SOFTWARE PRODUCT AND PROGRAM DOCUMENTATION INCLUDE TRADE SECRETS AND ARE CONFIDENTIAL AND PROPRIETARY PRODUCTS OF ANSYS, INC., ITS SUBSIDIARIES, OR LICENSORS. The software products and documentation are furnished by ANSYS, Inc., its subsidiaries, or affiliates under a software license agreement that contains provisions concerning non-disclosure, copying, length and nature of use, compliance with exporting laws, warranties, disclaimers, limitations of liability, and remedies, and other provisions. The software products and documentation may be used, disclosed, transferred, or copied only in accordance with the terms and conditions of that software license agreement.

ANSYS, Inc. and ANSYS Europe, Ltd. are UL registered ISO 9001: 2008 companies.

U.S. Government Rights

For U.S. Government users, except as specifically granted by the ANSYS, Inc. software license agreement, the use, duplication, or disclosure by the United States Government is subject to restrictions stated in the ANSYS, Inc. software license agreement and FAR 12.212 (for non-DOD licenses).

Third-Party Software

See the [legal information](#) in the product help files for the complete Legal Notice for ANSYS proprietary software and third-party software. If you are unable to access the Legal Notice, contact ANSYS, Inc.

Published in the U.S.A.

Table of Contents

Introduction	1
Customizing ANSYS Products	2
Licensing	3
Migration Notes	4
Known Issues and Limitations	4
Summary: Extension Examples	5
Defining Extensions	6
Common Capabilities	6
Mechanical Capabilities	6
APIs Description	7
Mechanical Examples	7
DesignXplorer Examples	7
Custom Workflows in Workbench Examples	8
Wizard Examples	9
Defining Extensions	11
Creating a Scripted Extension	11
Creating the XML Extension Definition File	12
Creating the IronPython Script	13
Setting Up the Directory Structure	13
Viewing Exposed Custom Functionality	15
Script Processing	16
Using Extension Libraries	17
Building a Binary Extension	18
Configuring Extension Options	21
Additional Extension Folders Option	21
Save Binary Extensions with Project Option	22
Journal Wizard Actions Option	22
Debug Mode Option	22
Using Extensions	25
Using the ACT Start Page	25
Accessing the ACT Start Page	25
Accessing the ACT Start Page from Workbench	25
Accessing the ACT Start Page from AIM	26
Accessing the ACT Start Page from a Standalone Instance of Electronics Desktop	26
Accessing the ACT Start Page from a Standalone Instance of Fluent	27
Accessing the ACT Start Page from a Standalone Instance of SpaceClaim	28
ACT Start Page Interface	29
Using the Extension Manager	29
Extension Manager Accessed via the ACT Start Page	30
Extension Manager Accessed via Extensions Menu	32
Installing and Uninstalling Extensions	32
Installing a Scripted Extension	33
Uninstalling a Scripted Extension	33
Installing a Binary Extension	33
Uninstalling a Binary Extension	34
Loading and Unloading Extensions	35
Configuring Extensions to be Loaded by Default	37
Feature Creation Capabilities	41
Common Capabilities	41
Defining Toolbars and Toolbar Buttons	41

Binding Toolbar Buttons with ACT Objects	44
Defining Pop-up Dialog Boxes	45
Storing Data in Your Extension	46
ACT-Based Properties	47
Creating Property Groups	47
Using <PropertyGroup> and <PropertyTable> Blocks	47
Using Templates to Create Property Groups	51
Parameterizing Properties	51
Common Property Parameterization Processes	52
Parameterizing Properties in ANSYS Workbench	53
Parameterizing Properties in ANSYS Mechanical	54
Defining Input Parameters in ANSYS Mechanical	54
Defining Output Parameters in ANSYS Mechanical	55
Parameterizing Properties in ANSYS DesignModeler	56
Parameterizing Properties in a Third-Party Solver	58
Defining Parameters under a Load in a Third-Party Solver	58
Defining Parameters in Analysis Settings in a Third-Party Solver	58
Defining Parameters under Results in a Third-Party Solver	58
Defining DesignXplorer Properties	59
Properties in the DesignXplorer Interface	59
Additional Attributes for DesignXplorer Extensions	61
Advanced Usage Examples	61
Managing Dependencies between Properties	61
Controlling Property Visibility with a Callback	61
DOE Example	62
Optimization Example	62
Modifying an Attribute with a Callback	62
DOE Example	63
Optimization Example	63
Capabilities for ANSYS Mechanical	63
Adding a Preprocessing Feature in ANSYS Mechanical	64
Adding a Postprocessing Feature in ANSYS Mechanical	70
Creating Results with Imaginary Parts	74
Responding to a Change to the Active Unit System	74
Obsolete Callbacks "OnStartEval" and "GetValue"	75
Connecting to a Third-Party Solver	75
Third-Party Solver Connection Extension	75
Post Processing	80
Load and Save Data	81
Capabilities for ANSYS DesignModeler	84
Geometry Definition in the XML Extension Definition File	84
Geometry Definition in the IronPython Script	86
Functions Associated with the Callback <congenerate>	87
Functions Associated with the Callback <onaftergenereate>	89
Capabilities for ANSYS DesignXplorer	90
The Design Exploration Process	91
DesignXplorer Systems and Component	92
Design of Experiments Component	93
Optimization Components	93
DesignXplorer Extensions	94
Implementing a DesignXplorer Extension	94
Implementation Requirements	94

DesignXplorer Extension Definition and Configuration	94
DesignXplorer Extension Capabilities	97
Main Capabilities	97
Optional Capabilities	98
Notes on Method Class Implementation	99
Notes on Monitoring	99
Notes on Results	100
Capabilities for ANSYS AIM	100
Adding a Preprocessing Feature in ANSYS AIM Structural	101
Custom Load Definition in the XML Extension Definition File	101
Custom Load Definition in the IronPython Script	102
Creating a Custom Object to Merge Existing AFD Features (Process Compression)	104
CFD System Definition in the XML Extension Definition File	104
CFD System Definition in the IronPython Script	105
Using SetScriptVersion Within ACT Extensions for ANSYS AIM	107
Capabilities for ANSYS Fluent	108
Simulation Workflow Integration	109
Exposing Custom Task Groups and Tasks on the Project Schematic	109
Using Global Callbacks	110
Invoking Custom Schematic Actions	110
Invoking Custom Workbench Actions	111
Global Callback Observation Process	111
Progress Monitoring	112
Using Process Utilities	114
The Custom Workflow Creation Process	115
Creating the XML Definition File	116
Defining a Global Callback	118
Invoking Global Callbacks	119
Filtering Global Callbacks	119
Specifying Global Callback Execution Order	120
Available Schematic Callbacks	120
Available Workbench Callbacks	123
Defining a Task	124
Defining Task-Level Attributes	125
Defining Task-Level General Data Transfer	125
Defining Task-Level Data Transfer Access	127
Task-Level File Management Capabilities	129
Defining Task-Level Callbacks	129
Defining Task-Level Context Menus	130
Defining Task-Level Property Groups and Properties	130
Defining Task-Level Property Callbacks	131
Using Standardized Property Interaction	135
Using Path-Based Property Control Support	135
Using String-Based and Option-Based Task Properties	136
Specifying Property Default Values	136
Defining Task-Level Parameters	137
Defining Task-Level Inputs and Outputs	137
Defining a Remote Job Execution Specification	139
Accessing and Invoking Task-Related Workbench Data	139
Executing Container-Level Commands on a Task	139
Accessing Workbench Parameters from a Task	140
Accessing a Workbench-Based Component from a Task	140

Accessing the Owning Task Group from a Task	140
Defining a Task Group	141
Creating the IronPython Script	143
Upstream Data Consumption (Input)	143
Data Generation (Output)	143
Convenience APIs	144
Simulation Wizards	147
Types of Wizards	147
Creating Wizards	148
Parts of a Wizard	148
XML Extension Definition File	149
Example: XML Extension Definition File	149
IronPython Script	154
Example: IronPython Script for a Project Wizard	154
Custom Help for Wizards	159
Viewing Custom Help for Non-AIM Wizards	160
Viewing Custom Help for AIM Wizards	160
Creating Conditional Steps	161
Customizing the Layout of a Wizard	162
Installing and Loading Wizards	164
Using Wizards	164
Using a Workbench Project or Non-AIM Target Product Wizard	165
Launching a Wizard from the Wizards Page	165
The Wizard Interface	166
Entering Data in a Wizard	167
Exiting a Wizard or Project	168
Using an AIM Target Product Wizard	168
Creating a Custom Template	169
Creating a Guided Simulation	169
Creating a Guided Setup	171
APIs Description	173
ACT Automation API	173
APIs for Custom Workflows in the Workbench Project Page	173
Accessing Project-Level APIs	174
Accessing Task APIs	174
Accessing Task Template APIs	177
Accessing Task Group APIs	180
Accessing Task Group Template APIs	182
Accessing Parameter APIs	183
Accessing Property APIs	184
Accessing State-Handling APIs	184
Accessing Project Reporting APIs	187
APIs for ANSYS Mechanical	192
Directly Accessing an Object	193
Handling Property Types	193
Customizing the User Interface and Existing Toolbars	194
Working with Command Snippets	194
Manipulating Fields for Boundary Conditions	195
Input and Output Variables	195
Variable Definition Types	195
Input Loading Data Definitions	195
Setting the Variable Definition Type	197

Creating a Displacement and Verifying Its Variable Definition Types	197
Changing the Y Component from Free to Discrete and Verifying	198
Changing the Y Component Back to Free and Verifying	199
Setting Discrete Values for Variables	200
Getting and Setting Discrete Values for an Input Variable	200
Getting and Setting Discrete Values for an Output Variable	202
Mechanical Worksheets	203
Named Selection Worksheets	203
Creating the Named Selection Worksheet	203
Adding New Rows	204
Mesh Order Worksheets	206
Displaying the Mesh Worksheet	207
Specifying the Mesh Worksheet and Named Selections	208
Adding New Rows	208
Meshing the Named Selections	209
Additional Commands	210
Layered Section Worksheets	211
Displaying the Layered Section Worksheet	211
Getting the Layered Section Worksheet and Its Properties	211
Getting and Displaying the Material Property	211
Getting and Displaying the Thickness Property	211
Getting and Displaying the Angle Property	212
Setting and Displaying the Thickness Property	212
Setting and Displaying the Angle Property	212
Bushing Joint Worksheets	212
Displaying the Bushing Joint Worksheet	212
Getting the Bushing Joint Worksheet and Its Properties	212
Getting the Value for a Specified Unit	213
Tree Object	213
Model Object	215
Geometry: Point Mass	216
Geometry: Export Geometry Object to an STL File	216
Mesh: Mesh Control	217
Connections: Frictionless Contact and Beam	217
Analysis: Load Magnitude	217
Analysis: Extract Min-Max Tabular Data for a Boundary Condition	219
Result: Total Deformation Maximum	220
TraverseExtension	221
Traversing the Geometry	221
Traversing the Mesh	223
Traversing Results	224
Graphical Views	226
Getting the ModelViewManager Data Object	227
Setting the View	227
Creating a View	227
Deleting a View	228
Applying a View	228
Renaming a View	229
Rotating a View	229
Saving a View	229
Saving an Object	230
Importing a Saved View List	230

Exporting a Saved View List	230
Exporting an Object	231
APIs for ANSYS DesignModeler	231
Using the Selection Manager in DesignModeler	231
Working with the Current Selection	231
Creating a New Selection and Adding Entities	232
Creating Primitives	232
Creating a Sheet Body	233
Creating a Wire Body	234
Creating a Solid Body	235
Applying Operations	235
Applying the Extrude Operation	236
Applying the Transform Edges to Wire Tool	237
APIs for ANSYS DesignXplorer	238
DOE APIs	238
DOE Architecture	238
Sampling Process	239
Optimization APIs	240
Optimization Architecture	240
Optimization Process	241
Associated Libraries	243
Query to Material Properties	243
Units Conversion	245
MAPDL Helpers	247
Journaling Helper	248
Development and Debugging	251
ACT Console	251
Using the Scope Menu	252
Using the Command Line	252
Using the Command History	252
Using Autocompletion	255
Using Autocompletion Tooltips	256
Interacting with Autocompletion Suggestions	258
Using Snippets	258
ACT Console Keyboard Shortcuts	263
Binary Extension Builder	265
Debug Mode	265
Toolbar Buttons for Reloading Extensions	266
Extensions Log File	267
Debugging with Microsoft® Visual Studio	268
Advanced Programming in C#	269
Initialize the C# Project	269
C# Implementation for a Load	269
C# Implementation for a Result	270
Extension Examples	273
Mechanical Extension Examples	273
Von-Mises Stress as a Custom Result	273
Edge-Node Coupling Tool	278
DesignXplorer Extension Examples	282
DOE Extension Examples	282
Optimization Extension Examples	283
Custom ACT Workflows in Workbench Examples	283

Global Workflow Callbacks	284
XML Extension Definition File	284
IronPython Script	285
Custom User-Specified GUI Operation	287
XML Extension Definition File	287
IronPython Script	288
Custom, Lightweight, External Application Integration with Parameter Definition	288
XML Extension Definition File	289
IronPython Script	290
Custom, Lightweight, External Application Integration with Custom Data and Remote Job Execution	291
XML Extension Definition File	292
IronPython Script	294
Using RSM Job Submission Capabilities	296
Generic Material Transfer	298
XML Extension Definition File	299
IronPython Script	300
Material File	301
Generic Mesh Transfer	302
XML Extension Definition File	303
IronPython Script	304
Custom Transfer	305
XML Extension Definition File	305
IronPython Script	306
Parametric	307
XML Extension Definition File	307
IronPython Script	308
Wizard Examples	308
Project Wizard (Workbench Project Tab)*	308
XML Extension Definition File	308
IronPython Script	313
Project Wizard (AIM Project Tab)	317
XML Extension Definition File	317
IronPython Script	319
Reviewing Wizard Results	322
DesignModeler Wizard*	325
XML Extension Definition File	325
IronPython Script	328
SpaceClaim Wizard*	329
XML Extension Definition File	329
IronPython Script	332
Mechanical Wizard*	333
XML Extension Definition File	333
IronPython Script	336
Mixed Wizard*	337
XML Extension Definition File	338
IronPython Script	351
Fluent Wizard (MSH Input File)	351
XML Extension Definition File	351
Reviewing the Analysis	355
Fluent Wizard (CAS Input File)	358
XML Extension Definition File	358

Reviewing the Analysis	362
Electronics Desktop Wizard	365
XML Extension Definition File	365
IronPython Script	367
Reviewing the Analysis	369
SpaceClaim Wizard	371
XML Extension Definition File	371
IronPython Script	374
AIM Custom Template (Single-Step)	379
XML Extension Definition File	379
IronPython Script	382
Defining Custom Help for the Single-Step Custom Template	383
AIM Custom Template (Multiple-Step)	384
XML Extension Definition File	384
IronPython Script	388
Defining Custom Help for the Multiple-Step Custom Template	390
Wizard Custom Layout Example	391
XML Extension Definition File	391
IronPython Script	392
Exposure of the Wizard Layouts	393
XML Extension Definition	395
<extension>	396
<Application>	397
<appStoreId>	398
<assembly>	398
<author>	399
<description>	399
<Guid>	399
<Interface>	400
<Licenses>	402
<script>	402
<simdata>	403
<Templates>	405
<UIDefintion>	405
<Wizard>	405
<Workflow>	406
A. Component Input and Output Tables	411
B. ANSYS Workbench Internally Defined System Template and Component Names	471
C. Data Transfer Types	489
D. Addin Data Types and Data Transfer Formats	493
E. Pre-Installed Custom Workflow Support	497

Introduction

ANSYS ACT is the unified and consistent tool for the customization and expansion of ANSYS products. Using ACT, you can create *extensions* to customize these products:

- Workbench
- Mechanical
- AIM
- DesignModeler
- DesignXplorer
- Electronics Desktop
- Fluent
- SpaceClaim

This guide offers an introduction to ACT, describing its capabilities and showing how to use them. While primarily intended for developers of ACT extensions, this guide also provides information for end users who manage and use extensions.

Basic extension examples demonstrate various types of ANSYS product customizations. Each of these extensions is written and tested on supported Windows platforms. For descriptions of all referenced extension examples, see [Summary: Extension Examples \(p. 5\)](#).

Note

- The development of extensions requires some knowledge of IronPython and XML. For extensions that customize the ANSYS solver, knowledge of APDL is also required.
- Extension examples are included in packages that you can download from the [ACT Resources](#) page on the [ANSYS Customer Portal](#). For more information, see [Extension Examples \(p. 273\)](#).

Related Topics:

- [Customizing ANSYS Products](#)
- [Licensing](#)
- [Migration Notes](#)
- [Known Issues and Limitations](#)
- [Summary: Extension Examples](#)

Customizing ANSYS Products

ACT enables customizations at the ANSYS product level, providing internal mechanisms that allow you to customize Workbench, AIM, and other supported ANSYS products without needing to compile external code or link with existing ANSYS libraries. Additionally, ACT enables you to manage the interfaces between these ANSYS products and the additional customizations, ensuring that they interact accurately.

ACT Customization Capabilities

ACT provides three types of customization: feature creation, simulation workflow integration, and process compression.

Feature Creation

Feature creation is the direct, API-driven customization of ANSYS products. In addition to leveraging the functionality already available in a product, ACT enables you to add functionality and operations of your own. Examples of feature creation include the creation of custom loads and geometries, the addition of custom preprocessing or postprocessing features, and the integration of third-party solvers, sampling methods, and optimization algorithms. The following products support feature creation: Workbench, Mechanical, DesignModeler, DesignExplorer, and AIM.

- For information on customization options that are generally applicable, such as the creation of toolbars and message dialog boxes, see [Common Capabilities \(p. 41\)](#).
- For information on the creation of custom properties and property groups, see [ACT-Based Properties \(p. 47\)](#).
- For information on Mechanical customizations such as specialized loads, results, and postprocessing functionality, see [Capabilities for ANSYS Mechanical \(p. 63\)](#).
- For information on DesignModeler customizations such as the creation of specialized geometries with custom properties and the application of various operations to the geometries, see [Capabilities for ANSYS DesignModeler \(p. 84\)](#).
- For information on DesignXplorer customizations such as the ability to integrate external sampling methods and optimization algorithms, see [Capabilities for ANSYS DesignXplorer \(p. 90\)](#).
- For information on AIM customizations such as exposing preprocessing functionality and creating custom objects, see [Capabilities for ANSYS AIM \(p. 100\)](#).

Simulation Workflow Integration

Simulation workflow integration is the incorporation of external knowledge such as apps, processes, and scripts into the ANSYS ecosystem. Using ACT, you can cusotmize simulation workflows in the Workbench **Project** page. For example, you can create custom *task groups* (systems) and custom *tasks* (components) for insertion in the **Project Schematic**, constructing consistent and cohesive simulation workflows that allow your business-specific elements to coexist and interface with pre-built ANSYS solutions. Currently, Workbench is the only ANSYS product to support simulation workflow integration. For more information, see [Simulation Workflow Integration \(p. 109\)](#) and [APIs for Custom Workflows in the Workbench Project Page \(p. 173\)](#).

Process Compression

Process compression is the encapsulation and automation of existing processes available in an ANSYS product. The result is a *wizard* that provides guidance within the product workflow, walking the end-user step-by-step through a simulation. You can create "single" wizards to be run in specific target products or "mixed" wizards to be run across several target products. The following products support process compression: Workbench, Mechanical, AIM, DesignModeler, DesignExplorer, Electronics Desktop, Fluent, and SpaceClaim.

Wizards allow you to leverage both the existing functionality of the targeted ANSYS products and the scripting capabilities of the Workbench and AIM framework API. With a wizard, you can manipulate existing features and simulation components, organizing them as needed to produce a custom automated process. You can also customize the user interface of a wizard that is not targeted for AIM. In addition to defining a layout for the interface as a whole, you can define layouts for individual interface components. Wizards can be applied to a single ANSYS product or across several ANSYS products. The degree of automation possible depends on the product being customized.

- For information on using wizards in general, see [Simulation Wizards \(p. 147\)](#).
- For examples of wizards for specific ANSYS products, see [Wizard Examples \(p. 308\)](#).
- For information on customizing the interface of a non-AIM wizard, see [Customizing the Layout of a Wizard \(p. 162\)](#) and the example [Wizard Custom Layout Example \(p. 391\)](#).
- For information on wizards for AIM, see [Using an AIM Target Product Wizard \(p. 168\)](#).

Where to Start

Guidance follows for topics that help you to start developing and using ACT extensions.

- To begin developing extensions, start with the general instructions provided in [Defining Extensions \(p. 11\)](#).
- To begin using an extension that you've created, review the general information provided in [Using Extensions \(p. 25\)](#).
- To review sample extensions for supported ANSYS products, see [Extension Examples \(p. 273\)](#).

For your convenience, [Summary: Extension Examples \(p. 5\)](#) describes all referenced extension examples.

Licensing

ACT development is licensed, but binary extension usage is not licensed. This means that a developer creating a new extension must have a license. However, once the extension is built, a license is not required to run it.

The ACT license enables two main capabilities, which are:

- The ability to build an extension in a binary format (WBEX file). The ACT license is checked out when the build is launched and is released once the build has been completed.

- The ability to load a scripted extension. As a consequence, only binary extensions can be loaded when no ACT license is available. No matter the number of scripted extensions that are loaded, only one ACT license is checked out. This license is released once all scripted extensions are unloaded from the **Extension Manager**.
-

Note

If an ACT license is already checked out by a loaded scripted extension, the build operation does not require a second ACT license to run.

Migration Notes

This section typically provides information that you need when migrating your ACT extensions. However, for migration from 18.1 to 18.2, no items have been identified.

Note

It is possible that additional items are published in the standalone *ANSYS ACT 18.2 Migration Notes* document. You can display this document from the **ACT Resources** page on the [ANSYS Customer Portal](#). To display this page, select **Downloads > ACT Resources**. To display the document, expand the **Help & Support** section and click **ACT Migration Notes** under **Migration**.

Known Issues and Limitations

This section provides a list of known issues and limitations in the ACT 18.2 release.

Graphic API Issues in DesignModeler and Mechanical when No Extensions are Loaded

There are some limitations on the Graphic API in Mechanical and DesignModeler when no extensions are loaded. For instance, the Factory2D does not work. It is therefore advised to have one or more extensions loaded prior to the usage of the Graphic API.

Launching the ACT Start Page on Linux can cause an unexpected shutdown

For a given configuration, launching the **ACT Start Page** on Linux causes an unexpected shutdown. Because this problem cannot be duplicated on testing machines, a problematic configuration rather than a code issue is suspected.

Localization Support Limitation

Localization of ACT is limited to the languages currently supported in ANSYS Workbench. This limitation does not apply to the ability to manage various languages within the extension. For example, the property names created by an extension do not have to be in the same language as the current activated language in ANSYS Workbench.

There is no mechanism to integrate localization for the property names defined by an extension. To manage different languages for your property names, you must develop localization yourself. Both regional settings based on the "." or the "," decimal symbol are available. However, the implementation of the extension should use the "." symbol for any value defined at the XML or IronPython level.

Automation API UI Thread and Performance

When you are using the ACT automation API, automation methods and properties must be called in the UI thread. Before 18.1, error messages or unexpected shutdowns were the potential results of the call of the automation API from a callback that was not executed in the UI thread.

As of 18.1, this issue is solved by automatically redirecting every call to the automation API in the UI thread. While you will no longer experience error messages or unexpected shutdowns, automatic redirection can potentially decrease performance. Poor performance is most likely to occur when you repeatedly call the automation API many times, such as when looping through nodes. If you notice a significant decrease in performance, you can speed up the process by using one of the following commands to bypass the multiple switches to the UI thread:

- `ExtAPI.Application.InvokeUIThread(func, arg)` // This executes the function func with the argument arg in the UI thread.
- `ExtAPI.Application.InvokeUIThread(func)` // This executes the function func in the UI thread.

A usage example follows.

```
def get_all_object_names(name_list):
    """ Returns the given list populated with all the names of the objects in the tree. """
    for obj in ExtAPI.DataModel.Tree.AllObjects:
        name_list.Add(obj.Name)
    return name_list

name_list = list()
all_object_names = ExtAPI.Application.InvokeUIThread(get_all_object_names, name_list)
```

Important

Due to technical reasons, the namespace of the automation API object has changed. Therefore, you should not use the method `GetType()` as a way to verify the nature of the object that you have. Instead, you must use the property `DataModelObjectCategory`, which is available on all objects. It returns an `enum` value indicating the type of the object. For example, the following command returns true:

```
ExtAPI.DataModel.Project.DataModelObjectCategory == DataModelObjectCategory.Project
```

Summary: Extension Examples

The following tables list all example extensions chronologically, noting the documentation sections in which they are referenced.

[Defining Extensions](#)

[Common Capabilities](#)

[Mechanical Capabilities](#)

[APIs Description](#)

[Mechanical Examples](#)

[DesignXplorer Examples](#)

[Custom Workflows in Workbench Examples](#)

[Wizard Examples](#)

Note

The package **ACT Developer's Guide Examples** contains most of the extensions described in this guide. Template packages contain a few additional examples. You download these packages from the [ACT Resources](#) page on the [ANSYS Customer Portal](#). For more information, see [Extension Examples \(p. 273\)](#).

Defining Extensions

Section	ANSYS Product	Extension	Description
Creating a Scripted Extension (p. 11)	Mechanical	ExtSample1	Demonstrates how a scripted extension is created. This extension example adds a toolbar and button to the user interface of Mechanical.

Common Capabilities

Section	ANSYS Product	Extension	Description
Common Capabilities (p. 41) > Defining Toolbars and Toolbar Buttons (p. 41)	Mechanical, but applicable to others	ExtToolbarSample	Adds a toolbar with three buttons to the user interface of Mechanical.
Common Capabilities (p. 41) > Defining Pop-up Dialog Boxes (p. 45)	Mechanical, but applicable to others	AdvancedProperties	Adds a property group and a time-dependent worksheet (property table) to the user interface of Mechanical.
ACT-Based Properties (p. 47) > Using <PropertyGroup> and <PropertyTable> Blocks (p. 47)	Mechanical, but applicable to others	ExtDialogSample	Adds a message dialog box to the user interface of Mechanical.

Mechanical Capabilities

Section	ANSYS Product	Extension	Description
Adding a Preprocessing Feature in ANSYS AIM Structural (p. 101)	Mechanical	DemoLoad	Shows how to add a preprocessing feature. This extension example adds a toolbar and buttons to create a generic load in Mechanical.
Adding a Postprocessing Feature in ANSYS Mechanical (p. 70)	Mechanical	DemoResult	Shows how to add a post-processing feature. This extension example adds a toolbar and buttons to create a customized result in Mechanical.
Connecting to a Third-Party Solver (p. 75)	Mechanical	DemonstrationSolver	Shows how to add a connection to a third-party solver. This extension example provides the ability to launch an external process (instead of an ANSYS solver) from Mechanical.

APIs Description

Section	ANSYS Product	Extension	Description
TraverseExtension (p. 221)	Mechanical	TraverseExtension	Shows how to traverse the nodes in the Mechanical tree. This extension example queries geometry, mesh, simulation, and results properties in Mechanical.

Mechanical Examples

Section	ANSYS Product	Extension	Description
Von-Mises Stress as a Custom Result (p. 273)	Mechanical	Mises	Shows how to define a custom result. This extension example adds a toolbar and button to create a customized result in Mechanical.
Edge-Node Coupling Tool (p. 278)	Mechanical	Coupling	Shows how to create a tool for coupling two sets of nodes related to two edges. This extension example adds a toolbar and button to couple the nodes in Mechanical.

DesignXplorer Examples

DesignXplorer extension examples are contained in the package **ACT Templates for DesignXplorer**, which you can download from the [ACT Resources](#) page on the [ANSYS Customer Portal](#). For more information, see [Extension Examples \(p. 273\)](#).

Section	ANSYS Product	Extension	Description
DOE Extension Examples (p. 282)	DesignXplorer	PythonSampling	Shows how to define sampling features in an extension using IronPython. This extension example runs a simple random exploration of the parametric space, generating the requested number of points.
DOE Extension Examples (p. 282)	DesignXplorer	CSharpSampling	Shows how to completely set up a sampling extension when the interface ISamplingMethod is implemented in C#. A Microsoft Visual Studio 2010 project is provided.
Optimization Extension Examples (p. 283)	DesignXplorer	PythonOptimizer	Shows how to define optimization features in an extension using IronPython. This extension example runs a simple random exploration of the parametric space, generating the requested number of points and returning the best candidates identified.
Optimization Extension Examples (p. 283)	DesignXplorer	CSharpOptimizer	Shows how to completely set up an optimization extension when the interface IOptimizationMethod is implemented in C#. A Microsoft Visual Studio 2010 project is provided.

Section	ANSYS Product	Extension	Description
Optimization Extension Examples (p. 283)	DesignXplorer	CppOptimizer	Shows how an extension can be implemented from existing C/C++. The interface IOptimizationMethod is implemented in IronPython as a wrapper to the C++ symbols, using the ctypes foreign function library for Python.

Custom Workflows in Workbench Examples

Section	ANSYS Product	Extension	Description
Global Workflow Callbacks (p. 284)	Workbench Project Page	WorkflowCallbacksDemo	Shows how to use global callbacks to implement a custom process or action before or after a Workbench Project Schematic operation. This extension example adds pre- and post-operation callbacks for each available operation.
Custom User-Specified GUI Operation (p. 287)	Workbench Project Page	EmptyGUI	Shows how to implement a custom GUI operation for a task. This extension example adds a custom context menu to the Workbench schematic.
Custom, Lightweight, External Application Integration with Parameter Definition (p. 288)	Workbench Project Page	Squares	Shows how to use parameter definitions to integrate an external application. This extension example uses parameters to drive an external application in squaring an input number. The result is displayed in the Workbench Parameter Set bar.
Custom, Lightweight, External Application Integration with Custom Data and Remote Job Execution (p. 291)	Workbench Project Page	DataSquares	Shows how to use custom task properties to integrate an external application and submit a task to the ANSYS Remote Solve Manager. This extension example uses custom task properties to drive an external application in squaring an input number. The result is displayed in the Workbench Parameter Set bar.
Generic Material Transfer (p. 298)	Workbench Project Page	GenericMaterialTransfer	Shows how to implement two custom material transfer task groups, with each passing material data to a downstream task group or task.
Generic Mesh Transfer (p. 302)	Workbench Project Page	GenericMeshTransfer	Shows how to implement two custom mesh transfer task groups, with each including "consuming" and "providing" connections.
Custom Transfer (p. 305)	Workbench Project Page	CustomTransfer	Shows how to implement a custom transfer from a producing task group to a consuming task group, creating connections between custom tasks. This

Section	ANSYS Product	Extension	Description
			extension example also illustrates the creation of single-task vs. multi-task task group.
Parametric (p. 307)	Workbench Project Page	Parametric	Shows how to create a parametric task group using the attribute isparametric-group .

Wizard Examples

The wizard examples with an asterisk are all defined in the same extension, **WizardDemos**. All other wizard examples are defined in separate extensions.

Section	ANSYS Product	Extension/Wizard	Description
Project Wizard (Workbench Project Tab)* (p. 308)	Workbench	WizardDemos/ProjectWizard	Shows how to create a wizard to be run from the Workbench Project tab.
Project Wizard (AIM Project Tab) (p. 317)	AIM	PressureLoss/PressureLoss	Shows how to create a wizard to be run from the AIM Project tab.
DesignModeler Wizard* (p. 325)	DesignModeler	WizardDemos/CreateBridge	Shows how to create a wizard to be run from DesignModeler.
Mechanical Wizard* (p. 333)	Mechanical	WizardDemos/SimpleAnalysis	Shows how to create a wizard to be run from Mechanical.
Mixed Wizard* (p. 337)	Workbench, Mechanical, DesignModeler, SpaceClaim	WizardDemos/BridgeSimulation	Shows how to create a wizard to be run across multiple ANSYS products.
Fluent Wizard (MSH Input File) (p. 351)	Fluent	FluentDemo1/Simple Analysis (Fluent Demo 1)	Shows how to create a wizard that imports a MSH file, runs a simple analysis, generates results, and displays the results in a customized panel in the user interface.

Section	ANSYS Product	Extension/Wizard	Description
Fluent Wizard (MSH Input File) (p. 351)	Fluent	FluentDemo2/Simple Analysis (Fluent Demo 2)	Shows how to create a wizard that imports a CAS file, runs a simple analysis, generates results, and displays the results in a customized panel in the user interface.
Electronics Desktop Wizard (p. 365)	Electronics Desktop	WizardDemo/ED Wizard Demo	Shows how to create a wizard to be run from Electronics Desktop.
SpaceClaim Wizard (p. 371)	SpaceClaim	SC_BGA_Extension/BGAWizard	Shows how to create a wizard to be run from SpaceClaim.
AIM Custom Template (Single-Step) (p. 379)	AIM	StudyDemo/StudyDemo1	Shows how to create a single-step custom template (wizard) to be run from AIM.
AIM Custom Template (Multiple-Step) (p. 384)	AIM	PressureLossMultiple/PressureLossMultiple	Shows how to create a multiple-step custom template (wizard) to be run from AIM.
Wizard Custom Layout Example (p. 391)	Workbench, but applicable to others	CustomLayout/CustomLayoutWizard	Shows how to create an ACT wizard with customized wizard-level and component-level layouts.

Defining Extensions

With ACT, the extensions that you create can exist in two different formats.

Extension Format	Description
Scripted extension	A scripted extension consists of an XML extension definition file and an IronPython script. It can also include supporting files, such as custom help or input files. Callbacks in the XML file invoke the functions or methods defined in the script.
Binary extension	A binary extension is an extension that has been compiled into a WBEX file. A binary extension can be shared with users who can execute the extension. However, they cannot view or edit the contents of the compiled WBEX file.

An ACT license is required to create and build extensions. However, an ACT license is not required to execute extensions. For more information, see [Licensing \(p. 3\)](#).

Note

When creating an extension, you use the IronPython language to develop the functions that execute the extension. The following links provide documentation on the IronPython language:

- [IronPython documentation](#)
 - [Python reference book](#)
-

Related Topics:

[Creating a Scripted Extension](#)
[Building a Binary Extension](#)
[Configuring Extension Options](#)

Creating a Scripted Extension

An ACT scripted extension has two basic components:

- **XML extension definition file:** Defines and configures the content of the extension.
- **IronPython script:** Defines the functions that are invoked by user interactions and implements the extension's behavior.

An extension can potentially be created using additional components such as external Python libraries or even C# code. However, the two components described above represent the basic definition for an extension.

This section uses a very simple example to demonstrate how to define a scripted extension. The extension **ExtSample1** customizes ANSYS Mechanical, adding a single-button toolbar to its interface. When this

button is clicked, a dialog box appears and displays the following message: **High five! ExtSample1 is a success!**

Depending on the ANSYS product to be customized, the content of the extension differs, but the basic organization of an ACT extension remains consistent with the example presented.

Note

The extension **ExtSample1** is included in the package **ACT Developer's Guide Examples**, which can be downloaded from the [ACT Resources](#) page on the ANSYS Customer Portal. For more information, see [Extension Examples](#) (p. 273).

Related Topics:

[Creating the XML Extension Definition File](#)
[Creating the IronPython Script](#)
[Setting Up the Directory Structure](#)
[Viewing Exposed Custom Functionality](#)
[Script Processing](#)
[Using Extension Libraries](#)

Creating the XML Extension Definition File

The XML extension definition file defines and configures the extension.

The file `ExtSample1.xml` for the extension `ExtSample1` follows.

```
extension version="1" name="ExtSample1">
  <guid shortid="ExtSample1">2cc739d5-9011-400f-ab31-a59e36e5c595</guid>
  <script src="sample1.py" />
  <interface context="Mechanical">
    <images>images</images>
    <callbacks>
      <oninit>init</oninit>
    </callbacks>
    <toolbar name="ExtSample1" caption="ExtSample1">
      <entry name="HighFive" icon="hand">
        <callbacks>
          <onclick>HighFiveOut</onclick>
        </callbacks>
      </entry>
    </toolbar>
  </interface>
</extension>
```

Key elements of this XML file are:

- Extension name
- Extension GUID
- IronPython script name
- Targeted product specified for `interface context`
- Callback `oninit` method name
- Toolbar and toolbar button definition

A GUID is a unique identifier for the extension that enables you to change the extension name in a new version without compromising the compatibility with projects that were saved with an older version of the extension.

The `<entry>` block defines the **HighFive** toolbar button, which uses the icon image from the file `hand.bmp`. The callback `<onclick>` defines the name of the function to invoke when the toolbar button is selected. The next section addresses the IronPython script defined for the function `HighFiveOut`.

Note

- For images to display as toolbar buttons or next to menu commands, Mechanical requires BMP files. ANSYS Workbench and all other ANSYS products require PNG files.
- To designate a transparent background for a BMP file, in an image editor, set the background color to 192, 192, 192 and then save the file with 256 colors.

For more information on working with XML extension definition files, see [Using Extensions \(p. 25\)](#).

Creating the IronPython Script

The IronPython script defines functions that respond to user and GUI interactions and implements the behavior of the extensions. Typically, functions are invoked through the different events or callbacks in the XML extension definition file.

In the extension `ExtSample1`, the IronPython script is named `sample1.py`. The contents of this script follow:

```
clr.AddReference("Ans.UI.Toolkit")
clr.AddReference("Ans.UI.Toolkit.Base")
from Ansys.UI.Toolkit import *

def init(context):
    ExtAPI.Log.WriteMessage("Init ExtSample1...")

def HighFiveOut(analysis_obj):
    MessageBox.Show("High five! ExtSample1 is a success!")
```

The script contains two functions:

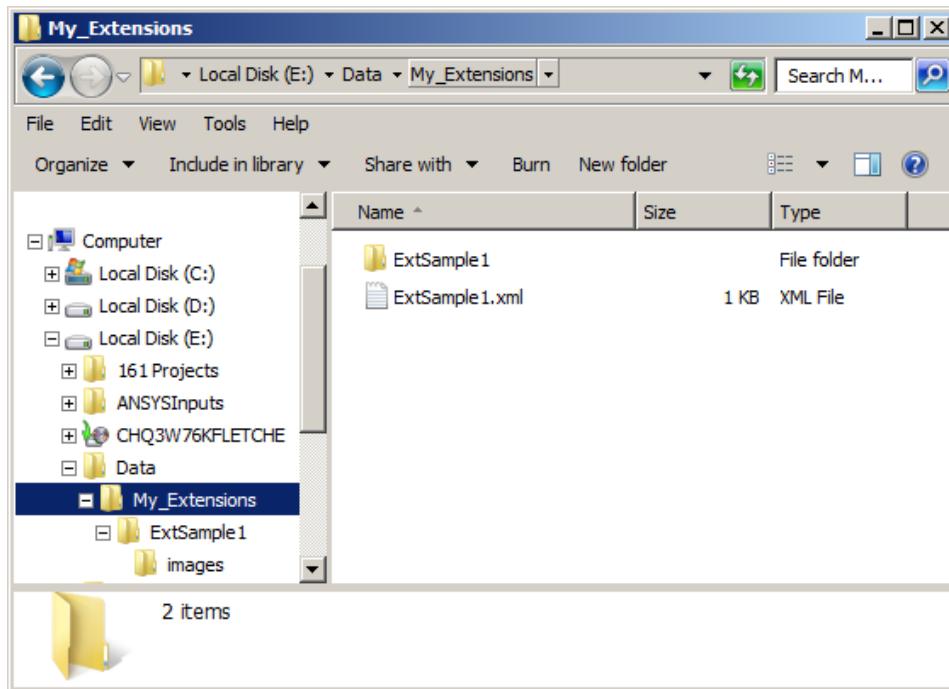
- `init()`: Called when the ANSYS product is opened. In the XML file, the argument `context` for the `<interface>` block contains the name of the product ("`Mechanical`").
- `HighFiveOut()`: Called when the toolbar button **HighFive** is clicked. The callback `<onclick>` passes an instance of the active object `Analysis` as an argument.

For any function, the global variable `ExtAPI` represents the main entry point for all the services provided by ACT. As an example, the function `init()` uses the ACT interface `ILog` to write one message in the log file. For more information on the available interfaces, see the [ANSYS ACT API Reference Guide](#).

Setting Up the Directory Structure

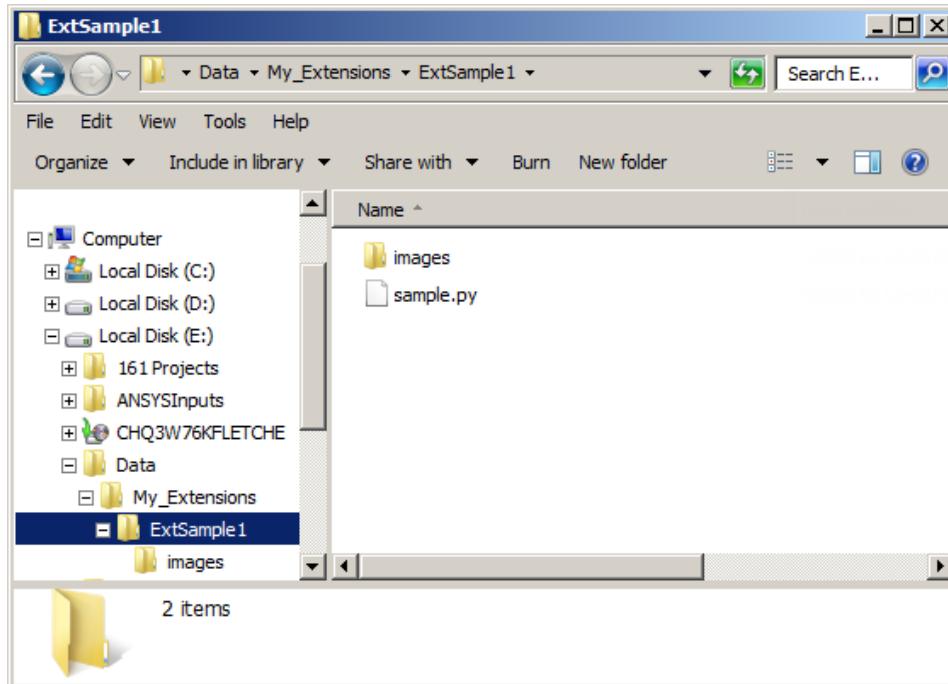
In the directory in which you store your extensions, the XML extension definition file and the folder with the extension's IronPython script and supporting files are stored at the same level. Both the XML file and the folder must have the same name as the extension.

The following figure shows the top-level directory structure for the extension ExtSample1. You can see that the extension's XML file is named `ExtSample1.xml` and its folder is named `ExtSample1`.

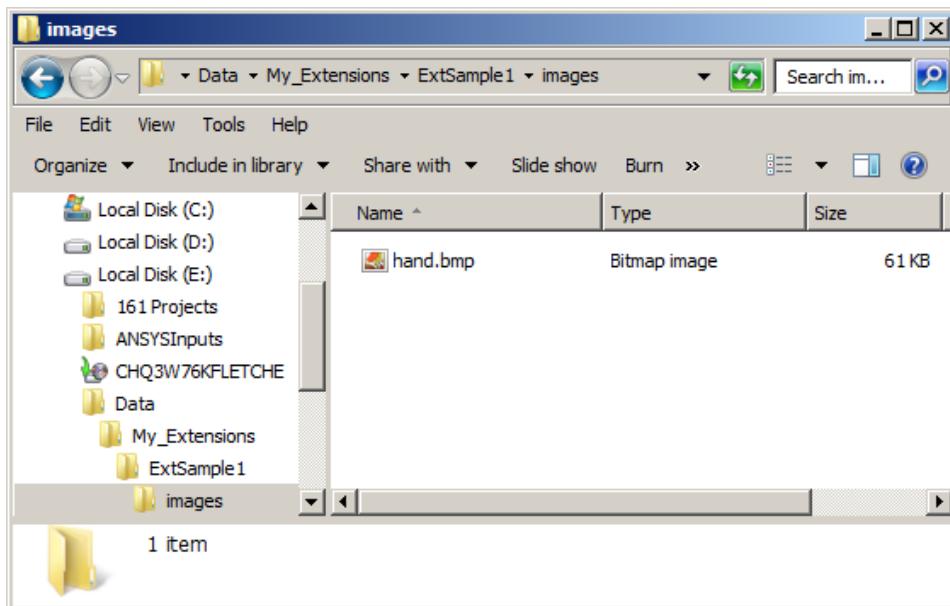


The following figure shows the contents of the folder `ExtSample1`.

- The IronPython script `sample1.py` contains the code for fulfilling the behavior of the example extension.
- The folder `images` contains the image file to display as the icon in the user interface of the extension.

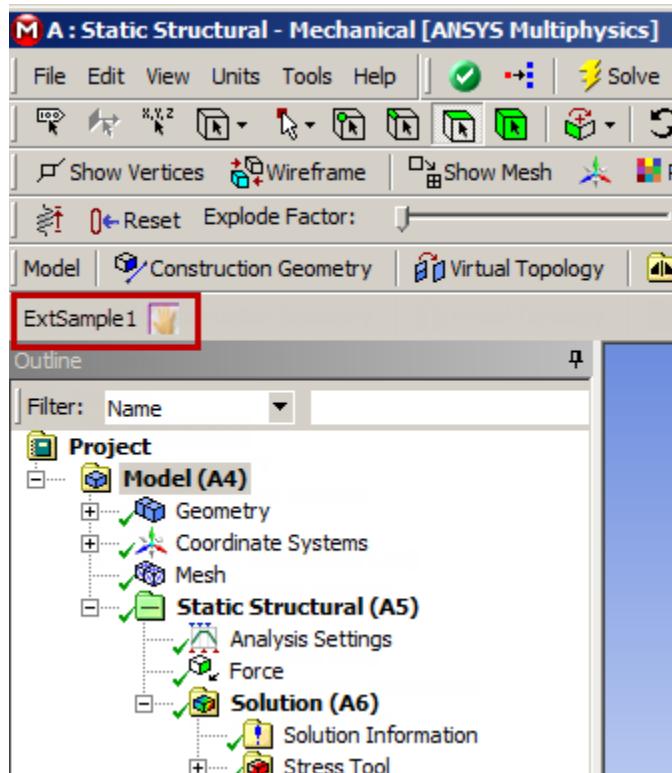


The following figure shows the contents of the folder `images`. It contains the image file `hand.bmp`, which is used as the icon for the custom button exposed on the Mechanical toolbar.

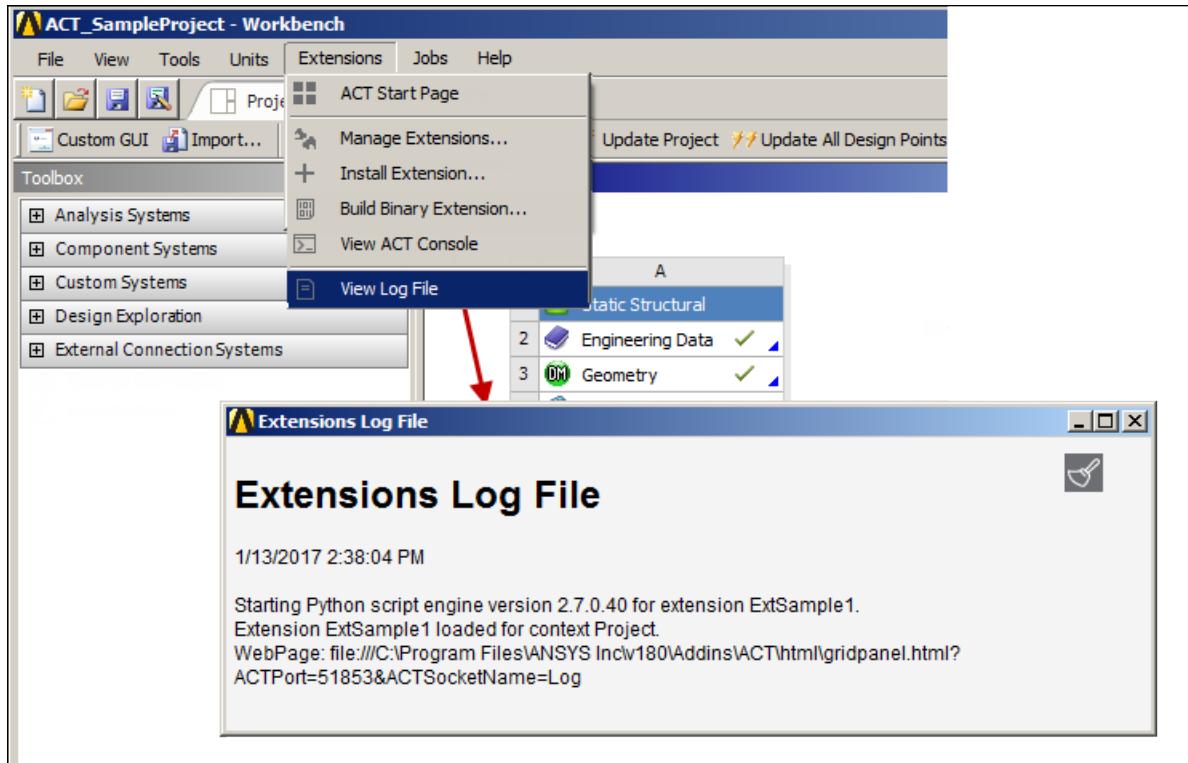


Viewing Exposed Custom Functionality

In the extension **ExtSample1**, the attribute **context** is set to **Mechanical** so that the extension loads with ANSYS Mechanical. In the following figure, the extension toolbar **ExtSample1** is shown in the Mechanical user interface.



The **Extensions Log File** shown in the following figure indicates that the extension **ExtSample1** was loaded for context **Project**.



Tip

You can open the **Extensions Log File** in your web browser by copying all of the text that follows **WebPage** and pasting it into the address bar. For example, given the preceding log file, you'd copy and paste the following text into the address bar:

```
file:///C:/Program Files/ANSYS Inc/v181/Addins/ACT/html/gridpanel.html?ACTPort=50422&ACTSocketName=Log
```

Script Processing

The XML extension definition file for an extension can reference multiple IronPython scripts. For example, consider this excerpt from the XML file for the extension **WizardDemos**:

```
<extension version="2" minorversion="1" name="WizardDemos">  
  <guid shortid="WizardDemos">7fdb141e-3383-433a-a5af-32cb19971771</guid>  
  <author>ANSYS Inc.</author>  
  <description>Simple extension to test wizards in different contexts.</description>  
  
  <script src="main.py" />  
  <script src="ds.py" />  
  <script src="dm.py" />  
  <script src="sc.py" />
```

This extension has four **<script>** blocks. All four scripts referenced by this extension are placed in the extension folder. When ACT processes multiple scripts, it loads them into a single scope, as if the contents of all scripts are contained in a single flat file. This works for a scripted extension that uses **import** statements because all scripts reside in the extension folder.

When a scripted extension is to be compiled into a binary extension, you can specify which files to compile by setting the attribute **compiled** for the **<script>** blocks to **true** before building the WBEX file:

```
<script src="main.py" compiled="true" />
<script src="ds.py" compiled="true" />
<script src="dm.py" compiled="true" />
<script src="sc.py" compiled="true" />
```

When ACT processes a WBEX file with multiple `<script>` blocks that are marked as `compiled`, it essentially pushes all of the content in these scripts into the binary buffer stream. When the installed WBEX file is loaded, ACT then reads the `<script>` blocks from the binary buffer and loads the script content into a single scope.

Consequently, the result for the binary version is the same as the scripted version. All contents in the multiple IronPython scripts are loaded into a single scope without any module designations, just as if you had originally combined the different scripts into one large, single script.

Unlike a scripted extension, the installed binary extension can no longer import another IronPython script as a module because the scripts no longer reside in an extension folder. Before building a binary extension, you must remove import statements and module prefixes because methods and classes are invoked as if all scripts are in one large, single script. By flattening IronPython scripts in this way, both the scripted and binary versions of the extension run successfully.

This flattening does make it difficult to test the scripted extension before building the binary version because you cannot truly test the implementation until after you create and install the WBEX file. However, you can compile your IronPython modules into DLLs, remove their corresponding `<script>` blocks from the XML extension definition file, and import them as required in your main IronPython script.

Note

You cannot currently flag files to have the binary extension builder skip them. During the building of a WBEX file, any messages that you see about files being skipped are the result of ACT no longer needing the plain-text scripts because the `<script>` blocks are marked as `compiled="true"`.

Using Extension Libraries

As you develop extensions to integrate customizations in ANSYS products, you can use additional libraries to share IronPython functions between extensions. Some libraries are installed by default with ACT to help you to customize ANSYS products.

Supported libraries related to Mechanical are located in the following folder:

```
%ANSYSversion_DIR%\..\Addins\ACT\libraries\Mechanical
```

Supported libraries related to AIM are located in the following folder:

```
%ANSYSversion_DIR%\..\Addins\ACT\libraries\Study
```

Descriptions of the libraries included with ACT follow:

ansys.py

Provides helpers to generate an APDL command block.

chart.py

Provides helpers to generate charts, such as curves and histograms.

units.py

Provides a set of generic functions to convert data from one unit system to another.

wbjn.py

Provides a tool to communicate data to and from the Workbench **Project** tab.

For more information, see [Associated Libraries \(p. 243\)](#).

Building a Binary Extension

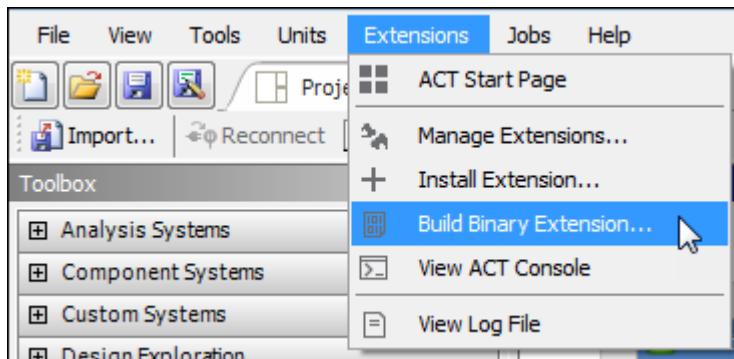
Once you've created a scripted extension, you can turn it into a compiled binary extension to be shared with ANSYS users. A separate compiler is not necessary. ACT provides its own process for encapsulating the scripted extension to a binary format that users cannot view or edit.

The encapsulation of the scripted extension generates a WBEX file. When you build a binary extension, the resulting WBEX file contains all files or directories necessary for the extension.

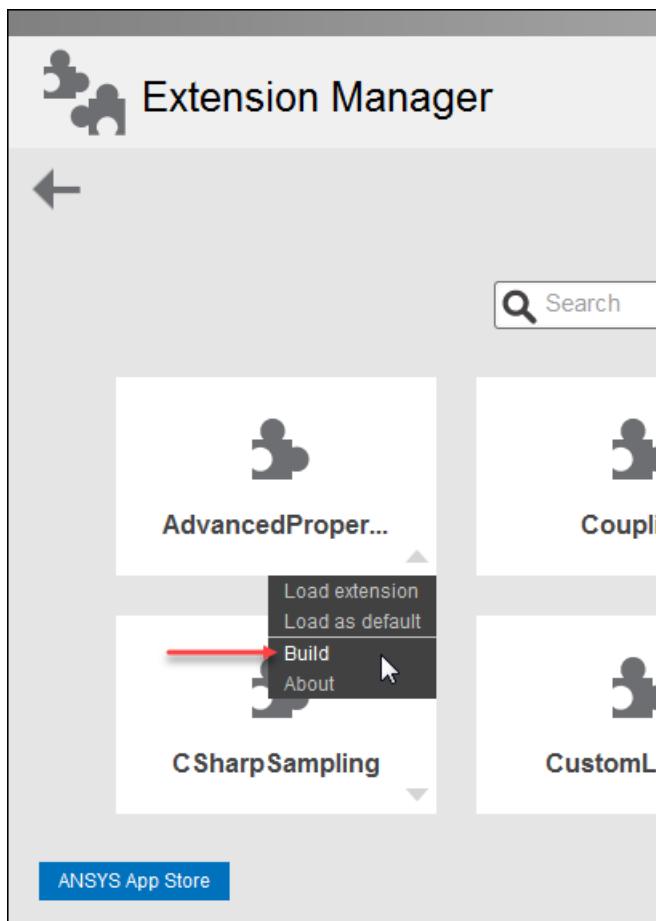
Accessing the Binary Extension Builder

You can use any of the following methods to access the binary extension builder:

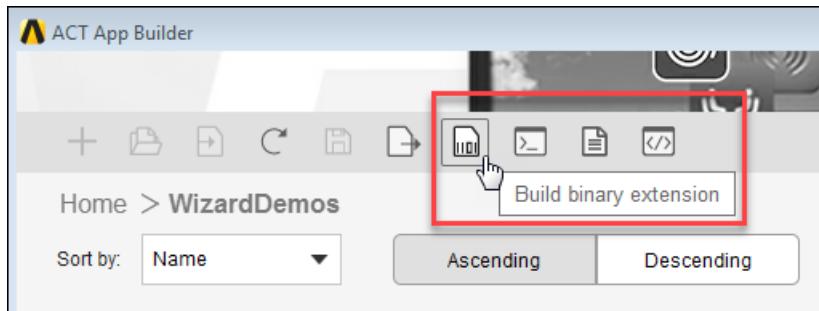
- From the menu in Workbench or AIM, select **Extensions > Build Binary Extension**.



- From the **Extension Manager**, right-click the extension and select **Build**. The **Extension Manager** is fully described in [Using the Extension Manager \(p. 29\)](#).



- From an app project open in the **ACT App Builder**, click the toolbar button for building a binary extension.



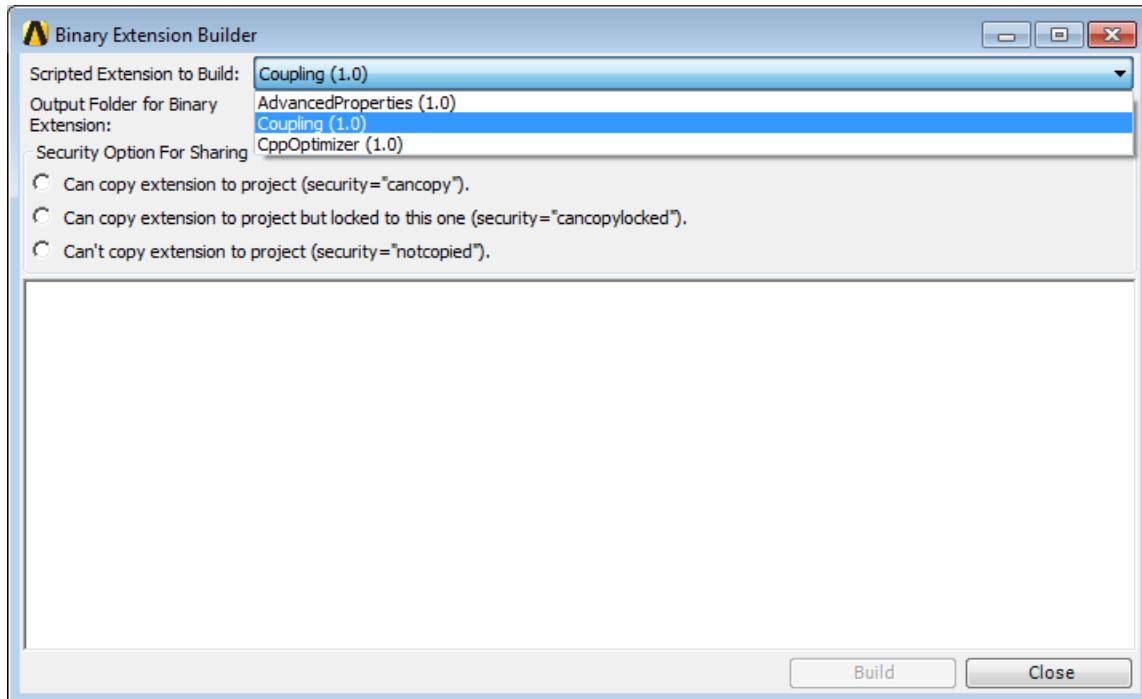
The appearance of the binary extension builder depends on how it was accessed.

Building a Binary Extension

To build a binary extension, do the following:

- If the builder was accessed from the **Extensions** menu, for **Scripted Extension to Build**, select the extension to build as a WBEX file.

Choices include all scripted extensions in directories defined in the **Extension** tab of the **Options** dialog box. For more information, see [Configuring Extension Options \(p. 21\)](#). Both the extension name and version are shown to avoid confusion if multiple versions of an extension are defined.



If the builder was accessed from the **Extension Manager** or **ACT App Builder**, the **Scripted Extension to Build** option is not shown because an extension selection has already been made.

2. Identify the folder in which to output the WBEX file.
3. Specify a security level for sharing the extension.

Your selection specifies whether the extension can be saved within an ANSYS Workbench project and, when that project is shared, if the extension can be loaded with the shared project. The security level allows the developer of the extension to control how the extension can be shared and used with various projects. Choices are:

- **Can copy extension to project:** Each time a user asks to save the extension with a project, the extension itself is copied into the project and consequently is available each time the project is opened. The extension can also be used with other projects.
- **Can copy extension to project but locked to this one:** The extension can be saved within a project, as with the previous option, but the use of the extension is limited to the current project. If a user opens a new project, the extension is not available.
- **Can't copy extension to project:** The extension is not saved with the project and is not shared with other users of the project.

Note

- The extension can be sent separate from a project. The process for saving extensions within a project is described in [Configuring Extensions to be Loaded by Default \(p. 37\)](#).
 - Security options only apply if the user wants to save the extension with the project. Otherwise, they are not applicable.
-

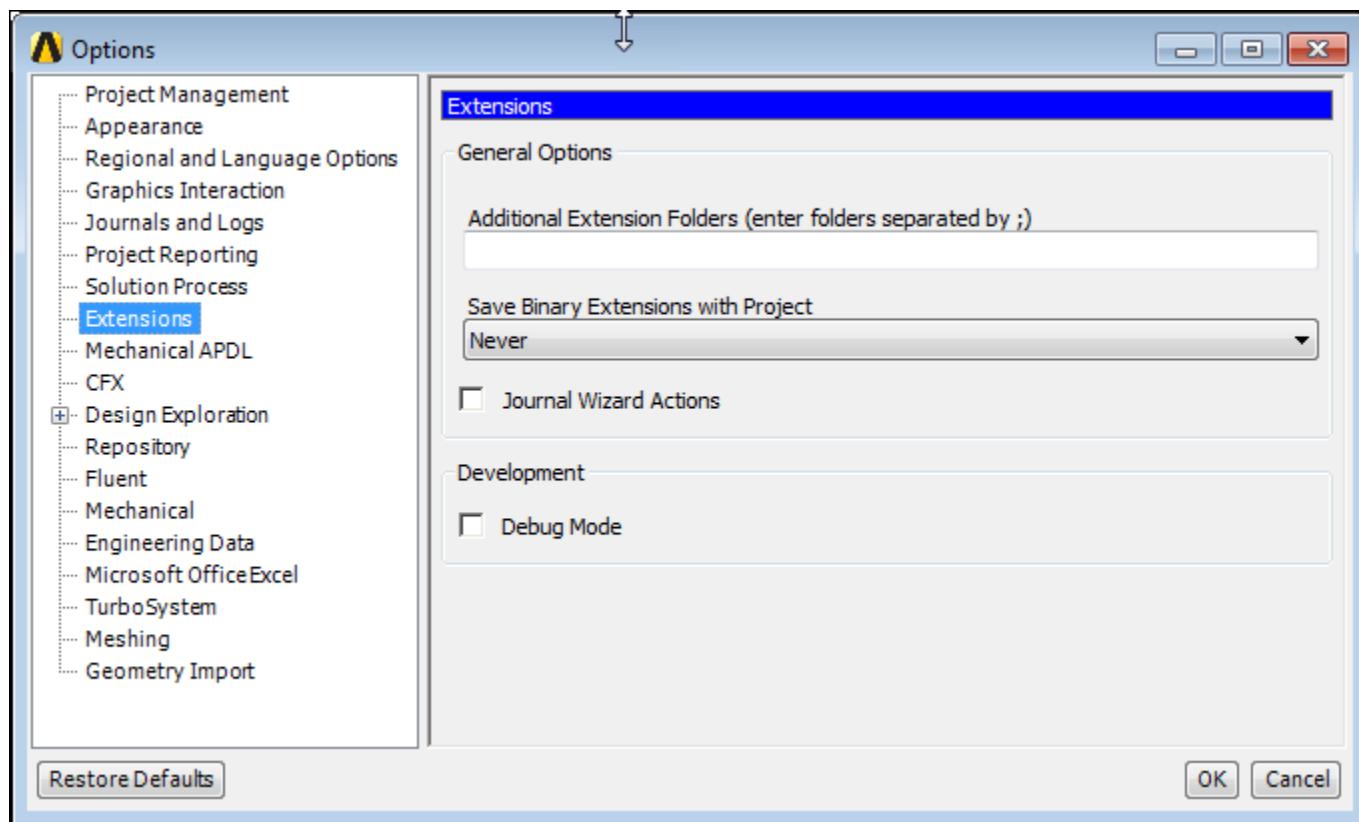
Once all options are specified, the **Build** button is enabled.

- Click **Build** to launch the build.

While the build is in progress, the bottom part of the window displays build information. [Script Processing \(p. 16\)](#) describes how references to multiple IronPython scripts in an XML extension definition file are processed.

Configuring Extension Options

From the menu in Workbench or AIM, you select **Tools > Options** to define all types of different options in the **Options** dialog box. ACT options are defined on the **Extensions** tab.



You use the following options to configure extensions:

- [Additional Extension Folders Option](#)
- [Save Binary Extensions with Project Option](#)
- [Journal Wizard Actions Option](#)
- [Debug Mode Option](#)

Additional Extension Folders Option

This option defines the additional folders in which ACT searches for the extensions to expose in the **Extension Manager**. You can define several folder names, separating them with a semicolon (;).

Folders that you define here are added to the user's **ApplicationData** folder: **%APPDATA%\Ansys\<version>\ACT\extensions**.

Because these folders are searched for extensions by default, the **Extension Manager** includes any extensions located in them.

Note

The default folders are not searched recursively. Because extensions stored in subdirectories are not found, store your extensions at the top level of the directory.

During this process, warning messages are returned for any conflicts. These messages are logged in the [Extensions Log File \(p. 267\)](#).

Save Binary Extensions with Project Option

This option specifies whether to save binary extensions within the project when the project is saved. Choices are:

- **Never** (default): The current loaded extensions are not saved within the project.
- **Copied but locked to the project**: Extensions are saved within the project but are limited to only this project.
- **Always**: The extensions are saved within the project and no restrictions exist as to their use in other projects. This option represents what most users expect when the project is saved. However, the behavior is dependent on the security option that the developer selected when building the binary extension. For more information, see [Building a Binary Extension \(p. 18\)](#). In particular, the following scenarios can occur:
 - The extension was built with the security option set to **Can't copy extension to project**. If the user sets the save option to **Always** or **Copied but locked to the project**, the security option has the priority. The extension is not saved within the project.
 - The extension was built with the security option set to **Can copy extension to project but locked to this one**, and the user sets the save option to **Always**. Although the save option does not impose any restriction on the extension, the security level limits the use of the extension to only the given project.

Journal Wizard Actions Option

This option indicates whether to automatically create a journal file when a wizard is run. The wizard can be a Workbench or AIM project wizard or a target product wizard. For more information, see [Types of Wizards \(p. 147\)](#).

Debug Mode Option

This option indicates whether the debug mode is enabled. When the **Debug Mode** check box is selected, the developer can debug the scripted extension.

The debug mode also exposes the following features, which help to debug the extension:

- In Workbench and AIM, a button for reloading the extension is added to the toolbar.
- In Mechanical and DesignModeler, an **ACT Development** toolbar displays. This toolbar includes buttons for reloading the extension and viewing the [Extensions Log File \(p. 267\)](#).

For more information, see [Debug Mode \(p. 265\)](#).

Using Extensions

ACT can manage an entire set of extensions for different ANSYS products. When extensions target multiple products, each extension provides its own customization level. However, the end user can define a personal configuration based on the extensions previously loaded in ANSYS Workbench, ANSYS AIM, or the target product of choice. Consequently, the level of customization is directly dependent on the choices the end user makes before opening the customized product.

Once a project using an ACT extension has been created and saved, any further use of the project must integrate any previously used extensions. If these extensions are available, Workbench loads the extensions automatically when the project is opened. If an expected extension has not been detected, an error message displays. For more information about controlling the availability of extensions, see [Configuring Extension Options \(p. 21\)](#).

Related Topics:

- [Using the ACT Start Page](#)
- [Using the Extension Manager](#)
- [Installing and Uninstalling Extensions](#)
- [Loading and Unloading Extensions](#)

Using the ACT Start Page

The **ACT Start Page** is a single page that provides you with convenient access to ACT functionality. From this page, you can access multiple tools for both development and execution of extensions. The **ACT Start Page** is available in all ANSYS products that support the use of extensions. This includes:

- Workbench
- AIM
- Products opened within Workbench, such as DesignModeler, DesignXplorer, Mechanical, and SpaceClaim
- Products opened independently as standalone instances, such as Electronics Desktop, Fluent, and SpaceClaim

Related Topics:

- [Accessing the ACT Start Page](#)
- [ACT Start Page Interface](#)

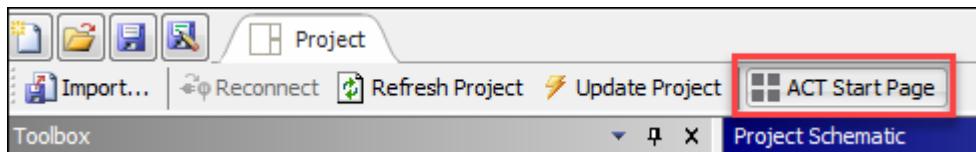
Accessing the ACT Start Page

You can access the **ACT Start Page** from Workbench, AIM, and standalone instances of Electronics Desktop, Fluent, and SpaceClaim.

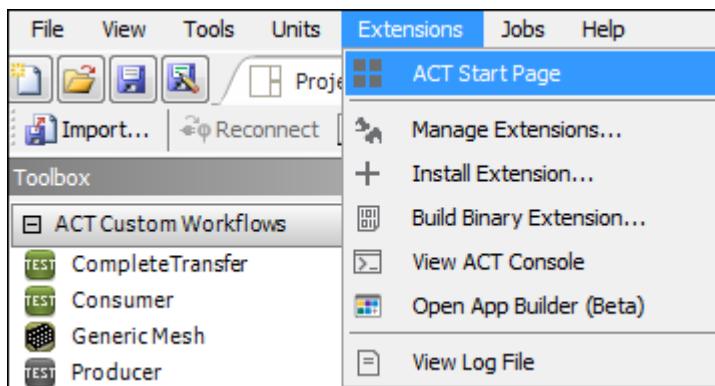
Accessing the ACT Start Page from Workbench

In Workbench, you access the **ACT Start Page** by doing one of the following:

- Clicking the **ACT Start Page** toolbar button.

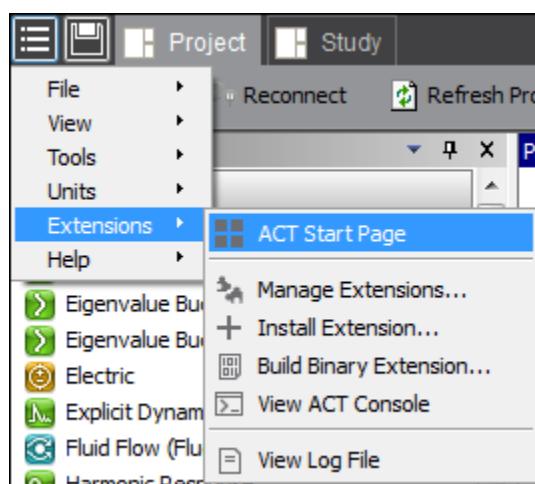


- Selecting **Extensions > ACT Start Page** from the main menu.



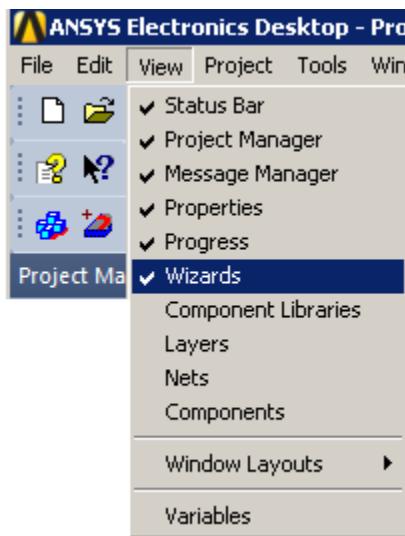
Accessing the ACT Start Page from AIM

From an AIM project or study, you access the **ACT Start Page** by clicking **Home** and then selecting **Extensions > ACT Start Page** from the menu.



Accessing the ACT Start Page from a Standalone Instance of Electronics Desktop

From a standalone instance of Electronics Desktop, you access the **ACT Start Page** by selecting **View > Wizards**.



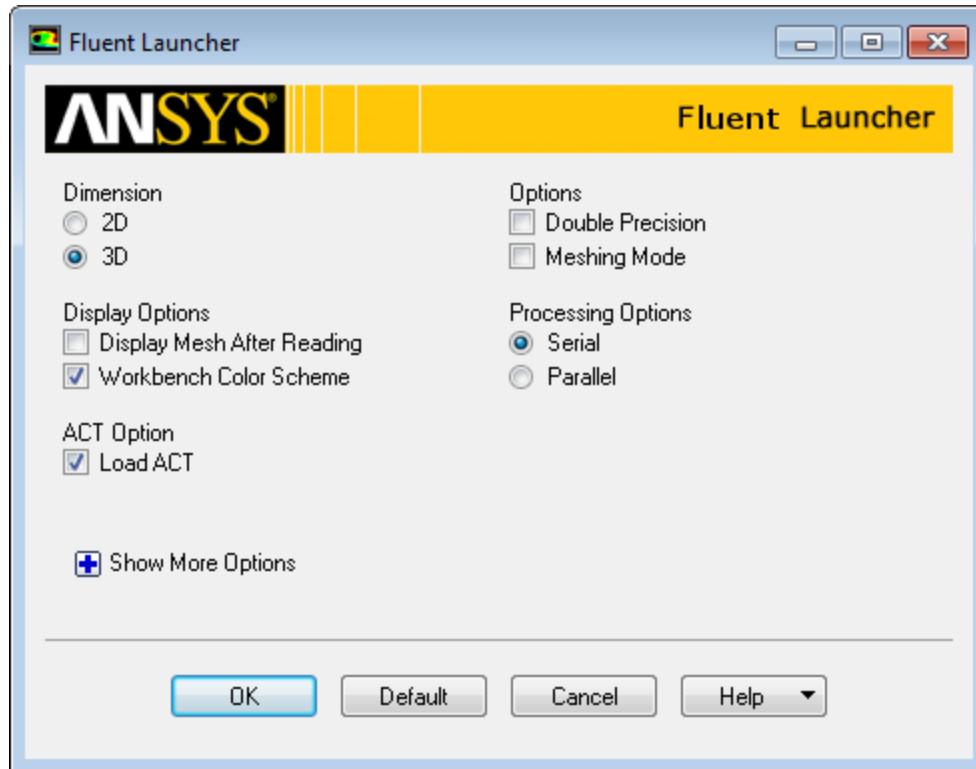
Note

When Electronics Desktop is opened as a standalone instance, the **ACT Start Page** provides an **Extension Manager** button because extensions must be managed from this independent instance. However, when Electronics Desktop is opened within Workbench, this page does not provide an **Extension Manager** button because extensions must be managed from the **ACT Start Page** for Workbench. For more information, see [Using the Extension Manager \(p. 29\)](#).

Accessing the ACT Start Page from a Standalone Instance of Fluent

From a standalone instance of Fluent, you access the **ACT Start Page** by doing one of the following:

- During startup, select the **Load ACT** check box in the **Fluent Launcher**.
- When Fluent is running, click **Arrange the workspace** and then **ACT**, which toggles between displaying and hiding the **ACT Start Page**.

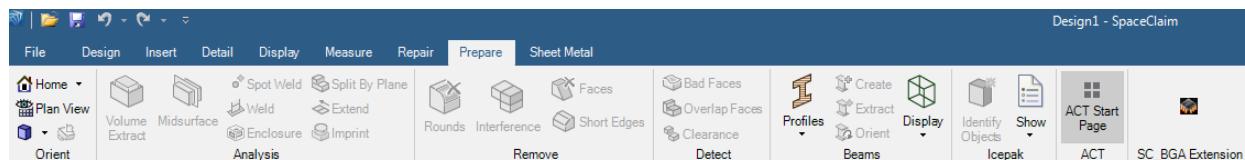


Note

When Fluent is opened as a standalone instance, the **ACT Start Page** provides an **Extension Manager** button because extensions must be managed from this independent instance. However, when Fluent is opened within Workbench, this page does not provide an **Extension Manager** button because extensions must be managed from the **ACT Start Page** for Workbench. For more information, see [Using the Extension Manager \(p. 29\)](#).

Accessing the ACT Start Page from a Standalone Instance of SpaceClaim

From a standalone instance of SpaceClaim, you access the **ACT Start Page** by clicking the **ACT Start Page** icon in the **Prepare** toolbar.



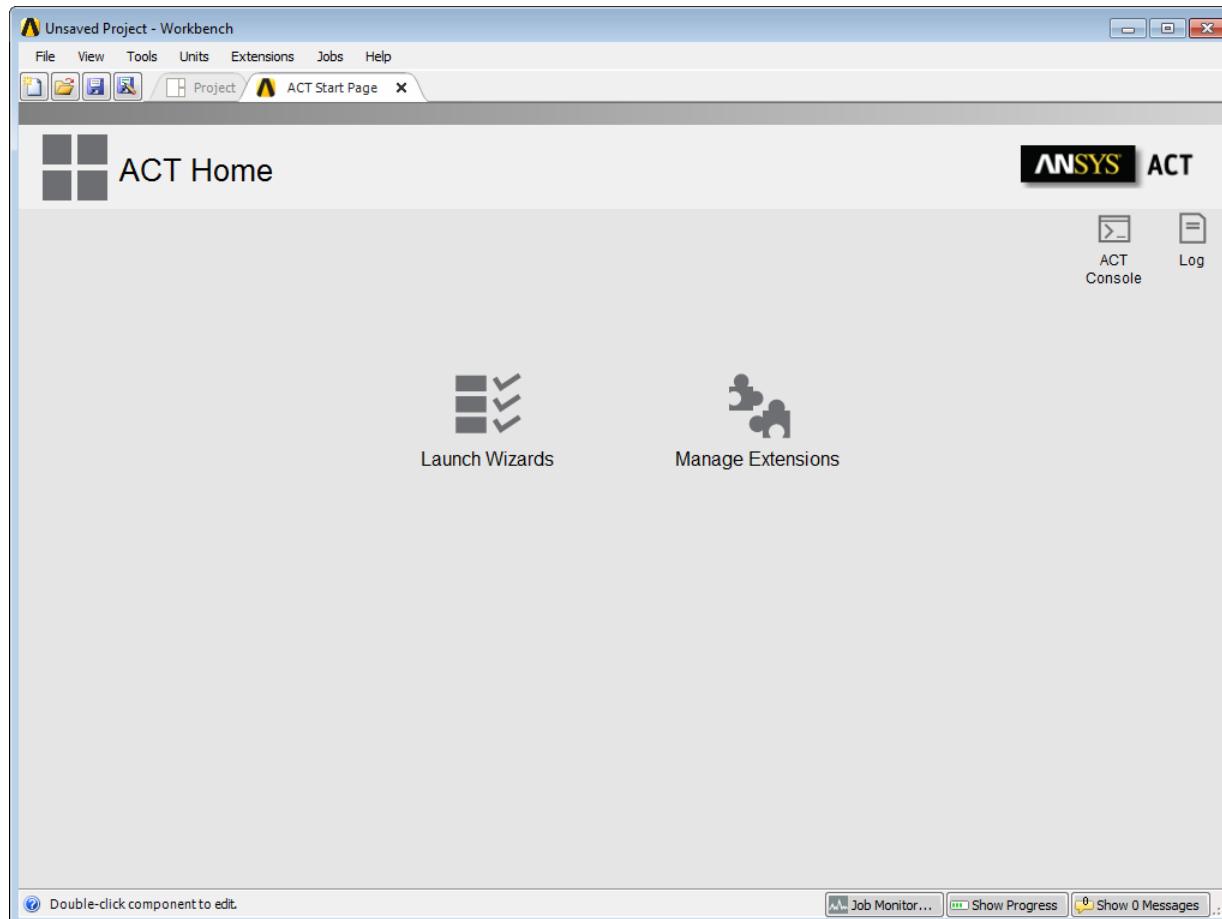
Note

When SpaceClaim is opened as a standalone instance, the **ACT Start Page** provides an **Extension Manager** button because extensions must be managed from this independent instance. However, when SpaceClaim is opened within Workbench, this page does not provide an **Extension Manager** button because extensions must be managed from the **ACT Start Page** for Workbench. For more information, see [Using the Extension Manager \(p. 29\)](#).

ACT Start Page Interface

The **ACT Start Page** includes the following icons for accessing ACT functionality and tools:

- **Launch Wizards.** See [Launching a Wizard from the Wizards Page \(p. 165\)](#).
- **Manage Extensions.** See [Using the Extension Manager \(p. 29\)](#).
- **ACT Console.** See [ACT Console \(p. 251\)](#).
- **Log.** See [Extensions Log File \(p. 267\)](#).



Using the Extension Manager

The **Extension Manager** is a tool for performing tasks like loading and unloading extensions and setting the extensions to load by default. There are two versions of the **Extension Manager**, but both versions offer the same capabilities:

- A newer version is accessed from the **ACT Start Page**.
- A legacy version is accessed from the Workbench or AIM **Extensions** menu.

The **Extension Manager** lists all extensions that have been installed and are available to be loaded. For information on how the list is populated, see [Additional Extension Folders Option \(p. 21\)](#).

Note

While the **Extension Manager** always lists installed binary extensions, it lists installed scripted extensions only if you have an ACT license.

In the XML extension definition file, you can view the name, type, and version. When an icon and description are defined, you can also view information about them.

The following sections address both versions of the **Extension Manager**:

Extension Manager Accessed via the ACT Start Page

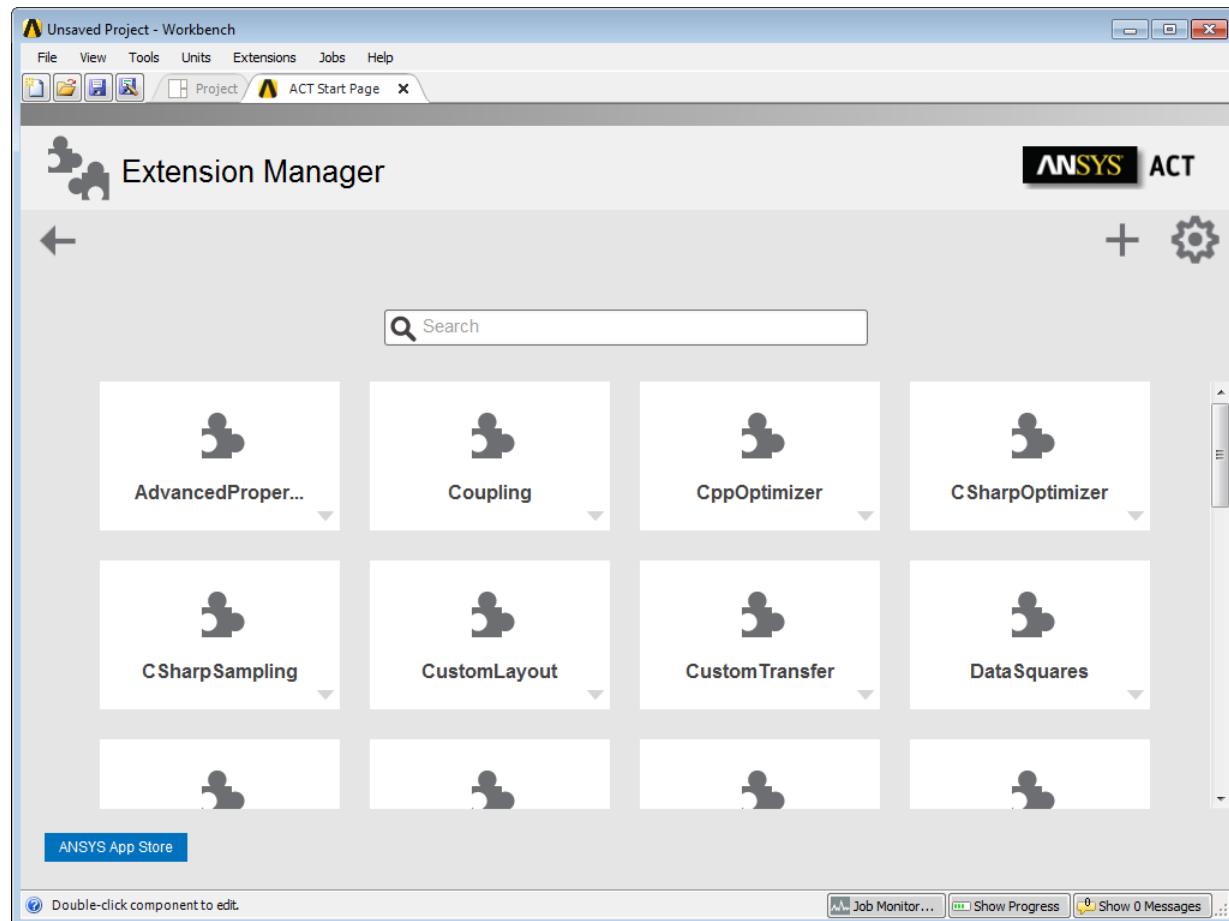
Extension Manager Accessed via Extensions Menu

Extension Manager Accessed via the ACT Start Page

To access the **Extension Manager** via the **ACT Start Page**:

1. Open the **ACT Start Page** as described in [Using the ACT Start Page \(p. 25\)](#).
2. Click the **Extension Manager** button.

The **Extension Manager** opens.



The **Extension Manager** shows all extensions that are currently installed. Scripted extensions are shown only if you have an ACT license. The green extensions are already loaded. You can do the following:

- Click one of the "+" icons to install a new binary extension.
- Click the gear icon to add a directory in which ACT is to search for extensions. For more information, see [Additional Extension Folders Option \(p. 21\)](#).
- Click the **ANSYS App Store** button in the lower left corner to access extensions available for download.
- Click the arrow icon to return to the **ACT Start Page**.
- Use the **Search** field to find a specific extension. This case-insensitive search tool looks for all strings entered that are separated by a blank (serving as the AND operation) and performs a combined search for strings separated by the OR operation. You can search by the following items:

Name, description, or author of an extension

Example 1: Enter "mydemowizards" to return all extensions of that name.

Example 2: Enter "authorname" to return all extensions with **author** set to **authorname**.

Name of a context

Example: Enter "Mechanical" to return all extensions with a **context** set to **Mechanical**.

Extension object type

Example: Enter the object to return all extensions containing at least one object of that type. For instance, enter "wizard", "workflow", or any **simdata** object, such as "load", "result", "solver", or "geometry".

Extension object type and object name separated by a colon

Example: Enter "load:my_load" to return all extensions having a **load** named **my_load**.

You can also right-click an extension to display a context menu showing the options available for that extension. The options shown depend on the current state of the extension. Descriptions follow of all options that can show:

Load extension

Available only for an unloaded extension. Loads the extension.

Unload extension

Available only for a loaded extension. Unloads the extension.

Load as default

Available only for an extension that is not automatically loaded to a project by default. Loads the extension to either an existing or new project when ANSYS Workbench or the corresponding targeted product is launched. For more information, see [Configuring Extensions to be Loaded by Default \(p. 37\)](#).

Do not load as default

Available only for an extension that is automatically loaded to a project by default. Does not load the extension to either an existing or new project when ANSYS Workbench or the corresponding targeted product is launched.

Uninstall

Available only for unloaded binary extensions. Uninstalls the binary extension.

Build

Available only for scripted extensions. Opens the binary extension builder as described in [Building a Binary Extension \(p. 18\)](#).

About

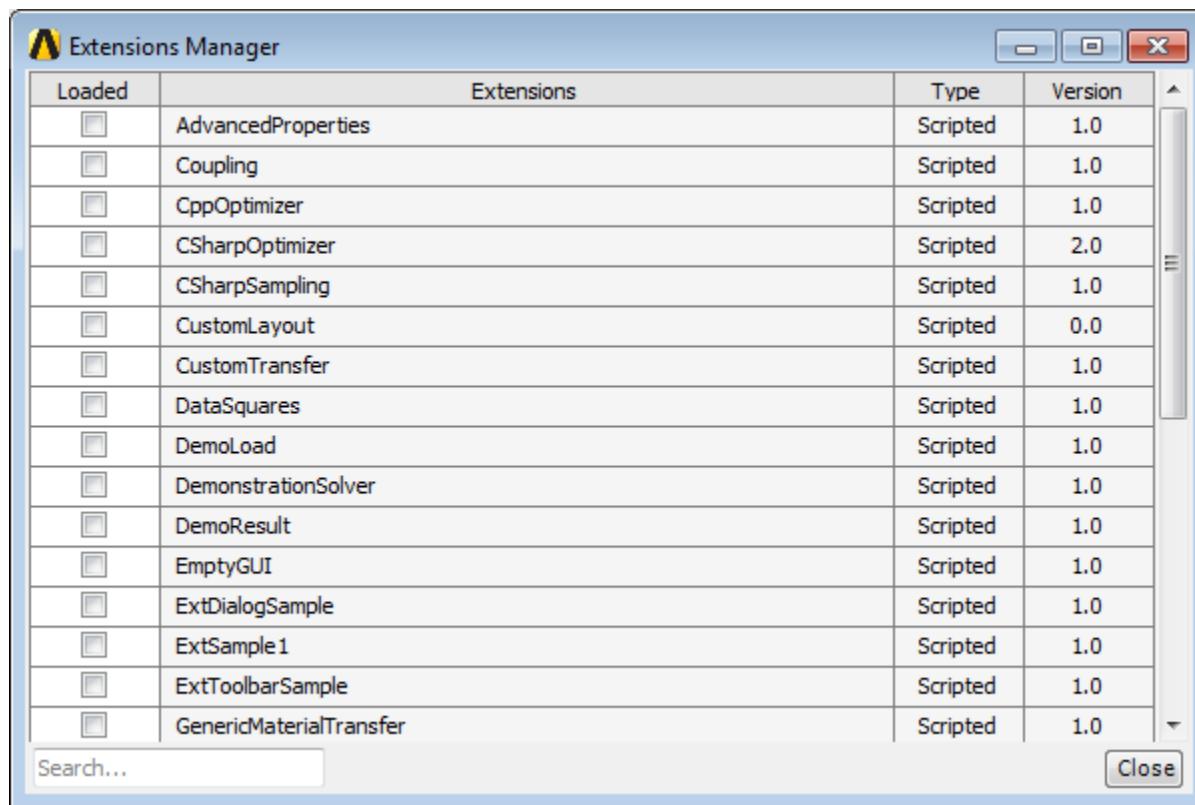
Displays extension information such as version, format (xml or wbex), folder, context, and feature.

Extension Manager Accessed via Extensions Menu

To access the legacy version of the **Extension Manager** in Workbench or AIM, do the following:

- In Workbench, select **Extensions > Manage Extensions** from the menu.
- In AIM, click **Home** and then select **Extensions > Manage Extensions** from the menu.

The **Extension Manager** opens.



This version of the **Extension Manager** shows all extensions that are currently installed along with the type and version for the extension. As noted earlier, scripted extensions are shown only if you have an ACT license. To load or unload an extension, select or clear the **Loaded** check box. Right-clicking an extension displays a context menu with comparable options as those in the other version of the **Extension Manager**. You also use the **Search** field in this version in the same way as in the other version.

Installing and Uninstalling Extensions

Once an extension has been created, it cannot be used until it is installed. The installation process differs for scripted and binary extensions.

Installing and uninstalling both types of extensions are addressed in the following sections:

- [Installing a Scripted Extension](#)
- [Uninstalling a Scripted Extension](#)
- [Installing a Binary Extension](#)
- [Uninstalling a Binary Extension](#)

Installing a Scripted Extension

To install a scripted extension, save the extension and associated files in one of the following locations:

- `%ANSYSversion_DIR%\..\Addins\ACT\extensions`
- `%APPDATA%\Ansys\v182\ACT\extensions`
- Any of the include directories defined in the **Additional Extension Folders** field on the **Extensions** page of the **Options** dialog.
- Any of the include directories specified by using the gear icon on the **Extension Manager** (accessed via the **ACT Start Page**).

Uninstalling a Scripted Extension

To uninstall a scripted extension, remove the extension and its associated files and directories. If you remove a scripted extension while the **Extension Manager** is open, it continues to display this extension until you close and reopen the **Extension Manager**.

Note

The **Uninstall** option on the **Extension Manager** is applicable only to binary extensions.

Installing a Binary Extension

A binary extension can be installed using either the **Extensions** menu or the version of the **Extension Manager** accessed via the **ACT Start Page**.

To install a binary extension:

1. Access the install functionality using one of the following methods:
 - In Workbench, select **Extensions > Install Extension** from the menu.
 - In AIM, click **Home** and then select **Extensions > Install Extension** from the menu.
 - In the **Extension Manager** accessed via the **ACT Start Page**, click one of the "+" icons.
2. Browse to the binary extension (WBEX file) that you want to install.
3. Select the WBEX file and click **Open**.

Extensions installed this way are located in the current user's **Application Data** folder and are available for loading in the **Extension Manager**.

Note

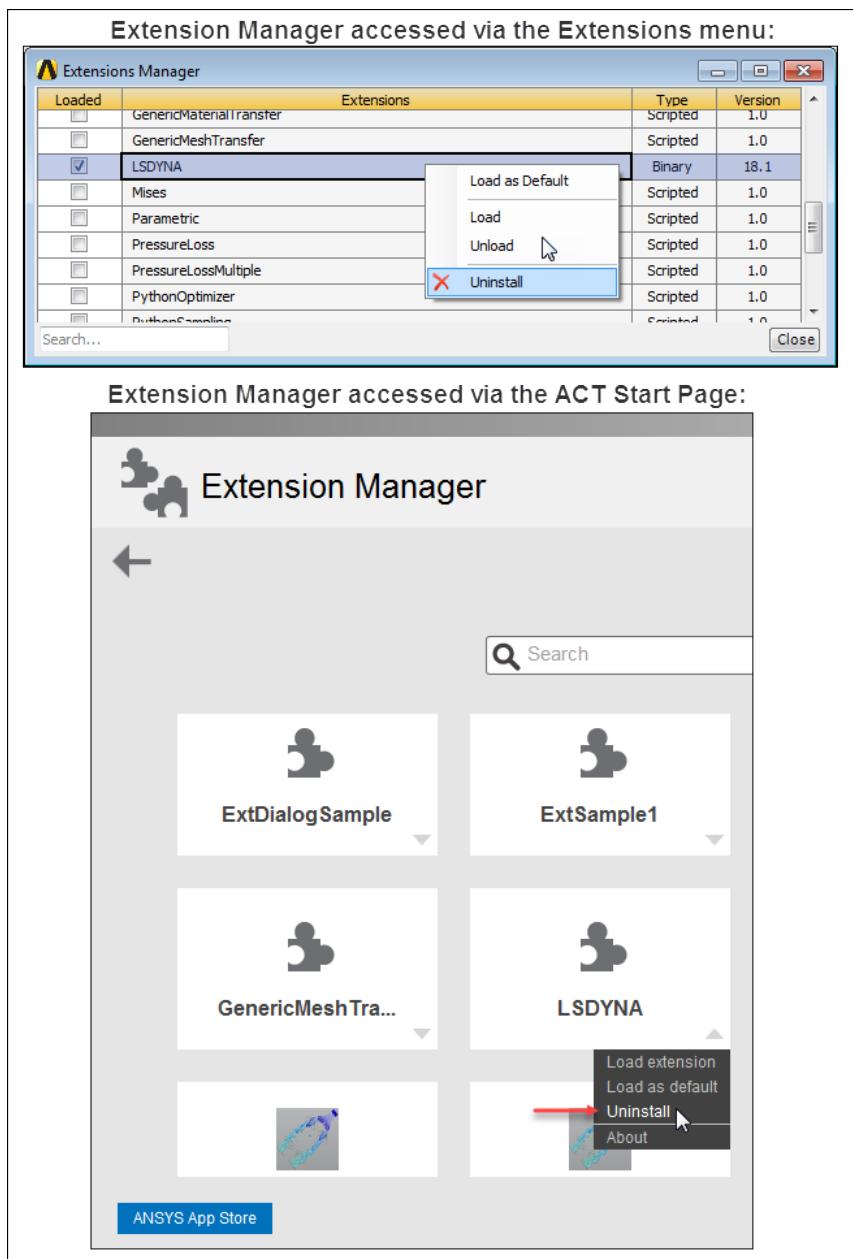
When a binary extension is installed, a new folder and a WBEX file are created because both are necessary for compatibility with ACT. If you need to move the new extension to a different folder, make sure that both the folder and the WBEX file are copied to the same folder at the same time.

Uninstalling a Binary Extension

This process for uninstalling extensions is available only for binary extensions. You can uninstall a binary extension using either version of the **Extension Manager**.

To uninstall a binary extension:

1. Open the **Extension Manager**.
2. Right-click the binary extension to display its context menu. For the **Extension Manager** accessed via the **ACT Start Page**, you can also display the context menu by clicking the down-arrow.
3. Select the **Uninstall** option.



Loading and Unloading Extensions

Extensions can be loaded and unloaded using either version of the **Extension Manager**. When an extension is marked as loaded in the **Extension Manager**, it is loaded with the corresponding targeted product, which you specify using the attribute `context` for the extension.

Note

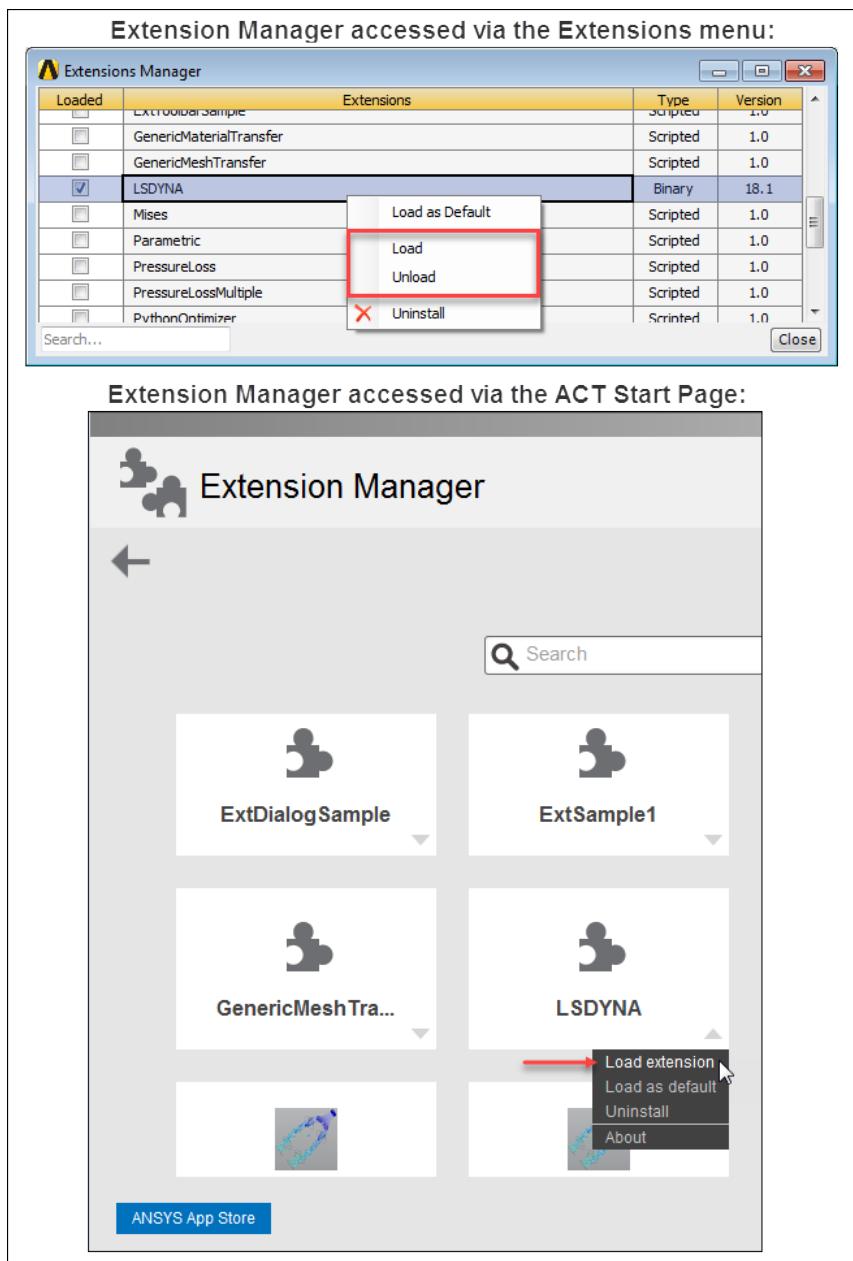
- The loading is automatic for extensions that have already been loaded and saved to the project. Any extensions to be automatically loaded must be available to the **Extension Manager**. If a required extension cannot be loaded when a project is opened, a warning message appears, indicating that you might encounter limited behavior if you proceed with project interaction. Clicking the **Show Details** button in this message lists the extensions that are missing from the project. Clicking **OK** proceeds with opening the project.

- Wizards that are to be run in standalone instances of Electronics Desktop, Fluent, or Space-Claim, which occurs when Workbench and AIM are not installed, must be loaded via the **Extension Manager** accessed on the **ACT Start Page**.
-

In the **Extension Manager** accessed via the **ACT Start Page**, you can also load or unload an extension simply by clicking the extension.

You can also load or unload an extension as follows:

1. Open the **Extension Manager**.
2. Access the option to load or unload the extension in one of the following ways:
 - For either version of the **Extension Manager**, right-click the extension and select **Load** or **Unload**.
 - For the legacy version accessed via the **Extensions** menu, select or clear the check box to the left of the extension.

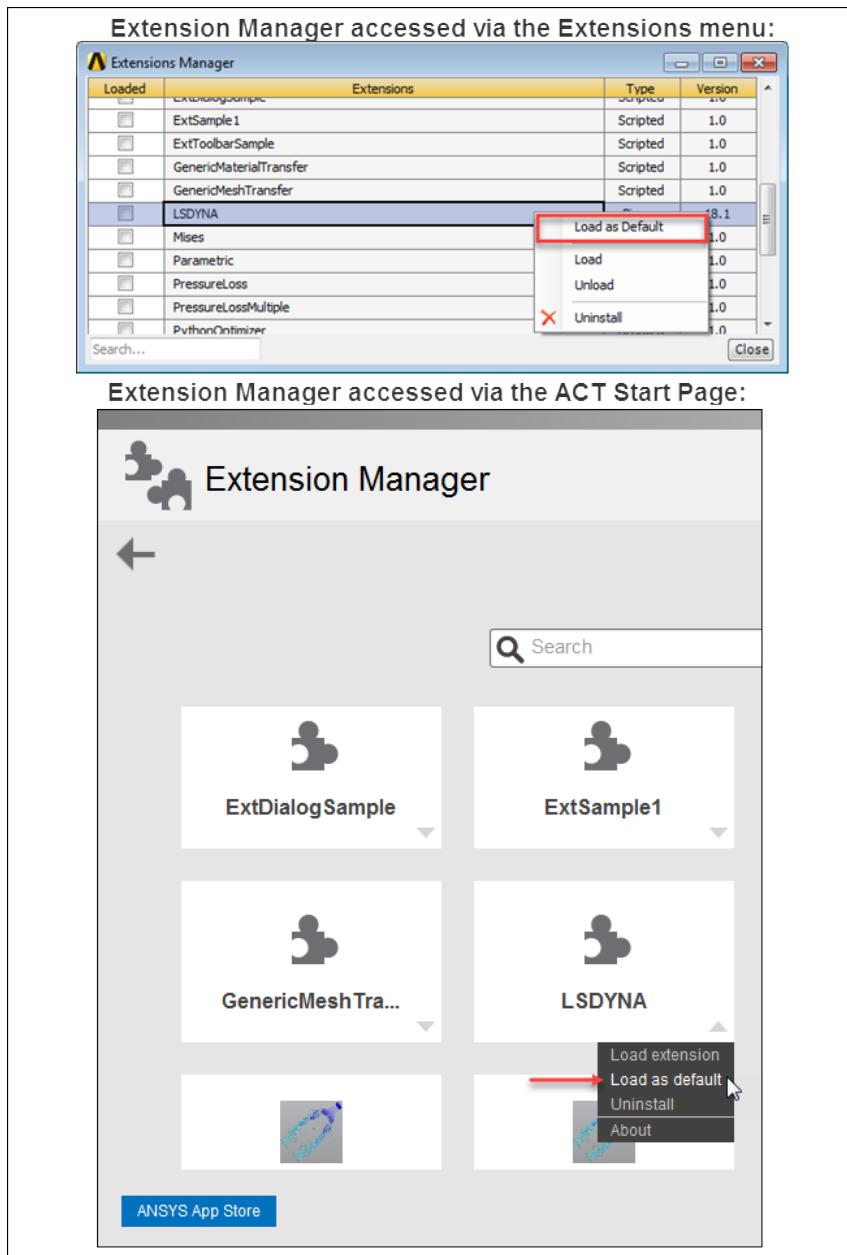


Configuring Extensions to be Loaded by Default

In either version of the **Extension Manager**, you can select the extensions to be loaded to the project by default. These extensions are automatically loaded when the corresponding targeted product is launched. No limit exists on the number of extensions that can be loaded by default. While the **Extension Manager** always shows binary extensions, it shows scripted extensions only if you have an ACT license.

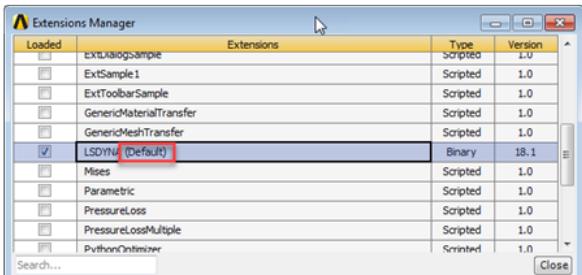
To specify whether an extension should be loaded by default:

1. Open the **Extension Manager**.
2. Right-click the extension and select **Load as Default** or **Do Not Load as Default**.

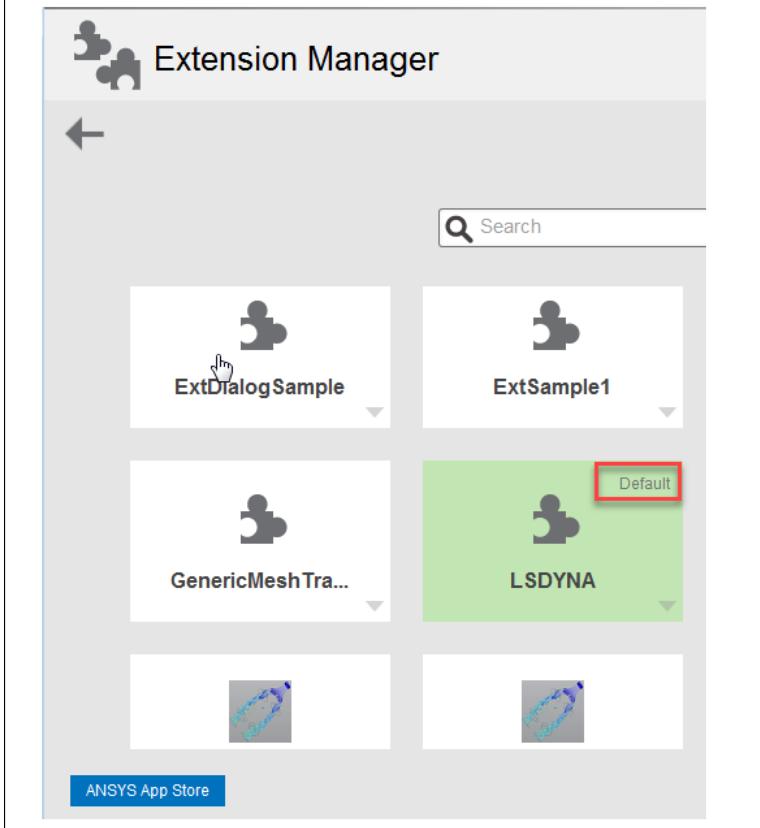


The word **Default** displays for extensions configured to be loaded by default.

Extension Manager accessed via the Extensions menu:



Extension Manager accessed via the ACT Start Page:



Feature Creation Capabilities

In the introductory chapter, the sample extension `ExtSample1` is used to describe the fundamental blocks in an extension. The `ExtSample1` extension demonstrates how to create a toolbar with a button that calls an IronPython scripted function. This chapter expands on this description and introduces the main feature creation capabilities that an extension can provide.

This chapter divides feature creation capabilities into the following categories:

[Common Capabilities](#)

[ACT-Based Properties](#)

[Capabilities for ANSYS Mechanical](#)

[Capabilities for ANSYS DesignModeler](#)

[Capabilities for ANSYS DesignXplorer](#)

[Capabilities for ANSYS AIM](#)

[Capabilities for ANSYS Fluent](#)

Common Capabilities

ACT provides feature creation capabilities that are common to supported ANSYS products: Workbench, AIM, Mechanical, DesignModeler, and DesignXplorer.

This section describes these common capabilities:

[Defining Toolbars and Toolbar Buttons](#)

[Binding Toolbar Buttons with ACT Objects](#)

[Defining Pop-up Dialog Boxes](#)

[Storing Data in Your Extension](#)

Note

The examples used in this section are included in the package **ACT Developer's Guide Examples**, which can be downloaded from the [ACT Resources](#) page. For more information, see [Extension Examples \(p. 273\)](#).

Defining Toolbars and Toolbar Buttons

This section focuses on the customization of toolbars and toolbar buttons in ANSYS products that expose their own toolbars. In addition to the target products that support ACT direct customization (Workbench, AIM, Mechanical, DesignModeler, and DesignXplorer), toolbar customization is also available for all products that support ACT wizard extensions (Fluent and Electronics Desktop).

ANSYS Mechanical is used for demonstration purposes. It requires that you use BMP files for the images to display for toolbar buttons. ANSYS Workbench and all other supported ANSYS products require you to use PNG files.

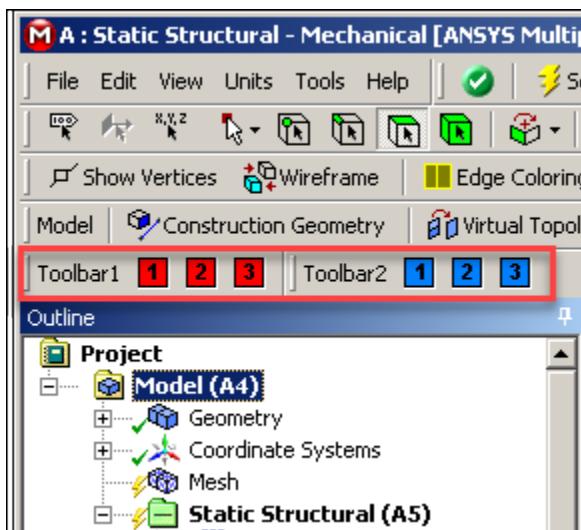
A toolbar serves as a parent container for one or more toolbar buttons. Conceptually, the toolbar should address a specific feature, with each toolbar button being associated with a function that supports the

feature. This relationship is reflected in the structure of the XML extension definition file that defines the toolbar and its buttons.

The XML file for the extension **ExtToolbarSample** follows.

```
<extension version="1" minorversion="0" name="ExtToolbarSample">
  <guid shortid="ExtToolbarSample">ccdbbed4-9fdd-4157-acea-abccddbd62fc</guid>
  <script src="toolbarsample.py" />
  <interface context="Mechanical">
    <images>images</images>
    <callbacks>
      <oninit>init</oninit>
    </callbacks>
    <toolbar name="ToolBar1" caption="ToolBar1">
      <entry name="TB1Button1" icon="button1Red">
        <callbacks>
          <onclick>OnClickTB1Button1</onclick>
        </callbacks>
      </entry>
      <entry name="TB1Button2" icon="button2Red">
        <callbacks>
          <onclick>OnClickTB1Button2</onclick>
        </callbacks>
      </entry>
      <entry name="TB1Button3" icon="button3Red">
        <callbacks>
          <onclick>OnClickTB1Button3</onclick>
        </callbacks>
      </entry>
    </toolbar>
    <toolbar name="Toolbar2" caption="Toolbar2">
      <entry name="TB2Button1" icon="button1Blue">
        <callbacks>
          <onclick>OnClickTB2Button1</onclick>
        </callbacks>
      </entry>
      <entry name="TB2Button2" icon="button2Blue">
        <callbacks>
          <onclick>OnClickTB2Button2</onclick>
        </callbacks>
      </entry>
      <entry name="TB2Button3" icon="button3Blue">
        <callbacks>
          <onclick>OnClickTB2Button3</onclick>
        </callbacks>
      </entry>
    </toolbar>
  </interface>
</extension>
```

The extension **ExtToolbarSample** adds two toolbars, Toolbar1 and Toolbar2. Each toolbar has three buttons.



This extension's XML file defines two `<toolbar>` blocks. Each `<toolbar>` block has two attributes, `name` and `caption`. The attribute `name` is required and is used for internal references. The attribute `caption` is the text displayed in Mechanical.

The `<toolbar>` block has a child `<entry>` block that defines the buttons in the toolbar. The `<entry>` block has two attributes, `name` and `icon`. The attribute `name` is required and is used for internal references. The name is also displayed as the tooltip. The attribute `icon` is the name of the image file to display as the button.

The `<entry>` block has a child `<callbacks>` block that defines callbacks to events. For instance, the callback `onclick` specifies the name of the IronPython function to invoke when the button is clicked.

Consider the second line in the previous XML code:

```
<script src="toolbarsample.py" />
```

This line defines `toolbarsample.py` as the IronPython script for the extension `ExtToolbarSample`. The code in this script follows. By default, ACT looks for the script in the directory of the extension. If the script is located in a different directory, an explicit path to the file must be inserted into the extension's XML file.

```
import os
import datetime
clr.AddReference("Ans.UI.Toolkit")
clr.AddReference("Ans.UI.Toolkit.Base")
from Ansys.UI.Toolkit import *

def init(context):
    ExtAPI.Log.WriteMessage("Init ExtToolbarSample ...")

def OnClickTB1Button1(analysis):
    LogButtonClicked(1, 1, analysis)

def OnClickTB1Button2(analysis):
    LogButtonClicked(1, 2, analysis)

def OnClickTB1Button3(analysis):
    LogButtonClicked(1, 3, analysis)

def OnClickTB2Button1(analysis):
    LogButtonClicked(2, 1, analysis)

def OnClickTB2Button2(analysis):
```

```
LogButtonClicked(2, 2, analysis)

def OnClickTB2Button3(analysis):
    LogButtonClicked(2, 3, analysis)

def LogButtonClicked(toolbarId, buttonId, analysis):
    now = datetime.datetime.now()
    outFile = SetUserOutput("ExtToolbarSample.log", analysis)
    f = open(outFile,'a')
    f.write("*.*.*.*.*\n")
    f.write(str(now)+"\n")
    f.write("Toolbar "+toolbarId.ToString()+" - Button "+buttonId.ToString()+" Clicked. \n")
    f.write("*.*.*.*.*\n")
    f.close()
    MessageBox.Show("Toolbar "+toolbarId.ToString()+" - Button "+buttonId.ToString()+" Clicked.")

def SetUserOutput(filename, analysis):
    solverDir = analysis.WorkingDir
    return os.path.join(solverDir,filename)
```

Each button defined in the extension `ExtToolbarSample` has a unique callback function. Each callback function passes the toolbar ID and the ID of the button pressed to the function `LogButtonClicked`, which stores them in the variables `toolbarId` and `buttonId`. These variables are referenced within the function where their string values are written. The functions `LogButtonClicked` and `SetUserOutput` demonstrate how to reduce redundant code in callbacks using utility functions. The object `Analysis` is passed to each callback `<entry>` and then used in the function `SetUserOutput` to query for the working directory of the analysis. The script in `toolbarsample.py` makes use of the namespace `datetime` from the .NET framework. The namespace `datetime` exposes a class that is also called `datetime`. The function `LogButtonClicked` invokes `datetime` to query the current date and time and stores the result in the variable named `now`. The utility `str()` is used to extract the string definition of the variable `now` to write out the date and time.

Binding Toolbar Buttons with ACT Objects

ACT provides the ability to bind a button from the ACT toolbar with an ACT object created in the tree of the product. This capability allows you to control the contextual availability of the buttons depending on the object selected in the tree. This method is similar to the control used for standard objects in Mechanical. The connection between the ACT object and the ACT button is made using the attribute `userobject` in the `<entry>` block of the interface definition in the XML extension definition file. The following code demonstrates this type of connection for a result object:

```
<interface context="Mechanical">
  <images>images</images>
  <toolbar name="My_Toolbar" caption="My_Toolbar">
    <entry name="Button_For_My_Result" icon="result" userobject="My_Result">
    </entry>
  </toolbar>
</interface>

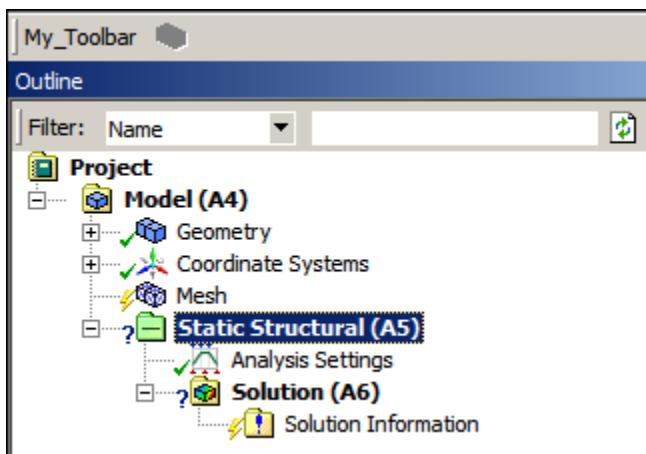
<simdata context="Mechanical">
  <result name="My_Result" version="1" caption="My_Result" icon="result" location="elemnode"
type="scalar">
  </result>
</simdata>
```

As an example, if the ACT button is bound with an ACT load in Mechanical, this button is activated only if the object is selected in the Mechanical environment. Otherwise, it is inactive.

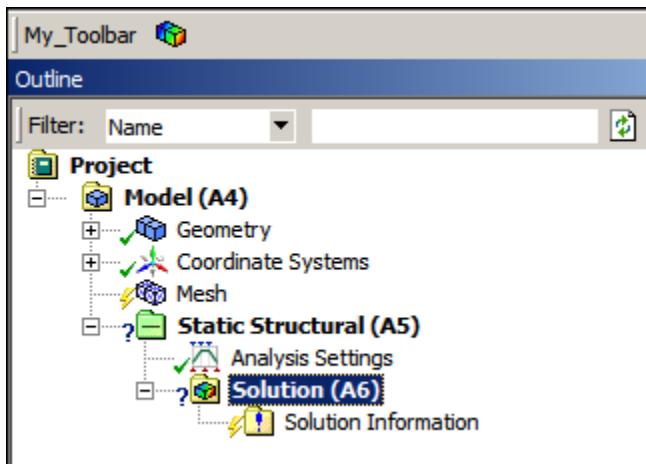
In the same way, if the ACT button is bound with an ACT result in Mechanical, this button is active only if the object is selected in the solution. Otherwise, it is inactive.

For any ACT object bound to a button, the callback `<onclick>` is not used.

The following figure illustrates the user interface behavior based on selection of an environment object in the tree. The button is inactive.



This next figure illustrate the user interface behavior based on selection of a solution object in the tree. The button is active.



In addition to the control provided by the connection between the ACT button and the ACT object, the callback <canadd> can be implemented to add new criteria to be considered for the activation and deactivation of the button. If the callback <canadd> of the object returns **false**, the associated button is deactivated. A typically example consists of filtering a particular analysis type to activate a specific load.

Defining Pop-up Dialog Boxes

This section discusses message dialog boxes using the extension ExtDialogSample as an example. This extension defines a menu labeled **DialogSample** with one menu item labeled **GetFilename**. It demonstrates how to open a file selection dialog box and display a message dialog box in the user interface.

The following XML extension definition file for the extension ExtDialogSample is similar to the one for the extension ExtToolbarSample.

```
extension version="1" name="ExtDialogSample">
<guid shortid="ExtDialogSample">25c00857-4e27-4b01-bd22-c52699ac39e9</guid>
<script src="dialogsample.py" />
<interface context="Mechanical">
```

```
<images>images</images>
<callbacks>
    <oninit>init</oninit>
</callbacks>
<toolbar name="DialogSample" caption="DialogSample">
    <entry name="DialogSample1" caption="GetFilename" icon="blank">
        <callbacks>
            <onclick>GUIMenuOpenFile</onclick>
        </callbacks>
    </entry>
</toolbar>
</interface>
</extension>
```

The callback function specified in the XML file for the GetFilename menu button is **GUIMenuOpenFile**. The IronPython script `dialogsample.py` follows.

```
clr.AddReference("Ans.UI.Toolkit")
clr.AddReference("Ans.UI.Toolkit.Base")
from Ansys.UI.Toolkit import *

def init(context):
    ExtAPI.Log.WriteMessage("Init ExtDialogSample ...")

def GUIMenuOpenFile(analysis):
    filters = "txt files (*.txt)|*.txt|All files (*.*)|*.*"
    dir = "c:\\\\"
    res = FileDialog.ShowDialog(ExtAPI.UserInterface.MainWindow,dir,filters,2,"ExtDialogSample","")

    if res[0]==DialogResult.OK:
        message = str("OPEN selected -> Filename is "+res[1])
    else:
        message = "CANCEL selected -> No file"
    MessageBox.Show(message)
```

When **GUIMenuOpenFile** is invoked, an instance of a modal file selection dialog box is created by calling `FileDialog.ShowDialog`. The class `FileDialog` is provided by the UI Toolkit. The user inputs the necessary information in the dialog box. When the user clicks the **Open** or **Cancel** button, the file selection dialog box closes and this information is returned to **GUIMenuOpenFile**. The returned information is then used to create a message dialog box. The message shown in the message dialog box validates the result returned from the file selection dialog box.

Storing Data in Your Extension

Two mechanisms are available to store data in your extension. These mechanisms are based on the callbacks `<onsave>` and `<onsave>` or on attributes. These mechanisms are available for any ANSYS product that supports ACT.

The callback `<onsave>` is called each time the target product saves the project. Consequently, this callback allows the creation of dedicated files to store data, in addition to the standard ANSYS Workbench project.

The callback `<onload>` is called each time the target product loads the project. The process here is similar to the callback `<onsave>`, but it is now devoted to reading in additional data.

Attributes represent an interesting way to store data because they do not require the creation of external files.

Attributes are defined by name and content. The content can be simply defined by a single value, such as an integer or a double, or with more complex data. The content must be serializable. To accomplish this, implement the serialization interface provided by .Net. Types such as `integer`, `double`, `list`, and `dictionary` are serializable by default.

Attributes are automatically saved and resumed with the project. Attributes can be associated with an extension, an ACT load or result, and a property.

Attributes are created or edited with the method:

```
Attributes["attribute_name"] = attribute_value
```

Content can be retrieved with the method:

```
attribute_value = Attributes["attribute_name"]
```

An example follows of an attribute associated with a property:

```
prop.Attributes["MyData"] = 2
val = prop.Attributes["MyData"]
```

A similar example follows for an attribute associated with a load:

```
load.Attributes["MyData"] = 2
val = load.Attributes["MyData"]
```

A similar example follows for an attribute associated with an extension:

```
ExtAPI.ExtensionMgr.CurrentExtension.Attributes["MyData"] = 2
v = ExtAPI.ExtensionMgr.CurrentExtension.Attributes["MyData"]
```

An attribute associated with an extension can be shared between products using the same extension. For example, two Mechanical sessions can share data.

The command to store the attribute in a shared repository is:

```
ExtAPI.ExtensionMgr.CurrentExtension.SetAttributeValueWithSync("MyData", 2.)
```

Similarly, the command to retrieve the content stored in a shared repository is:

```
ExtAPI.ExtensionMgr.CurrentExtension.UpdateAttributes()
v = ExtAPI.ExtensionMgr.CurrentExtension.Attributes["MyData"]
```

ACT-Based Properties

ACT has the ability to create customized objects that encapsulate ACT-based properties. This section discusses how to use ACT to create properties.

The following capabilities are discussed:

- [Creating Property Groups](#)
- [Parameterizing Properties](#)
- [Defining DesignXplorer Properties](#)

Creating Property Groups

This section focuses on the process of creating groups of properties.

The following methods are available:

- [Using <PropertyGroup> and <PropertyTable> Blocks](#)
- [Using Templates to Create Property Groups](#)

Using <PropertyGroup> and <PropertyTable> Blocks

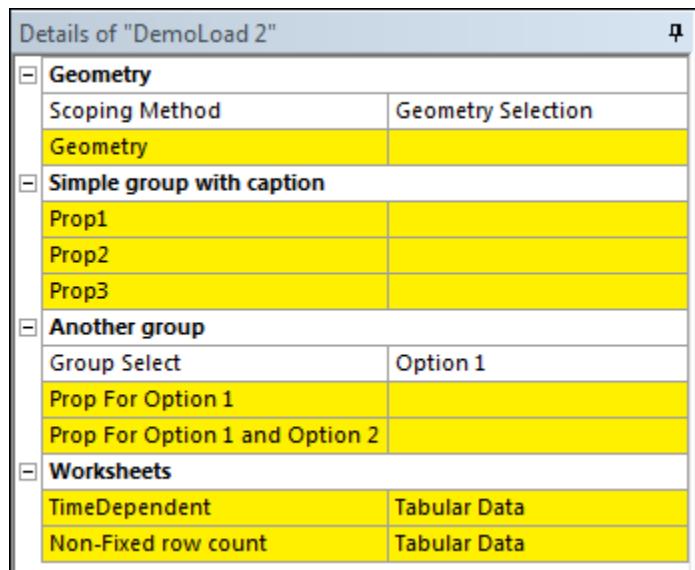
This section focuses on the process of creating groups of properties using <PropertyGroup> and <PropertyTable> blocks in the XML extension definition file.

The **<PropertyGroup>** and **<PropertyTable>** blocks are used to create groups of properties with a given caption. In this way, it is possible to manage dependencies between properties and to create worksheet views from a property.

- The **<PropertyGroup>** block encapsulates a list of child properties under one group.
- The **<PropertyTable>** block encapsulates a list of child properties under one table. Each child property creates a new column in the table. The user is able to control the line number of this table.

These functionalities are demonstrated in the extension **AdvancedProperties**.

The first example shows how to create a group of properties with a caption. The group can be collapsed or expanded by the user. The following figure shows the caption **Simple group with caption**.



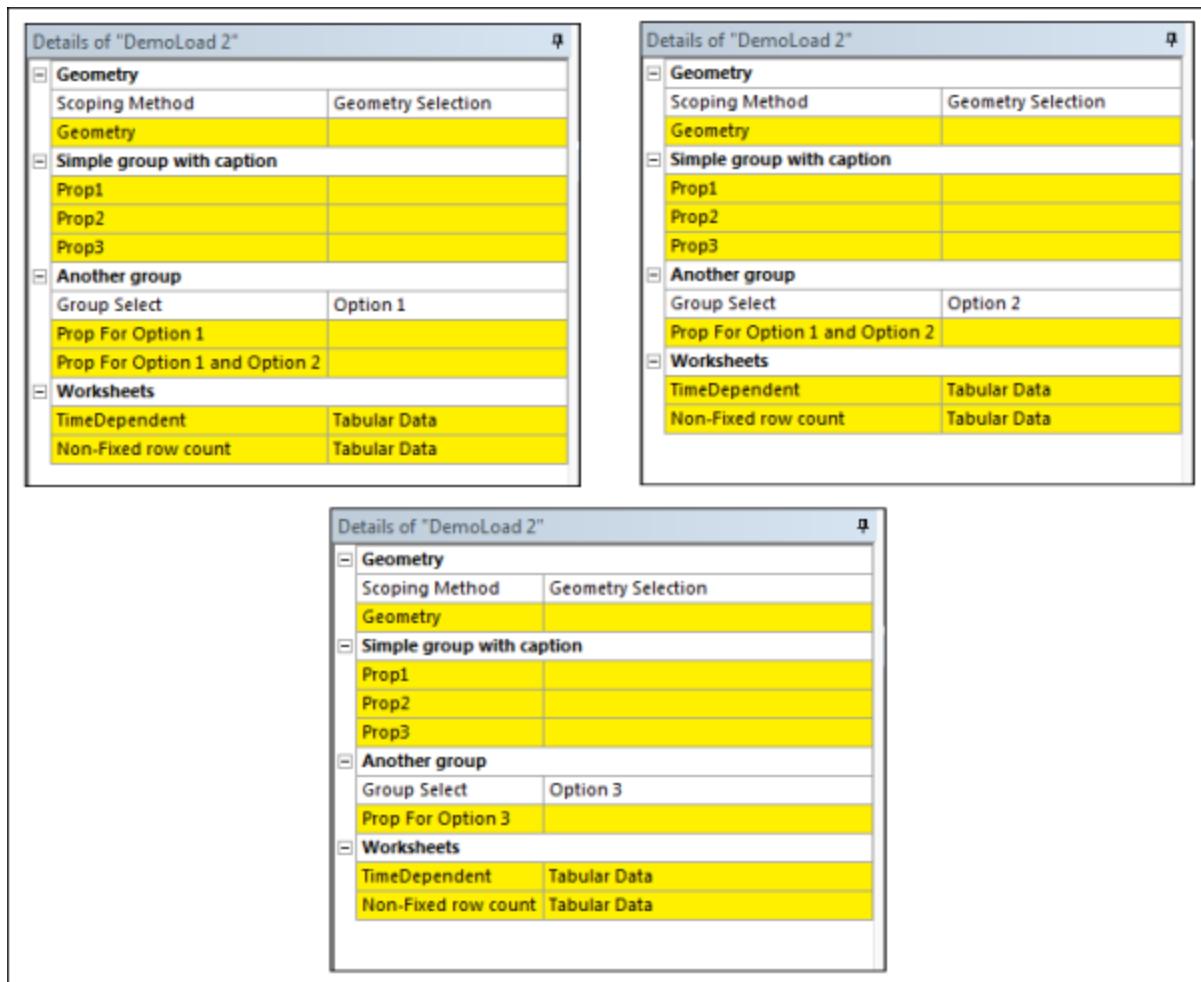
This group is created with the following XML code:

```
<propertygroup name="Group1" caption="Simple group with caption" display="caption">
  <property name="Prop1" caption="Prop1" control="text" />
  <property name="Prop2" caption="Prop2" control="text" />
  <property name="Prop3" caption="Prop3" control="text" />
</propertygroup>
```

The property group has a special attribute, **display**. In this case, **display** is set to **caption**, which means all child properties are displayed under the caption. If the **display caption** is omitted, **display** is set to **hidden**, which means that the property group is hidden.

The second example illustrates how to show or hide properties according to the value of another selected property.

As shown in the following figure, the visibility of the properties depends on the value of property **Group Select**.



This group is created with the following XML code, which creates dependencies between properties:

```

<propertygroup name="Group2" caption="Another group" display="caption">
  <propertygroup name="Group3" caption="Group Select" display="property" control="select"
default="Option 1">
    <attributes options="Option 1,Option 2,Option 3" />
    <property name="Prop1" caption="Prop For Option 1" visibleon="Option 1" control="text" />
    <property name="Prop2" caption="Prop For Option 1 and Option 2"
visibleon="Option 1|Option 2" control="text" />
    <property name="Prop3" caption="Prop For Option 3" visibleon="Option 3" control="text" />
  </propertygroup>
</propertygroup>

```

In this case, the attribute **display** is set to **property**. The **PropertyGroup** named **Group3** defines a standard ACT property that provides additional capabilities for all child properties.

Each child property can specify an attribute **visibleon**, which can take a value or a set of values. If the current value of the parent property fits with the attribute **visibleon**, the property is displayed. Otherwise, the property is hidden.

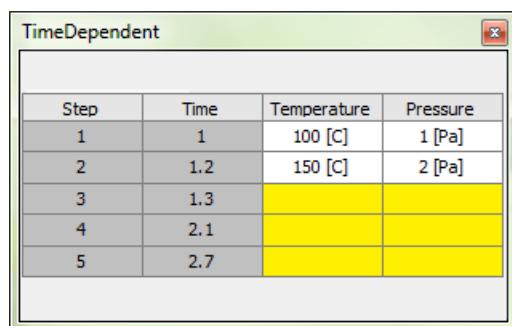
The following example demonstrates how to create properties that open a worksheet each time the user needs access to the content of the properties. The type **PropertyTable** is used for such an application. This property type is of particular interest as it allows you to create a worksheet that exposes a set of properties for your customization.

To facilitate the development, ACT already provides some predefined worksheets. Two different types of worksheets are currently available:

- Time-dependent
- Non-fixed row dimension

Time-dependent Worksheets

In a time-dependent worksheet, the row number is initialized with the number of steps defined in the object **AnalysisSettings**. If the user of the extension adds or removes time steps in the object **AnalysisSettings**, the worksheet is automatically updated. Consequently, this type of worksheet represents an efficient way to manage time-dependent data within an extension.



Step	Time	Temperature	Pressure
1	1	100 [C]	1 [Pa]
2	1.2	150 [C]	2 [Pa]
3	1.3		
4	2.1		
5	2.7		

This worksheet is created with the following XML code:

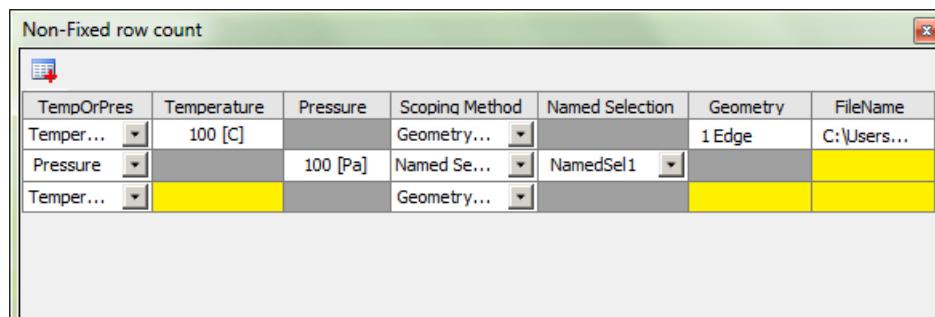
```
<propertytable name="TimeDep" caption="TimeDependent" display="worksheet" control="applycancel"
    class="Worksheet.TimeFreqTabularData.TFTabularData">
    <property name="Step" caption="Step" control="integer" readonly="true" />
    <property name="Time" caption="Time" control="float" readonly="true" />
    <property name="Temperature" caption="Temperature" unit="Temperature" control="float"></property>
    <property name="Pressure" caption="Pressure" unit="Pressure" control="float"></property>
</propertytable>
```

In this example, the attribute **display** is set to **worksheet**. In addition, the attribute **class** is used to specify the name of the IronPython class that manages the worksheet. For this application, the class **TFTabularData** is used. This class is defined in the script **TimeFreqTabularData.py**, which is located in the folder **libraries/Mechanical/Worksheet** that is part of the standard ACT installation.

The properties **Step** and **Time** integrate a specific treatment, as they are automatically populated with the information specified in the object **AnalysisSettings**. These two properties are optional.

Non-fixed Row Dimension Worksheets

In a non-fixed row dimension worksheet, the array is empty by default. You can add a new line by clicking the top left button.



TempOrPres	Temperature	Pressure	Scoping Method	Named Selection	Geometry	FileName
Temper...	100 [C]		Geometry...		1 Edge	C:\Users...
Pressure		100 [Pa]	Named Se...	NamedSel1		
Temper...			Geometry...			

This worksheet is created with the following XML code:

```
<propertytable name="Worksheet" caption="Non-Fixed row count" display="worksheet" control="applycancel"
  class="Worksheet.PropertyGroupEditor.PGEeditor">
<propertygroup name="TempOrPres" caption="TempOrPres" display="property" control="select"
  default="Temperature">
  <attributes options="Temperature,Pressure" />
  <property name="Temperature" caption="Temperature" unit="Temperature" control="float"
    visibleon="Temperature"></property>
  <property name="Pressure" caption="Pressure" unit="Pressure" control="float"
    visibleon="Pressure"></property>
</propertygroup>

<property name="Scoping" caption="Scoping" control="scoping">
  <attributes selection_filter="face|edge" />
</property>
<property name="FileName" caption="FileName" control="fileopen">
  <attributes filters="Command files (*.bat)|*.bat|All files (*.*)|*.*" />
</property>
</propertytable>
```

In this example, the class **PGEeditor** is used. The class **PGEeditor** is defined in the script **PropertyGroupEditor.py**, which is available in the folder **/libraries/Mechanical/Worksheet**.

You can access the content of the worksheet in the same manner as you do any other standard ACT property.

Using Templates to Create Property Groups

A template represents a generic method for defining a group of properties with the associated callback code. You can create a template to provide services that can be integrated into any extension. For example, you can build a template to generate a property that displays all the available coordinate systems for the current model. Advanced users find templates to be beneficial. You can enrich the templates currently integrated in ACT to customize the environment in a very efficient way.

In your ANSYS installation directory, **controltemplates.xml** for Mechanical is located here:
...\\Addins\\ACT\\templates\\Mechanical. The content of this file includes several templates, each of which is assigned a name. The following template is named **coordinatesystem_selection**.

```
<!-- Coordinate System Selection -->
<controltemplate name="coordinatesystem_selection" version="2">
  <property name="selectCoordinateSystem" control="select"
    class="templates.select.SelectCoordinateSystem">
  </property>
</controltemplate>
```

The class **SelectCoordinateSystem** associated with this template is defined in the script **select.py**, located in **...\\Addins\\ACT\\libraries\\Mechanical\\templates**. A template has to be made of one single property. If several properties need to be defined, they must be integrated in a group. In the same folder, the script **scoping.py** supplies a template definition with different properties.

To link this template to a property, you fill the attribute **control** for the property with the name of the template.

Parameterizing Properties

When defining ACT properties in ANSYS Workbench or in the Mechanical and DesignModeler products, you can also define the property value as either an input or an output parameter. All ACT parameters are inputs, unless you specify otherwise in the definition.

Once parameterized, the property is added to the **Parameter Set** bar in the Workbench **Project Schematic**, where it behaves and is treated as any other parameter.

You can incorporate parameters in a variety of places in your analysis.

- In ANSYS Workbench, parameters can be defined for custom task-level properties in the **Project Schematic** workflow.
- In ANSYS Mechanical, parameters can be defined for any ACT object, regardless of its location in the tree.
- In ANSYS DesignModeler, parameters can be incorporated into a number of custom ACT-based features, such as renaming selected bodies, specifying geometry generation details like material, dimensions, and so on.
- In a third-party solver implemented by ACT, parameters can be incorporated into the load, analysis settings, and result.

All combinations of ACT-based and standard parameters are supported:

- ACT inputs and standard outputs
- ACT inputs and ACT outputs
- Standard inputs and ACT outputs

The following parameterization topics are discussed:

[Common Property Parameterization Processes](#)

[Parameterizing Properties in ANSYS Workbench](#)

[Parameterizing Properties in ANSYS Mechanical](#)

[Parameterizing Properties in ANSYS DesignModeler](#)

[Parameterizing Properties in a Third-Party Solver](#)

Common Property Parameterization Processes

To define a property as a parameter, you must add the attribute **isparameter** to the XML extension definition file and set it to **true**. By default, a property is created as an input parameter. To further specify that the property is to be an output parameter, you must also add the attribute **readonly** to the XML file and set it to **true**. The following sections use sample code segments to illustrate the creation of input and output parameters.

When you define a property as a parameter, it is not parameterized by default. Defining a property as a parameter only makes parameterization possible by adding a check box to the user interface. To actually parameterize the property, the user must select the check box provided by the ACT attribute.

Once a property has been selected for parameterization, it is automatically sent to the **Parameter Set** bar on the **Project Schematic** in Workbench. It can be viewed in both the **Outline** and **Table** views for the **Parameter Set** bar. While output parameters are read-only, input parameters from ACT can be defined in the same way as any other input parameter. For example, you can:

- Change the unit of the parameterized input property. The unit proposed respects the type of unit specified in the XML extension definition file.
- Add a design point for the input parameter by clicking in the bottom row of the design points table.

- Modify the value of a design point by selecting an input parameter and entering a new value. However, the type of value must respect the one specified in the XML extension definition file. As such, design point values in the previous value must be floats. They cannot be strings.

There should be at least one input parameter and one output parameter. When finished setting up the system and working with parameters and design point values, you can solve the problem by updating the design points for the **Parameter Set** bar.

Parameterizing Properties in ANSYS Workbench

In ANSYS Workbench, you can define ACT-based task properties either as input parameters, output parameters, or non-parameterized values. The following sections describe how to implement input and output parameters on ACT objects within an ANSYS Workbench **Project Schematic** workflow.

To define either an input parameter or an output parameter in your schematic workflow, add the attribute **isparameter** to the XML extension definition of the task-level property that you want to parameterize. To illustrate, the extension **DataSquares** is used.

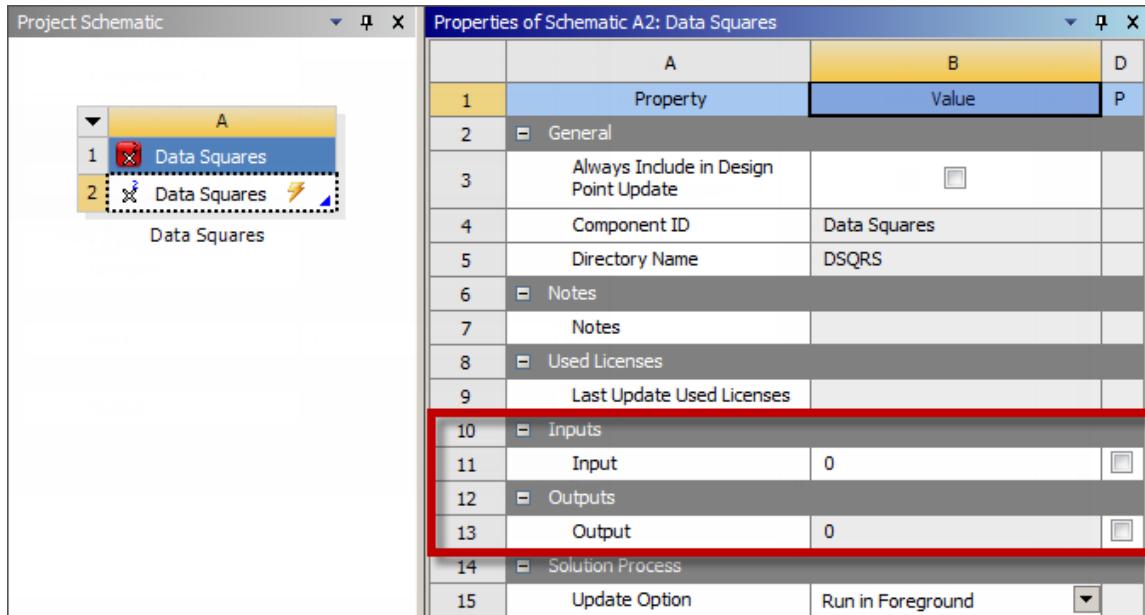
For example, the following code segment creates and parameterizes a task-level input property named **Input**. Because it is an input, the attribute **readonly** is set to **False**. The property also has the attribute **control** set to **integer**.

```
...
<propertygroup name="Inputs">
  <property name="Input" caption="Input" control="integer" default="0" readonly="False"
    needupdate="true" visible="True" persistent="True" isparameter="True" />
</propertygroup>
...
```

In the same way, the following code segment creates and enables parameterization for a task-level output property named **Output**. Because it is an output property, the attribute **readonly** is set to **true**. The property also has the attribute **control** set to **integer**.

```
...
<propertygroup name="Outputs">
  <property name="Output" caption="Output" control="integer" default="0" readonly="True"
    visible="True" persistent="True" isparameter="True" />
</propertygroup>
...
```

A custom system named **DataSquares** is created on the **Project Schematic**. The new properties are shown in the custom **Inputs** and **Outputs** sections exposed in the task properties. These custom sections are defined by the task groups.



When you select the check box for parametrizing the two new properties, they are added to the **Parameter Set** bar.

For a more information about this example, see [Custom, Lightweight, External Application Integration with Custom Data and Remote Job Execution \(p. 291\)](#).

Parameterizing Properties in ANSYS Mechanical

The following sections provide examples and methods for integrating ACT parameters into your ANSYS Mechanical model.

In Mechanical, it is possible to define ACT-based property values as either input parameters or output parameters. This section addresses how to implement a parameter on an ACT object in Mechanical. Specifically, it describes how to parameterize loads, results, and user objects. ACT objects in any of these categories behave and interact with parameters in exactly the same way.

The following sections provide examples and methods for integrating ACT parameters into your Mechanical model.

[Defining Input Parameters in ANSYS Mechanical](#)

[Defining Output Parameters in ANSYS Mechanical](#)

Defining Input Parameters in ANSYS Mechanical

To define a property as a parameter, add the attribute **isparameter** to the XML extension definition file and set it to **true**.

For example, the following code segment creates a property named **float_unit**. The property is a pressure and its value is a float.

```
<property name="float_unit" caption="float_unit" control="float" unit="Pressure" isparameter="true" />
```

Note

Other attributes such as **default** and **isload** could be added as well.

The attribute **isparameter** adds a check box to the user interface, making it possible for the user to select the property for parameterization.

Details of "CustomPressure"	
Geometry	
Scoping Method	Geometry Selection
Geometry	
Definition	
<input checked="" type="checkbox"/> float_unit	
<input type="checkbox"/> float	125
<input type="checkbox"/> float_default	123.4
<input type="checkbox"/> float_default_READONLY	123.4
<input type="checkbox"/> integer_DEFAULT_READONLY	100
<input type="checkbox"/> integer_DEFAULT	100
<input type="checkbox"/> integer	1

Once the property has been selected for parameterization, it automatically displays in both the **Outline** and **Table** views for the **Parameter Set** bar.

Outline of All Parameters				
	A	B	C	D
1	ID	Parameter Name	Value	Unit
2	Input Parameters			
3	Static Structural (A1)			
4	P1	CustomPressure float_unit	12.3	Pa
*	New input parameter	New name	New expression	
6	Output Parameters			
7	Static Structural (A1)			
8	P2	Total Deformation Maximum	946.13	m
*	New output parameter		New expression	
10	Charts			

Table of Design Points				
	A	B	C	D
1	Name	P1 - CustomPressure float_unit	P2 - SolutionObject DisplacementMaximum	E
2	Units	Pa	m	Export Note
3	Current	120.6	⚡	
4	DP 1	130.95	⚡	
5	DP 2	111.9	⚡	
*				

The following code is extracted from the XML file for the previous example. You can see how the definition of the property **float_unit** is incorporated into the file.

```

<load name="CustomPressure" version="1" caption="CustomPressure" icon="tload" support="false"
isload="true" color="#0000FF">
<callbacks>
<getsolvecommands order="1">writeNodeId</getsolvecommands>
</callbacks>
<property name="Geometry" caption="Geometry" control="scoping">
<attributes selection_filter="face"/>
</property>
<property name="float_unit" caption="float_unit" control="float" unit="Pressure" isparameter="true"/>
</load>

```

Defining Output Parameters in ANSYS Mechanical

It is also possible to define an ACT property as an output parameter.

The process for making a property available as a parameter is the same as for any input parameter. You add the attribute **isparameter** and set it to **true**. A check box allowing parameterization of the

property then becomes available. To specify that the property is to be an output parameter, you must also add the attribute **readonly** to the XML extension definition file and set it to **true**.

For example, the following code segment creates a property named **MyOutPutProperty**. As in the previous example, the property is a pressure and its value is a float. However, because this property has the attribute **readonly** set to **true**, it can be parameterized only as an output parameter.

```
<property name="MyOutPutProperty" caption="MyOutPutProperty" control="float" unit="Pressure"
readonly="true" isparameter="true"/>
```

Note

Other attributes such as **default** and **isload** could be added as well.

Again, the user must select the check box provided by the **ACT** attribute to parameterize the property. Once the property has been selected for parameterization, the corresponding output is automatically generated in the **Outline** view for the **Parameter Set** bar.

Outline of All Parameters				
	A	B	C	D
1	ID	Parameter Name	Value	Unit
2	Input Parameters			
3	Static Structural (A1)			
4	P2	CustomPressure float_unit	0	Pa
*	New input parameter	New name	New expression	
6	Output Parameters			
7	Static Structural (A1)			
8	P1	MyResult MyOutPutProperty		Pa
*	New output parameter		New expression	
10	Charts			

This is the output parameter coming from the property named **MyOutPutProperty**.



In addition, the minimum and maximum values of an ACT result object are available to become output parameters by default. This capability is not managed by the ACT extension, but takes advantage of the Mechanical environment.

Parameterizing Properties in ANSYS DesignModeler

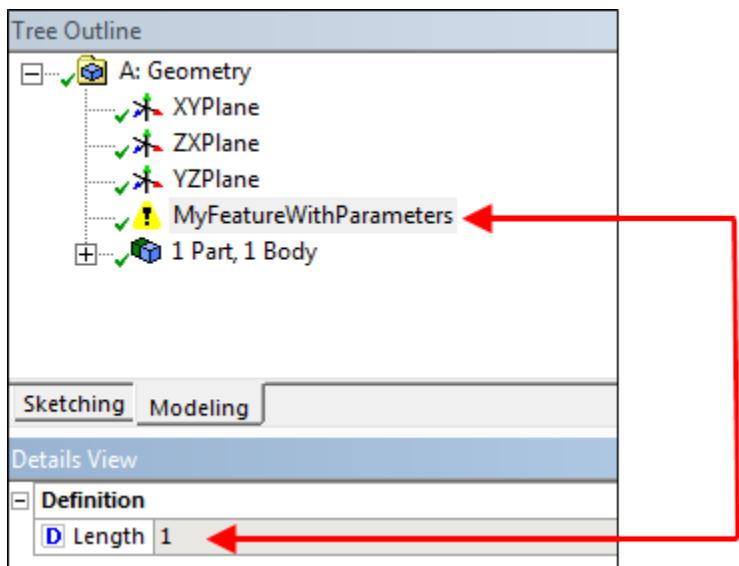
In ANSYS DesignModeler, it is possible to define ACT-based property values only as input parameters. This section addresses how to implement an input parameter on an ACT object in DesignModeler.

As with any other input parameter, to add an input to DesignModeler you must add the parameter **isparameter** to the XML extension definition file and set it to **true**. However, you are not able to add output parameters. The attribute **readonly** is not available for DesignModeler.

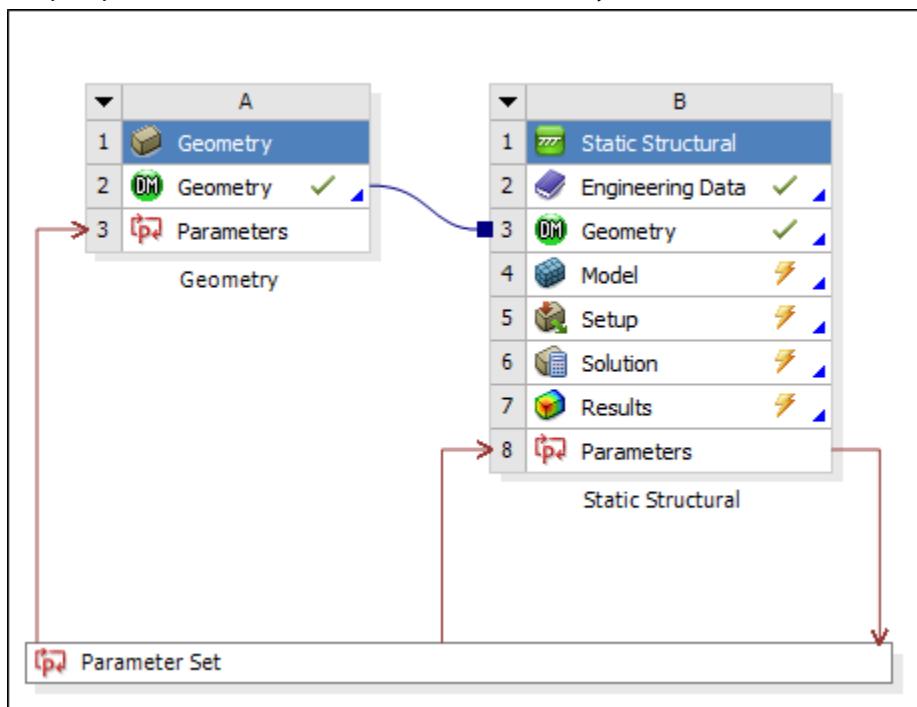
For example, in an ACT feature that generates a cylinder, you can add length as a parameter. The following excerpt from the XML file defines the parameter **Length**.

```
<geometry name="MyFeatureWithParameters" caption="MyFeatureWithParameters"
icon="Construction" version="1">
<callbacks>
  <ongenerate>generateMyFeature</ongenerate>
</callbacks>
<property name="Length" caption="Length" control="float" isparameter="true"></property>
</geometry>
```

In DesignModeler, the check box next to the property **Length** allows you to parameterize it as an input parameter.



You could add a static structural analysis based on the previous **Geometry** template by adding an output parameter in the **Static Structural** analysis. This results in the following schematic workflow:



The input parameter defined in DesignModeler is managed in exactly the same way as other any other input parameter. In the **Outline** view for the **Parameter Set** bar, the geometry input parameter is placed under the **Geometry** system.

Outline of All Parameters				
	A	B	C	D
1	ID	Parameter Name	Value	Unit
2	Input Parameters			
3	Geometry (A1)			
4	P1	MyFeatureWithParameters.Length	1	
*	New input parameter	New name	New expression	
6	Output Parameters			
7	Static Structural (B1)			
8	P2	Total Deformation Maximum	⚡	m
*	New output parameter		New expression	
10	Charts			

The custom geometry parameter is placed under the **Geometry** system.

Parameterizing Properties in a Third-Party Solver

When you have used ACT to deploy a third-party solver, it is possible to define ACT-based properties as either input parameters or output parameters. This section addresses how to implement a parameter on an ACT object in an external third-party solver—specifically, under a load, in the analysis settings, and under results.

The definition of parameters for a third-party solver is no different than the process described in earlier sections: To parameterize a property, in the property definition you must add the attribute **isparameter** and set it to **true**. By default, it is an input parameter. To define it as an output parameter, you must set the attribute **readonly** to **true**.

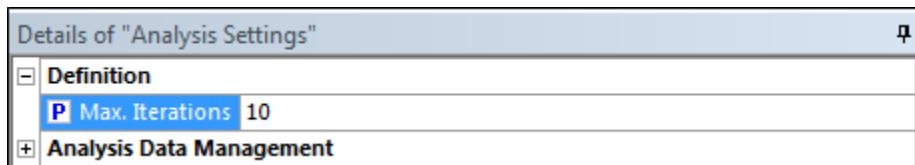
Defining Parameters under a Load in a Third-Party Solver

The process of defining parameters under a load in ANSYS DesignModeler is identical to the process described in [Defining Input Parameters in ANSYS Mechanical \(p. 54\)](#).

Defining Parameters in Analysis Settings in a Third-Party Solver

It is possible to parameterize the **Analysis Settings** in a third-party external solver. The settings options available depend on the definition of the third-party solver.

For example, you can opt to parameterize the maximum number of iterations in the analysis settings:



The following code segment, placed under the solver section of the XMLfile, defines the property **MaxIter**, for which the attribute **caption** is set to **Max. Iterations**.

```
<solver...
    <property name="MaxIter" caption="Max. Iterations" control="integer" isparameter="true" default="10" />
/>
```

Defining Parameters under Results in a Third-Party Solver

The process of defining parameters under **Results** in ANSYS Mechanical is identical to the process described in [Parameterizing Properties in ANSYS Mechanical \(p. 54\)](#). Results parameters can be either inputs or outputs, depending on whether the attribute **readonly** is set to **true** or **false**.

Defining DesignXplorer Properties

An external sampling or optimization method can expose settings to allow the user to control algorithm options and view outputs.

For instance:

- A Design of Experiment (DOE) can expose the editable setting **Number of Levels** to allow the user to define this limit. The sampling is responsible for handling this setting as needed. The DOE can also expose the read-only setting **Actual Number of Points** to inform the user of the number of points generated once the sampling is completed.
- An optimizer can expose the editable setting **Maximum Number of Iterations** to allow the user to define this limit. The optimizer is responsible for handling this setting as needed. The optimizer can also expose the read-only setting **Actual Number of Iterations** to inform the user of the number of iterations actually performed once the optimization is completed.

ACT provides support for the general definition of properties with attributes and callbacks. Each setting to expose to the DesignXplorer user must be declared and defined in the XML extension definition file as a **<property>** block under the **<sampling>** or **<optimizer>** block.

```
<property name="[property internal name (string)]"
    caption="[property display name (string)]"
    readonly="[true | false(default)]"
    default="[default value]"
    control="[text(default) | select | float | integer | ...]"
    visible="[true(default) | false]"
    visibleon="[values(separator '|'))]"
    ...
    <attributes>mldr</attributes>
    <callbacks>
</property>
```

Optionally, settings can be placed together in groups. Each group must be declared and defined in the XML file as a **<propertygroup>** block under the **<sampling>** or **<optimizer>** block.

For a description of common attributes and callbacks, see "property" in the "XML Tags" section of the [ANSYS ACT API Reference Guide](#) for DesignXplorer. For more information on settings groups, see "propertygroup" in the [ANSYS ACT XML Reference Guide](#).

Properties in the DesignXplorer Interface

An external method can be selected in DesignXplorer for **Design of Experiments Type** for a DOE or **Method Name** for an optimization. When an external method is selected, its properties are shown in the **Properties** view.

For example, the following figure shows the **Properties** view when the external sampling named **Full Factorial** is selected for **Design of Experiments Type**.

Properties of Outline : Method		
	A	B
1	Property	Value
2	Design Points	
3	Preserve Design Points After DX Run	<input type="checkbox"/>
4	Failed Design Points Management	
5	Number of Retries	0
6	Design of Experiments	
7	Design of Experiments Type	Full Factorial
8	General	
9	Number of Levels	3
10	Sampling Status	
11	Number of Points	0
12	Log File	

This next figure shows the **Properties** view when the external optimizer named **Python Optimizer** is selected for **Method Name**.

Properties of Outline : Method		
	A	B
1	Property	Value
2	Design Points	
3	Preserve Design Points After DX Run	<input type="checkbox"/>
4	Failed Design Points Management	
5	Number of Retries	0
6	Optimization	
7	Method Name	Python Optimizer
8	General	
9	My Number of Samples	50
10	My Maximum Number of Candidates	2
11	My Output Results	CandidatesAndSamplesAndParetos
12	Group Select	Option 1
13	Prop For Option 1	
14	Prop For Option 1 and Option 2	
15	Optimization Status	
16	My Number of Evaluations	0
17	Log File	

By default, properties are shown under the **General** category. If a **propertygroup** is specified, the corresponding category is created in the property view. The properties under the **General** and respective **Status** categories are specific to the external method and are defined in the XML file.

If the sampling supports the **LogFile** capability, the property **LogFile** is automatically defined under the **Status** category in the **Properties** view. Once the sampling is generated or the optimization has been run, the user can view the log file in a dialog box by clicking the available link.

Additional Attributes for DesignXplorer Extensions

In the context of sampling and optimization extensions, DesignXplorer also recognizes the additional attributes **min**, **max**, and **advanced** for each property. The attributes **min** and **max** are used to specify a minimum value, maximum value, or both values for a double or integer property.

In the following sampling example, the minimum value is **1** and the maximum value is **100**.

```
<property name="NumberOfLevels" caption="Number of Levels" control="integer" default="3">
    <attributes min="1" max="100"/>
</property>
```

In the following optimization example, the minimum value is **2** and the maximum value is **200**.

```
<property name="MyNumberOfSamples" caption="My Number of Samples" control="integer" default="50">
    <attributes min="2" max="200"/>
</property>
```

The attribute **advanced** is used to classify the property as an advanced option. Advanced options are visible only if activated. To activate them, in the **Options** dialog box, select the **Show Advanced Options** check box on the **Design Exploration** tab.

The following example is applicable to either type of DesignXplorer extension.

```
<property name="MyRandomNumber" caption="My Random Number provider" control="select"
default="Python">
    <attributes options="Python,CLibrary" advanced="true"/>
</property>
```

Advanced Usage Examples

The following advanced examples illustrate the use of properties in DesignXplorer extensions.

Managing Dependencies between Properties

The following example demonstrates how to manage the dependency between properties:

- **Prop1** is visible when the value **Group3** is equal to **Option 1**
- **Prop2** is visible when the value **Group3** is equal to **Option 1** or **Option 2**
- **Prop3** is visible when the value **Group3** is equal to **Option 3**

```
<propertygroup name="Group3" caption="Group Select" display="property" control="select"
default="Option 1">
    <attributes options="Option 1,Option 2,Option 3" />
    <property name="Prop1" caption="Prop For Option 1" visibleon="Option 1" control="text" />
    <property name="Prop2" caption="Prop For Option 1 and Option 2" visibleon="Option 1|Option 2"
control="text" />
    <!-- Prop3 is applicable and visible only when Group3 == "Option 3" -->
    <property name="Prop3" caption="Prop For Option 3" visibleon="Option 3" control="text" />
</propertygroup>
```

Controlling Property Visibility with a Callback

Property attributes, such as **visibility**, can be modified dynamically by using a callback.

DOE Example

The following example shows how to control the visibility of the property **PropForSingleInput** with a callback coded in IronPython:

```
<property name="PropForSingleInput" caption="My Property for single input parameter"
control="text">
    <callbacks>
        <isvisible>isVisibleForSingInput</isvisible>
    </callbacks>
</property>
```

Where the function **isVisibleForSingInput** is defined by the following IronPython code:

```
def isVisibleForSingInput(entity,property):
    if entity.NumberOfObjectivesDefined <= 1:
        return True
    return False
```

The function **isVisibleForSingInput** takes two arguments:

- The first argument-named entity is of type **DXUserSampling**.
- The second argument-named property is of type **SimProperty**.

This function returns **True** when only one input parameter is defined.

Optimization Example

The following example shows how to control the visibility of the property **PropForSingleObjective** with a callback coded in IronPython:

```
<property name="PropForSingleObjective" caption="My Property for single-objective optimization"
control="text">
    <callbacks>
        <isvisible>isVisibleForSOO</isvisible>
    </callbacks>
</property>
```

Where the function **isVisibleForSOO** is defined by the following IronPython code:

```
def isVisibleForSOO(entity,property):
    if property.Name != PropForSingleObjective
        return True
    if entity.NumberOfObjectivesDefined <= 1:
        return True
    return False
```

The function **isVisibleForSOO** takes two arguments:

- The first argument-named entity is of type **DXUserOptimizer**.
- The second argument-named property is of type **SimProperty**.

This function returns **True** when only one objective is defined.

Modifying an Attribute with a Callback

The following examples show how you can modify the values property attributes when input parameters are edited by the user.

DOE Example

This example shows how you can modify the values of the attribute **max** for the property **NumberOfLevels** when input parameters are edited by the user.

Given the partial sampling definition:

```
<sampling ...>
  <callbacks>
    <InputParametersEdited>InputParametersEdited</InputParametersEdited>
  </callbacks>
  <property name="NumberOfLevels" caption="Number of Levels" control="integer" default="3"/>
    <attributes min="1">
  </property>
</sampling>
```

The function **InputParametersEdited** is defined by the following IronPython code:

```
def InputParametersEdited(entity):
    entity.Properties["NumberOfLevels"].Attributes["max"] = 3 * entity.NumberOfInputParametersDefined
```

The function **InputParametersEdited** is called when a user edits the input parameters. This function calculates the new minimum allowed value for the property **NumberOfLevels** from the current number of input parameters and then sets this value to the attribute **max**.

Optimization Example

This example shows how you can modify the values of the attribute **min** for the property **MyNumberOfSamples** when input parameters are edited by the user.

Given the partial optimizer definition:

```
<optimizer ...>
  <callbacks>
    <InputParametersEdited>InputParametersEdited</InputParametersEdited>
  </callbacks>
  <property name="MyNumberOfSamples" caption="My Number of Samples" control="integer" default="50" />
</optimizer>
```

The function **InputParametersEdited** is defined by the following IronPython code:

```
def InputParametersEdited(entity):
    entity.Properties["MyNumberOfSamples"].Attributes["min"] = 10 * entity.NumberOfInputParametersDefined
```

The function **InputParametersEdited** is called when input parameters are edited by the user. It calculates the new minimum allowed value for the property **MyNumberOfSamples** from the current number of input parameters and then sets this value to the attribute **min**.

Capabilities for ANSYS Mechanical

The following capabilities are discussed:

- [Adding a Preprocessing Feature in ANSYS Mechanical](#)
- [Adding a Postprocessing Feature in ANSYS Mechanical](#)
- [Creating Results with Imaginary Parts](#)
- [Responding to a Change to the Active Unit System](#)
- [Obsolete Callbacks "OnStartEval" and "GetValue"](#)

Connecting to a Third-Party Solver

Note

The examples used in this section are included in the package **ACT Developer's Guide Examples**. You can download this package from the [ACT Resources](#) page on the [ANSYS Customer Portal](#). To display the page, select **Downloads > ACT Resources**. To download the package, expand the **Help & Support** section and click **ACT Developer's Guide Examples** under **Documentation**.

Adding a Preprocessing Feature in ANSYS Mechanical

This section describes customization for ANSYS Mechanical only as this targeted product needs to expose preprocessing features natively.

Thus far the discussion has been focused on extending the user interface by adding toolbars. This section discusses how to use toolbars to perform meaningful operations such as adding a preprocessing feature to Mechanical. The example used in the discussion is defined in the extension DemoLoad. This extension, which adds a generic load to a project, is for demonstration purposes only.

The XML extension definition file DemoLoad.xml follows. As in previous examples, this file first defines the extension using the attributes **version** and **name**. The path to the IronPython script main.py is specified by the **<script>** block. The definition of the toolbar and the buttons is done in the **<interface>** block. The callback function **CreateDemoLoad** is used to create and add the load to the simulation environment.

```

<extension version="1" minorversion="0" name="DemoLoad">
  <guid shortid="DemoLoad">7dfd2b34-dfe1-469d-b244-1c7e5c222e54</guid>
  <script src="main.py" />
  <interface context="Mechanical">
    <images>images</images>
    <toolbar name="Loads" caption="Loads">
      <entry name="DemoLoad" icon="tload">
        <callbacks>
          <onclick>CreateDemoLoad</onclick>
        </callbacks>
      </entry>
    </toolbar>
  </interface>

  <simdata context="Mechanical">
    <load name="DemoLoad" version="1" caption="DemoLoad" icon="tload" color="#00FFFF">
      <callbacks>
        <getnodalvaluesfordisplay>GetNodalValuesForDisplay_DL</getnodalvaluesfordisplay>
        <getprecommands>GetPreCommands_DL</getprecommands>
        <getsolvecommands order="1">GetSolveCommands_DL</getsolvecommands>
      </callbacks>

      <property name="Geometry" caption="Geometry" control="scoping">
        <attributes selection_filter="edge" />
      </property>
      <property name="Text" caption="Text" control="text"></property>
      <property name="SelectStatic" caption="Select (static)" control="select">
        <attributes options="Option 1,Option 2,Option 3" />
      </property>
      <property name="SelectDynamic" caption="Select (dynamic)" control="select">
        <callbacks>
          <onactivate>StringOptions_DL</onactivate>
        </callbacks>
      </property>
    </load>
  </simdata>
</extension>

```

```

<property name="Double" caption="Double" unit="Length" control="float"
default="1 [m]"></property>

</load>
</simdata>
</extension>
```

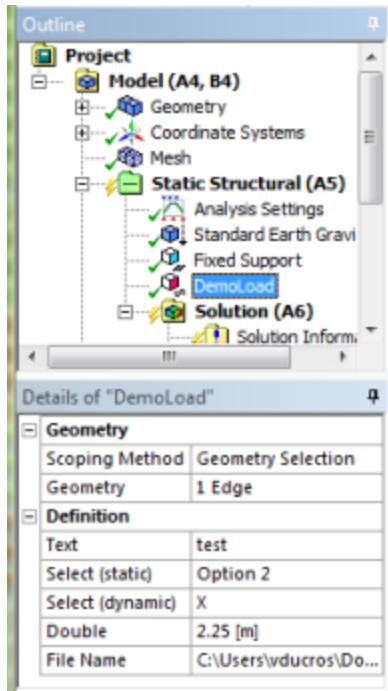
The definition of the load is encapsulated in the `<simdata>` block. The attributes in the child `<load>` block provide the name, version, caption, icon, and color that apply to this load. The attribute `color` is defined in a hexadecimal format. This color is used to contrast the load when it is displayed on the model. Its child `<callbacks>` block encapsulates three callbacks: `<getnodalvaluesfordisplay>`, `<getprecommands>`, and `<getsolvecommands>`.

- The callback `<getnodalvaluesfordisplay>` specifies the name of the function that is invoked when the load is displayed in the product.
- The callback `<getprecommands>` controls where the specific solver commands are inserted in the solver input file. This callback inserts the solver commands before the standard loads and boundary conditions defined in the product. For the ANSYS solver, the new commands are inserted in the context of the /PREP7 preprocessor after the mesh has been written.

The callback `<getsolvecommands>` specifies the name of the function that is invoked when the solver input file is generated by the product. Consequently, the related function is responsible for generating the APDL commands that describe the load within the ANSYS input file. When the callback `<getsolvecommands>` is invoked, the solver commands are inserted into the solver input file just before the SOLVE command.

In the callback definition, you can define the properties to apply to the actual definition of the load. These properties are displayed in the **Details** view of Mechanical, where the user provides the necessary values to complete the load definition. In the IronPython script, you see how the load's properties can be retrieved and modified.

The following figure shows that each property defined above appears in the **Details** view with the corresponding names and types of interface control.



The two properties, **Select (static)** and **Select (dynamic)**, have their attribute **control** set to **select**. The first property populates the list of options from the XML extension definition file, while the second one defines the callback `<onactivate>`. This callback, which is called when the control is activated, populates the available options for the property. This second approach provides full control of the options to expose in the drop-down menu and makes the list of options dependent on the current status of the project. Many different situations that can impact the content of the list of options can be addressed, as long as they are implemented in the callback `<onactivate>`.

The next property, **Double**, does not require the definition of one specific callback. Instead, the XML file introduces a physical quantity dependency with the option **unit**. This option specifies that this property is consistent with a length. In addition, a default value of **1 [m]** is introduced with the option **default**. This default value is exposed in the **Details** view each time a new load is created.

The IronPython script `main.py` for the extension `DemoLoad` follows.

```
import os
def init(context):
    ExtAPI.Log.WriteMessage("Init DemoLoads...")

def CreateDemoLoad(analysis):
    analysis.CreateLoadObject("DemoLoad")

def StringOptions_DL(load, prop):
    prop.ClearOptions()
    prop.AddOption("X")
    prop.AddOption("Y")
    prop.AddOption("Z")

def GetPreCommands_DL(load, stream):
    stream.WriteLine("//COM, ****" + "\n")
    stream.WriteLine("//COM, Load properties from DemoLoad getprecommands event" + "\n")
    stream.WriteLine("//COM, Text Property = " + load.Properties["Text"].ValueString + "\n")
    stream.WriteLine("//COM, SelectDynamic Property = " + load.Properties["SelectDynamic"].ValueString + "\n")
    stream.WriteLine("//COM, SelectStatic Property = " + load.Properties["SelectStatic"].ValueString + "\n")
    stream.WriteLine("//COM, Double Property = " + load.Properties["Double"].ValueString + "\n")
    stream.WriteLine("//COM, ****" + "\n")

def GetSolveCommands_DL(load, stream):
    stream.WriteLine("//COM, ****" + "\n")
```

```

stream.WriteLine("/COM, Load properties from DemoLoad getsolvecommands event" + "\n")
stream.WriteLine("/COM, Text Property = " + load.Properties["Text"].ValueString + "\n")
stream.WriteLine("/COM, SelectDynamic Property = " + load.Properties["SelectDynamic"].ValueString + "\n")
stream.WriteLine("/COM, SelectStatic Property = " + load.Properties["SelectStatic"].ValueString + "\n")
stream.WriteLine("/COM, Double Property = " + load.Properties["Double"].ValueString + "\n")
stream.WriteLine("/COM, *****" + "\n")

def GetNodalValuesForDisplay_DL(load, nodeIds):
    dval = load.Properties["Double"].Value
    coordselect = load.Properties["SelectDynamic"].ValueString
    mesh = load.Analysis.MeshData
    values = []
    for id in nodeIds:
        node = mesh.NodeById(id)
        dispval = float(0.0)
        if coordselect == "X":
            dispval = node.X * float(dval)
        elif coordselect == "Y":
            dispval = node.Y * float(dval)
        elif coordselect == "Z":
            dispval = node.Z * float(dval)
        else:
            dispval = float(0.0)
        values.Add(dispval)
    return values

```

The functions **GetNodalValuesForDisplay_DL** and **GetSolveCommands_DL** are critical to the behavior and application of the load. **GetNodalValuesForDisplay_DL** is called each time the graphics is refreshed. The required input arguments are **load** and **nodeIds**, where **load** is the load object for this load and **nodeIds** is a list of node identifiers.

In this example, **load.Properties["Double"].Value** queries for the "**Double**" property value. The argument **nodeIds** contain a list of node numbers on which one value has to be returned by the function. For every node in the list, the value of the property "**Double**" is assigned in the values array representing the output of the function. This output is subsequently treated by the graphics engine of the product so that the visualization on the FE model is available.

GetSolveCommands is intentionally simplified for this example. The prototype of this function is made of two input arguments, the load object and the filename in which the new specific solver commands are written. The output file is only a temporary file. The content is rewritten in the final solver input file to ensure that the specific commands related to the customized load are merged with all the other commands already defined by the standard features of the product.

You can also create a specific set of context menu options. These options use the property **Action**.

Actions are defined inside the **<callbacks>** block for the load.

```

<callbacks>

    <ongenerate>generate</ongenerate>
    <oncleardata>clearData</oncleardata>

    <action name="a1" caption="Action 1" icon="update">action1 </action>
    <action name="a2" caption="Action 2" icon="update">action2 </action>

</callbacks>

```

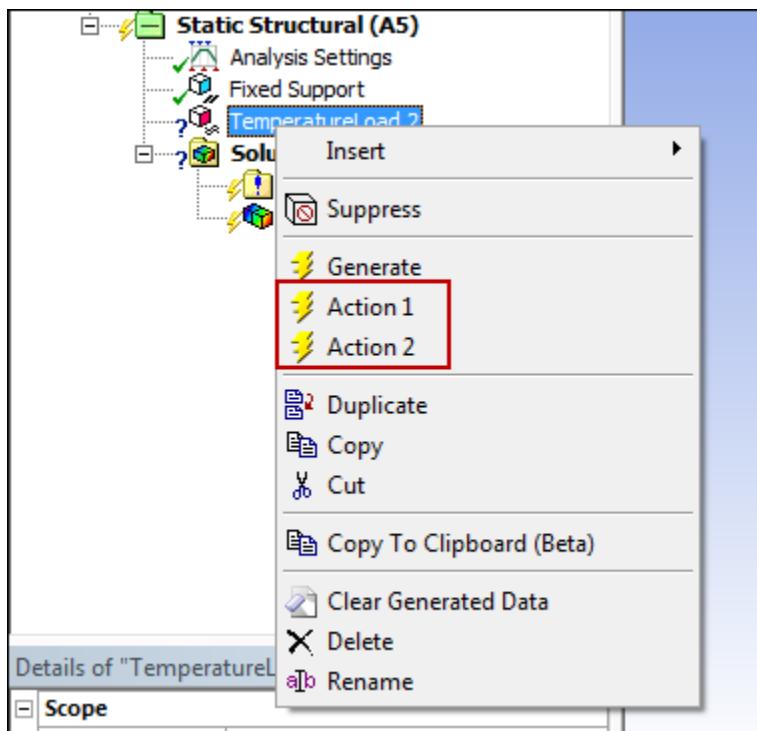
The **<action>** block takes three attributes:

- A name to identify the action.
- A caption to display in the context menu.

- The name of the image file to display as the icon. You must place the BMP file that Mechanical requires in the **images** directory specified in the extension's XML file.

The **<action>** block defines the name of the function that is called when the user selects the associated context menu option.

The following figure shows the result of the declaration. There are now two additional context menu items, **Action 1** and **Action 2**.



As illustrated, you can also add a **Generate** context menu option. This option derives from the standard **Generate** action provided by Mechanical. For that reason, the declaration of this particular option differs from the declaration of the options **Action1** and **Action2**. This option is always associated with the **Clear Generated Data** option.

These options allow you to create a load that can mimic a standard imported load. The callback associated with the **Generate** option replaces the standard function integrated in Mechanical.

The feature is activated when you define the callback **<ongenerate>** for the load.

The callback **<ongenerate>** is invoked each time the **Generate** context menu option is selected. It is also invoked during a solve if the state of the load is set to "not solved."

As for the standard imported load object, the callback **<ongenerate>** is called only if the mesh is already generated.

```

<callbacks>

  <ongenerate>generate</ongenerate>
  <oncleardata>clearData</oncleardata>

  <action name="a1" caption="Action 1" icon="update">action1 </action>
  <action name="a2" caption="Action 2" icon="update">action2 </action>

</callbacks>

```

The associated IronPython code looks like this:

```

def generate(load, fct):
    pct = 0
    fct(pct,"Generating data...")

    propEx = load.PropertyByName( "Expression" )
    exp = propEx.Value
    if exp=="":
        return False
    try:
        vexp = compile(exp,'','eval')
    except:
        return False

    values = SerializableDictionary[int,float]()
    nodeIds = []

    propGeo = load.PropertyByName( "Geometry" )
    refIds = propGeo.Value
    mesh = ExtAPI.DataModel.MeshDataByName( "Global" )
    for refId in refIds:
        meshRegion = mesh.MeshRegionById(refId)
        nodeIds += meshRegion.NodeIds

    nodeIds = list(set(nodeIds))

    for i in range(0,nodeIds.Count):
        id = nodeIds[i]
        node = mesh.NodeById(id)
        x = node.X
        y = node.Y
        z = node.Z
        v = 0.
        try:
            v = eval(vexp)
        finally:
            values.Add(id,v)
        new_pct = (int)((i*100.)/nodeIds.Count)
        if new_pct!=pct:
            pct = new_pct
            stopped = fct(pct,"Generating data...")
            if stopped:
                return False

    propEx.SetAttributeValue( "Values",values)
    fct(100,"Generating data...")

    return True

def clearData(load):
    ExtAPI.Log.WriteMessage( "ClearData: "+load.Caption)
    propEx = load.PropertyByName("Expression")
    propEx.SetAttributeValue("Values",None)

```

The callback <**ongenerate**> takes two arguments: the load object and a function to manage a progress bar. The function also takes two arguments: the message to display and the value (between 0 and 100) of the progress bar.

During the process, the generated data is stored using an attribute on the property **Expression**. For more information, see [Storing Data in Your Extension \(p. 46\)](#).

The callback <**oncleardata**> takes one argument: the load object. This callback is invoked each time the mesh is cleaned or when the **Clean Generated Data** context menu option is selected.

Adding a Postprocessing Feature in ANSYS Mechanical

This section discusses customization for ANSYS Mechanical only as this targeted product needs to expose postprocessing features natively.

This section discusses how to add a postprocessing feature to Mechanical. The example used in the discussion is defined in the extension `DemoResult`, which creates a customized result that computes the worst value of the principal stresses for the scoped geometry entities.

The XML extension definition file `DemoResult.xml` follows. This extension adds a result to a project. As in previous examples, this XML file begins by defining the extension with the attributes `version` and `name`. The path to the IronPython script `demoresult.py` is specified by the `<script>` block. The `<interface>` block defines the toolbar and buttons. The callback function `Create_WPS_Result` is used to create and add the result to the simulation environment.

```

<extension version="1" name="DemoResult">

<guid shortid="DemoResult">1ed8a677-3f81-49eb-9f31-41364844c769</guid>

<script src="demoresult.py" />

<interface context="Mechanical">

  <images>images</images>

  <callbacks>
    <oninit>init</oninit>
  </callbacks>

  <toolbar name="Stress Results" caption="Extension: Worst Principal Stress">
    <entry name="Worst Principal Stress" icon="result">
      <callbacks>
        <onclick>Create_WPS_Result</onclick>
      </callbacks>
    </entry>
  </toolbar>

</interface>

<simdata context="Mechanical">

  <result name="Worst Principal Stress" version="1" caption="Worst Principal Stress" unit="Stress" icon="result" >

    <callbacks>
      <onstarteval>WPS_Eval</onstarteval>
      <getvalue>WPS_GetValue</getvalue>
    </callbacks>

    <property name="Geometry" caption="Geometry" control="scoping"></property>

  </result>

</simdata>

</extension>

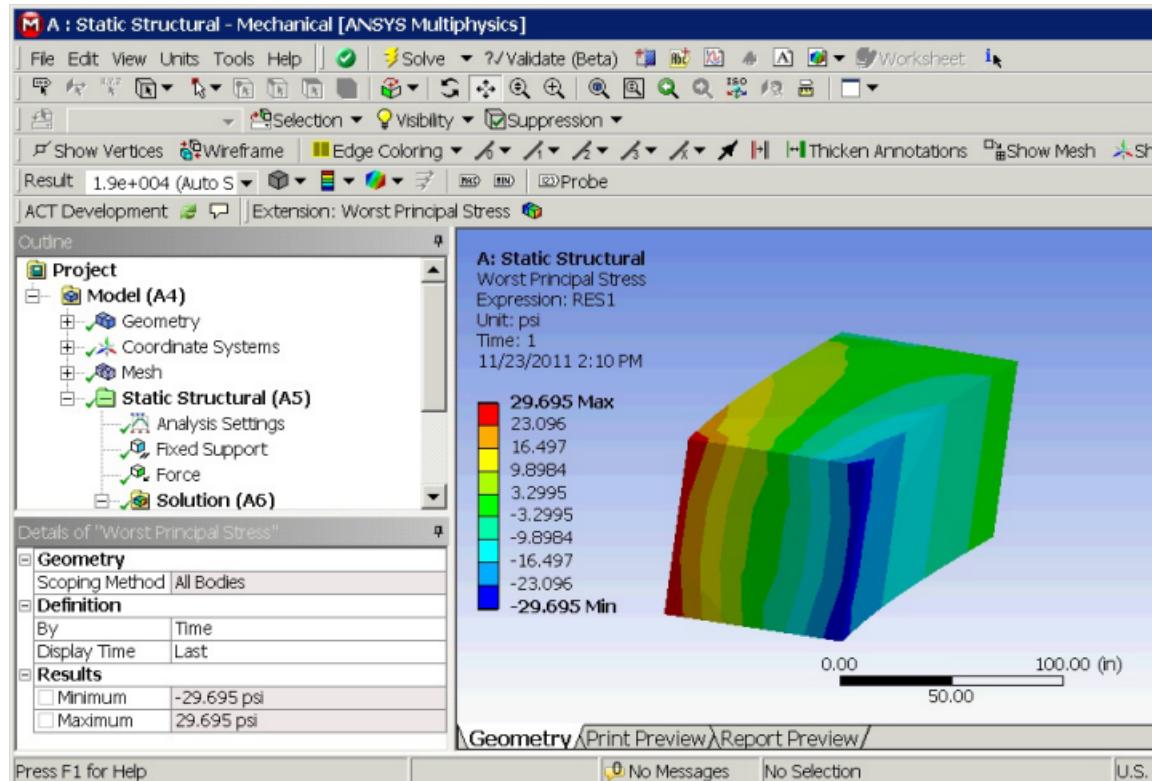
```

The definition of the result is encapsulated in the `<simdata>` block. The attributes in the child `<result>` block provide the name, version, caption, icon, and unit that apply to the result. In this child block, the `<callbacks>` block encapsulates the single callback `<onstarteval>` that is used for the evaluation.

The callback `<onstarteval>` gives the name of the function that is invoked to compute the result when the product (presently Mechanical) requests an evaluation. The output of this function is sent directly to Mechanical to display the result.

In the callback definition, you define the properties to apply to the actual result definition. These properties display in the **Details** view of Mechanical when a result is selected in the **Outline** view.

The **Details** view shows each property with the corresponding names and result values.



The IronPython script `demoresult.py` follows. The functions `Create_WPS_Result` and `WPS_Eval` are critical to the behavior and application of the result.

The function `Create_WPS_Result` creates the result and adds it to the simulation environment. This function uses IronPython dot notation, which allows you to chain objects with methods that return objects to each other. In the expression `analysis.CreateResultObject("Absolute Principal Stress")`, the `IAnalysis` object `analysis` is given in argument of the callback. The object `IAnalysis` calls the method `CreateResultObject`. From `CreateResultObject`, the XML is interpreted and the internal mechanisms are set into motion to add the details and register the callbacks that define the results framework. For more comprehensive descriptions of the API objects, methods, and properties, see the [ANSYS ACT API Reference Guide](#).

The function `WPS_Eval` is called when the result must be evaluated or re-evaluated, depending on the state of the simulation environment. The function definition requires the input arguments `result` and `step` and the output argument collector. `Result` is the result object for this result, and `stepInfo` is an `IStepInfo` object that gives information on the step currently prescribed in the details.

The function `WPS_Eval` queries for the component stresses at each node of elements. The stress tensor is used to compute the three principal stresses (eigenvalues), and then the signed maximum value over these three is stored as the final result for each node of elements. `WPS_Eval` also deals with the conversion of units. `DemoResults` uses a utility library called `units`, which is imported at the beginning of the script `demoresults.py`. With this library, `WPS_Eval` can derive a conversion factor for the stress units that is consistent with the units used during solution. The result to be returned by the evaluation function must be consistent with the international system of unit. Workbench does the

conversion to the current unit system in the product internally. Whenever the result values for **Worst Principal Stress** are needed for display purposes, Mechanical uses the output of the function **WPS_Eval**. The two callbacks and their registration to the event infrastructure of Mechanical make possible the full cycle of result definition, creation, evaluation, and display.

```

import units

wps_stress = {}
eigenvalues = {}

def init(context):
    ExtAPI.Log.WriteMessage("Init Demoresult Extension...")

def Create_WPS_Result(analysis):
    ExtAPI.Log.WriteMessage("Launch Create_WPS_Result...")
    analysis.CreateResultObject("Worst Principal Stress")

# This function evaluates the specific result (i.e. the Absolute principal stress) on each element required
# by the geometry selection
# The input data "step" represents the step on which you have to evaluate the result
def WPS_Eval(result,step):
    global wps_stress
    analysis = result.Analysis
    ExtAPI.Log.WriteMessage("Launch evaluation of the WPS result: ")
    # Reader initialization
    reader = analysis.GetResultsData()
    reader.CurrentResultSet = step
    # Get the stress result from the reader
    stress = reader.GetResult("S")

    # Define which component of the stress tensor you want to work with.
    stress.SelectComponents(["X","Y","Z","XY","XZ","YZ"])
    # Unit system management:
    # First get the unit system that was used during the resolution
    # Second compute the coefficient to be used so that the final result will be returned in the SI unit system
    unit_stress = stress.GetComponentInfo("X").Unit
    conv_stress = units.ConvertUnit(1.,unit_stress,"Pa","Stress")

    # Get the selected geometry
    propGeo = result.Properties["Geometry"]
    refIds = propGeo.Value.Ids
    # Get the mesh of the model
    mesh = analysis.MeshData
    #Loop on the list of the selected geometrical entities
    for refId in refIds:
        # Get mesh information for each geometrical entity
        meshRegion = mesh.MeshRegionById(refId)
        elementIds = meshRegion.ElementIds
        # Loop on the elements related to the current geometrical entity
        for elementId in elementIds:
            # Get the stress tensor related to the current element
            tensor = stress.GetElementValues(elementId)
            # Get element information
            element = mesh.ElementById(elementId)
            nodeIds = element.CornerNodeIds

            wps_stress[elementId] = []
            # Loop on the nodes of the current element to compute the Von-Mises stress on the element nodes
            for i in range(0,nodeIds.Count):
                local_wps = WPS(tensor[6*i:6*(i+1)])

                # Final stress result has to be returned in theSI unit system
                local_wps = local_wps * conv_stress
                wps_stress[elementId].Add(local_wps)

    # This function returns the values array. This array will be used by Mechanical to make the display available
    def WPS_GetValue(result,elementId):
        global wps_stress
        if elementId in wps_stress:
            values = wps_stress[elementId]
        else:

```

```

values = []
mesh = result.Analysis.MeshData
element = mesh.ElementById(elementId)
nodeIds = element.CornerNodeIds
for id in nodeIds:
    values.Add(System.Double.MaxValue)
return values

# This function computes the absolute principal stress from the stress tensor
# The Von-Mises stress is computed based on the three eigenvalues of the stress tensor
def WPS(tensor):

    # Computation of the eigenvalues
    eigenvalues = EigenValues(tensor)

    # Extraction of the worst principal stress
    wplocal_stress = eigenvalues[0]
    if abs(eigenvalues[1])>abs(wplocal_stress):
        wplocal_stress = eigenvalues[1]
    if abs(eigenvalues[2])>abs(wplocal_stress):
        wplocal_stress = eigenvalues[2]

    # Return the worst value of the three principal stresses S1, S2, S3
    return wplocal_stress

# This function computes the three eigenvalues of one [3*3] symmetric tensor
EPSILON = 1e-4
def EigenValues(tensor):
    global eigenvalues

    eigenvalues = []

    a = tensor[0]
    b = tensor[1]
    c = tensor[2]
    d = tensor[3]
    e = tensor[4]
    f = tensor[5]

    if ((abs(d)>EPSILON) or (abs(e)>EPSILON) or (abs(f)>EPSILON)):
        # Polynomial reduction
        A = -(a+b+c)
        B = a*b+a*c+b*c-d*d-e*e-f*f
        C = d*d*c+f*f*a+e*e*b-2*d*e*f-a*b*c

        p = B-A*A/3
        q = C-A*B/3+2*A*A*A/27
        if (q<0):
            R = -sqrt(fabs(p)/3)
        else:
            R = sqrt(fabs(p)/3)

        phi = acos(q/(2*R*R*R))

        S1=-2*R*cos(phi/3)-A/3
        S2=-2*R*cos(phi/3+2*3.1415927/3)-A/3
        S3=-2*R*cos(phi/3+4*3.1415927/3)-A/3
    else:
        S1 = a
        S2 = b
        S3 = c

    eigenvalues.Add(S1)
    eigenvalues.Add(S2)
    eigenvalues.Add(S3)

    return eigenvalues

```

To facilitate the development of IronPython functions to evaluate simulation results, the third output argument collector is internally initialized based on the content of the scoping property. When defined,

this property contains the list of FE entities on which results are evaluated. This information can be used directly in functions without looping over mesh regions. The function `WPS_Eval` demonstrates the use of this method.

This extension takes advantage of the fact that the property `collector.Ids` is initialized with the element numbers related to the selected geometrical entities. This list can be used to evaluate results for each element. The property `collector.Ids` contains nodes or element numbers depending on the type of result defined in the XML extension definition file. For results with the location set to `node`, `collector.Ids` contains a list of node numbers. For results with the location set to `elem` or `elemnode`, `collector.Ids` contains a list of element numbers.

Creating Results with Imaginary Parts

Creating results for analyses that have complex results requires managing both real and imaginary values. The method `SetValues()` is used to set values to the real part of the result. A second method, `SetImaginaryValues()`, is also available to set values to the imaginary part of the result.

An example of the creation of a complex result follows:

```
def Evaluate(result, stepInfo, collector):  
  
    for id in collectors.Ids:  
        real_value = 1.  
        # Set the real part of the result  
        collector.SetValues(id, real_value)  
  
        imaginary_value = 2.  
        # Set the imaginary part of the result  
        collector.SetImaginaryValues(id, imaginary_value)
```

Responding to a Change to the Active Unit System

When a user makes a change to the active units system in Mechanical, values in the extension might need to be converted accordingly. You can register a callback function that is invoked when such a change is made. The XML node `<onunitschanged>f_callback</onunitschanged>` provides the mechanism to register a callback function for the event `onunitschanged`. The value `f_callback` specifies the name of the callback function. A function with this name must be defined in the IronPython script for the extension.

In the following XML extension definition file, the callback `onunitschanged` specifies the function `unitschanged`.

```
<extension version="1" name="UnitsChanged">  
  <author>ANSYS, Inc.</author>  
  <guid shortid="UnitsChanged">D4C1ED2D-5104-4507-B078-5AD95B712DF1</guid>  
  <script src="rununitschanged.py" />  
  <interface context="Mechanical">  
    <images>images</images>  
    <callbacks>  
      <oninit>init</oninit>  
      <onunitschanged>unitschanged</onunitschanged>  
    </callbacks>  
  ...  
  </interface>  
</extension>
```

The IronPython script `rununitschanged.py` defines the callback function `unitschanged`.

```
clr.AddReference("Ans.UI.Toolkit.Base")  
clr.AddReference("Ans.UI.Toolkit")  
from Ansys.UI.Toolkit import *
```

```

import System

def init(context):
    ExtAPI.Log.WriteMessage("Init ACT unitschanged example...")
def unitschanged():
    ExtAPI.Log.WriteMessage("***** Units Changed *****")

```

Obsolete Callbacks "OnStartEval" and "GetValue"

Both callbacks <onstarteval> and <getvalue> have been replaced by the single callback <evaluate>. The callback <evaluate> simplifies the implementation of an ACT result, which now requires only a single IronPython function. The callbacks <onstarteval> and <getvalue> are still supported for extensions developed in previous versions of ACT, but the use of the callback <evaluate> is recommended for new extension development.

Connecting to a Third-Party Solver

This section discusses how to add a connection to a third-party solver, or the ability to launch an external process from the Mechanical system instead of launching the ANSYS solver. The example used in this section is defined in the extension ExtSolver1, which was written to demonstrate the ability to connect to a very simple solver. This solver distributes the values assigned at boundaries inside the structure. This example is for demonstration purposes only.

Third-Party Solver Connection Extension

The XML extension definition file ExtSolver1.xml follows. The extension ExtSolver1 adds a third-party solver in a project.

```

<extension version="1" name="ExtSolver1">

    <script src="[ext.Folder]\main.py" />

    <interface context="Mechanical">

        <images>[ext.Folder]\images</images>

        <callbacks>
            <onload>Load</onload>
            <onsave>Save</onsave>
        </callbacks>

        <toolbar name="ExtSolver1" caption="ExtSolver1">
            <entry name="Values Load" icon="tload">
                <callbacks>
                    <onclick>CreateValuesLoad</onclick>
                </callbacks>
            </entry>
        </toolbar>
    </interface>

    <simdata context="Project|Mechanical">

        <solver name="Solver1" version="1" caption="Solver1" icon="result" analysis="Static" physics="Structural">

            <callbacks>
                <onsolve>Solve</onsolve>
                <getsteps>GetSteps</getsteps>
                <getreader>GetReader</getreader>
            </callbacks>

            <property name="MaxIter" caption="Max. Iterations" control="integer" default="10" />
        </solver>
    </simdata>

```

```
</simdata>

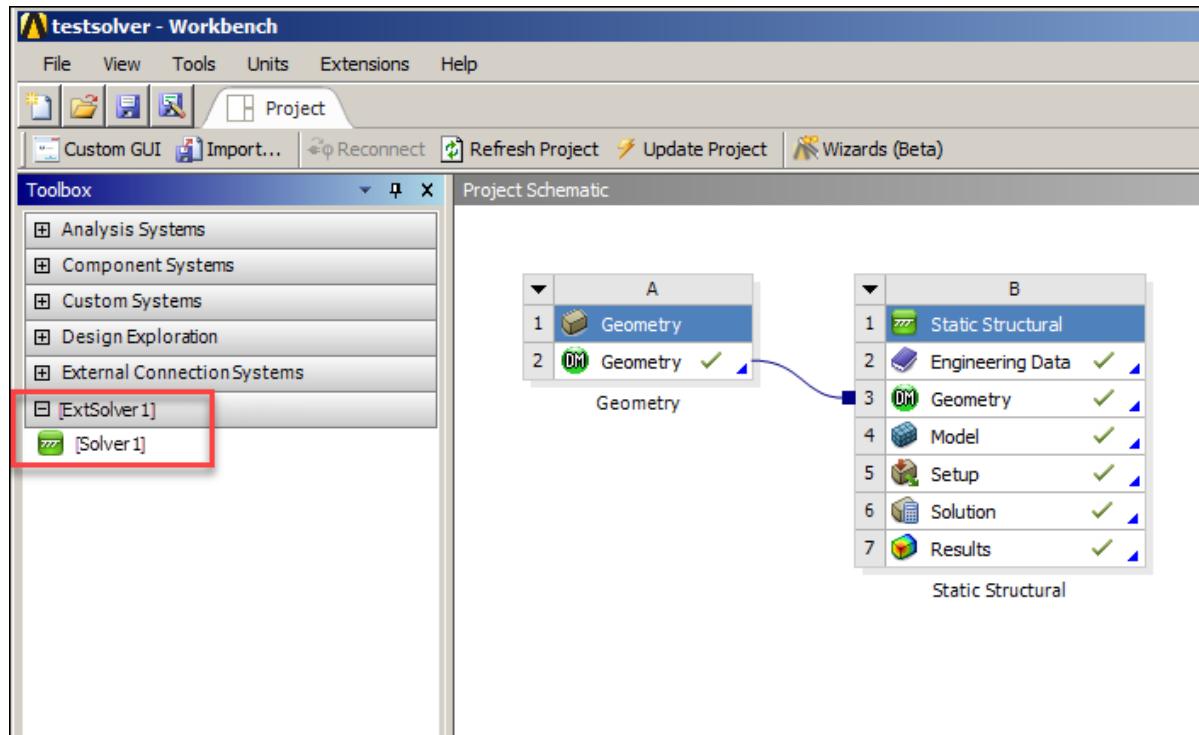
<simdata context="Mechanical">

<load name="Values" version="1" caption="Values" icon="tload" issupport="false" isload="true" color="#0000FF">
    <callbacks>
        <getsolvecommands>WriteInitialValues</getsolvecommands>
        <getnodalvaluesfordisplay>NodeValues</getnodalvaluesfordisplay>
    </callbacks>

    <property name="Geometry" control="scoping" />
    <property name="Expression" caption="Expression" control="text" />
</load>
</simdata>

</extension>
```

As with the loads and results features, the solver definition must be encapsulated within a `<simdata>` block. In this example, the attribute `context` is `Project | Mechanical`. When `Project` is specified, the solver appears as a system in the Workbench **Toolbox**.



Note

Custom solver-based systems support general data transfer to their Model, Setup, and Solution tasks. For an explanation of general data transfer, see [Defining Task-Level General Data Transfer \(p. 125\)](#).

In `ExtSolver1.xml`, the `<solver>` block specifies the definition for the third-party solver. This block has some mandatory attributes:

- `name`: Internal name of the solver

- **version**: Version of the solver
- **caption**: Display name of the solver
- **analysis**: Analysis type addressed by the solver. For compatibility reasons, this attribute must be set to **Static**, but this does not prevent you from integrating any type of third-party solver.
- **physics**: Physics type addressed by the solver. This attribute must be set to **Structural**, even though it has no real impact on what the solver computes.

You must define the callback `<onsolve>`. This callback is invoked when the product launches the solver, thereby taking the object **solver** as an argument.

You can define a set of properties, which appear in the **Details** view of the analysis. In this example, a simple property is created to define the maximum number of iterations to be performed by the solver. The default value is 10.

As shown in the previous figure, a new system for the third-party solver **ExtSolver1** is added to the **Toolbox**. Each third-party solver is added into a new category, identified by the caption of the extension. The system is named by the caption of the solver.

The system related to the third-party solver is equivalent to one standard system that can be created with the ANSYS solver. The components that build this new system remain the same. You can add the new system related to the solver to the **Project Schematic** just as you do for any other system.

The script `main.py` follows. The solver code is located in a separate script, `solver.py`, which is placed in the same folder as `main.py`. The second line of the script `main.py` is `import solver`. This technique of importing another script supports reuse and maintainability. The script in `solver.py` defines the class **SolverEngine**.

```
import os
import solver

def CreateValuesLoad(analysis):
    analysis.CreateLoadObject("Values")

initValues = {}
sol = None
solbystep = SerializableDictionary[int, dict]()
values = []
steps = []
res = [0.]
dScal = [0.]
dVec = [0., 0., 0.]

def WriteInitialValues(load, filename):
    global initValues
    propEx = load.Properties["Expression"]
    exp = propEx.Value
    if exp=="":
        return None
    vexp = compile(exp,'','eval')
    values = []
    propGeo = load.Properties["Geometry"]
    refIds = propGeo.Value.Ids
    mesh = load.Analysis.MeshData
    for refId in refIds:
        meshRegion = mesh.MeshRegionById(refId)
        nodeIds = meshRegion.NodeIds
        for nodeId in nodeIds:
            node = mesh.NodeById(nodeId)
            x = node.X
            y = node.Y
```

Feature Creation Capabilities

```
z = node.Z
v = 0.
try:
    v = eval(vexp)
    v = float(v)
finally:
    initValues.Add(nodeId,v)

def NodeValues(load,nodeIds):
    propEx = load.Properties["Expression"]
    exp = propEx.Value
    if exp=="":
        return None
    try:
        vexp = compile(exp,'','eval')
    except:
        return None
    values = []
    mesh = load.Analysis.MeshData
    for id in nodeIds:
        node = mesh.NodeById(id)
        x = node.X
        y = node.Y
        z = node.Z
        v = 0.
        try:
            v = eval(vexp)
            v = float(v)
        finally:
            values.Add(v)
    return values

def Solve(s):
    global steps
    global initValues
    global sol
    global solbystep
    global values

    solbystep = SerializableDictionary[int,dict]()
    solbystepTmp = {}

    f = open("solve.out","w")
    f.write("SolverEngine version 1.0\n\n\n")
    try:

        maxIter = int(s.Properties["MaxIter"].Value)
        f.write("Max. iteration : %d\n" % (maxIter))

        mesh = s.Analysis.MeshData

        numEdges = 0
        geo = ExtAPI.DataModel.GeoData
        nodeIds = []
        for asm in geo.Assemblies:
            for part in asm.Parts:
                for body in part.Bodies:
                    for edge in body.Edges:
                        numEdges = numEdges + 1
                        ids = mesh.MeshRegionById(edge.Id).NodeIds
                        nodeIds.extend(ids)

        steps = []
        stepsTmp = []
        f.write("Num. edges : %d\n" % (numEdges))
        sol = solver.SolverEngine(mesh,initValues,nodeIds)
        initValues = sol.Run(maxIter,f,stepsTmp,solbystepTmp)
        nodeIds = mesh.NodeIds
        sol = solver.SolverEngine(mesh,initValues,nodeIds)
        values = sol.Run(maxIter,f,steps,solbystep)

        initValues = {}
```

```

except StandardError, e:
    f.write("Error:\n");
    f.write(e.message+"\n");
    f.close()
    return False

f.close()

return True

def GetSteps(solver):
    global steps
    return steps

def Save(folder):
    global solbystep
    fm = System.Runtime.Serialization.Formatters.Binary.BinaryFormatter()
    try:
        stream = System.IO.StreamWriter(os.path.join(folder,"sol.res"),False)
    except:
        return
    try:
        fm.Serialize(stream.BaseStream,solbystep)
    finally:
        stream.Close()

def Load(folder):
    global solbystep
    if folder==None:
        return
    fm = System.Runtime.Serialization.Formatters.Binary.BinaryFormatter()
    try:
        stream = System.IO.StreamReader(os.path.join(folder,"sol.res"))
    except:
        return
    try:
        solbystep = fm.Deserialize(stream.BaseStream)
    finally:
        stream.Close()

class ExtSolver1Reader(ResultReaderBase):
    def __init__(self,infos):
        self.infos = infos
        self.step = 1

    def get_CurrentStep(self):
        return self.step

    def set_CurrentStep(self,step):
        self.step = step

    def StepValues(self):
        global steps
        return steps

    def ResultNames(self):
        return [ "U", "VALUES" ]

    def GetResultLocation(self,resultName):
        return "node"

    def GetResultType(self,resultName):
        if resultName=="U":
            return "vector"
        return "scalar"

    def ComponentNames(self,resultName):
        if resultName=="U":
            return [ "X", "Y", "Z" ]
        else:
            return [ "VALUES" ]

```

```
def ComponentUnit(self,resultName,componentName):
    if resultName=="U":
        return "Length"
    return "Temperature"

def GetValues(self,resultName,entityId):
    global values
    global solbystep
    global dVec
    global dScal
    if resultName=="U":
        values = solbystep[self.step]
        dVec[0] = 0.
        dVec[1] = 0.
        dVec[2] = 0.
        try:
            dVec[0] = values[entityId]
        finally:
            return dVec
    else:
        values = solbystep[self.step]
        try:
            dScal[0] = values[entityId]
        return dScal
    except:
        return None

def GetReader(solver):
    return [ "ExtSolver1Reader" ]
```

The function **Solve** is associated with the callback **<onsolve>**. This function creates a file `solve.out`, which is read interactively by the product and stored in the solution information.

By default, the product launches the resolution directly into the working directory, so it is not necessary to set the folder in which the file `solve.out` must be created.

The callback function must return **True** or **False** to specify if the solve has succeeded or failed.

Currently, it is not possible to return progress information.

Post Processing

You can create a result reader to expose results.

To do that, you create a class that implements the interface **ICustomResultReader**. The following methods need to be implemented. For each method, the expected results that must be returned are described.

GetCurrentStep(self)

This method must return the current step number.

SetCurrentStep(self,stepInfo)

This method is called each time the current step number is changed.

GetStepValues(self)

This method must return a list of double values that represents the time steps or frequencies.

GetResultNames(self)

This method must return a list of strings that represents the result names available for the reader.

GetResultLocation(self, resultName)

This method must return the location type of the result identified by the name `resultName`. The possible values are `node`, `element`, and `elemnode`.

GetResultType(self, resultName)

This method must return the type of the result identified by the name `resultName`. The possible values are `scalar`, `vector`, and `tensor`.

GetComponentNames(self, resultName)

This method must return a list of strings that represents the list of available components available for the result identified by the name `resultName`.

GetComponentUnit(self, resultName, componentName)

This method must return the unit name related to the result's component identified by the result name `resultName` and the component name `componentName`.

GetValues(self, resultName, collector)

This method must return a list of double values for each component associated with the result identified by the name `resultName`.

To specify a dedicated reader, add the callback `<getreader>` in the solver definition. This callback returns a list of strings, where the first is the name of the reader class and the remainder represents parameters given to the constructor of the class when the reader is instanced by ACT. Any result exposed in the method `ResultNames()` is available in Mechanical's **Results** worksheet.

If the name of the result matches a standard result name (like "U"), Workbench applies the same treatment for this result as it does for a standard one. So, if the result is named "U", Workbench uses this result to draw the deformed model.

In the extension `Extsolver1`, the reader declares one single scalar nodal result "VALUES". No deformation is considered.

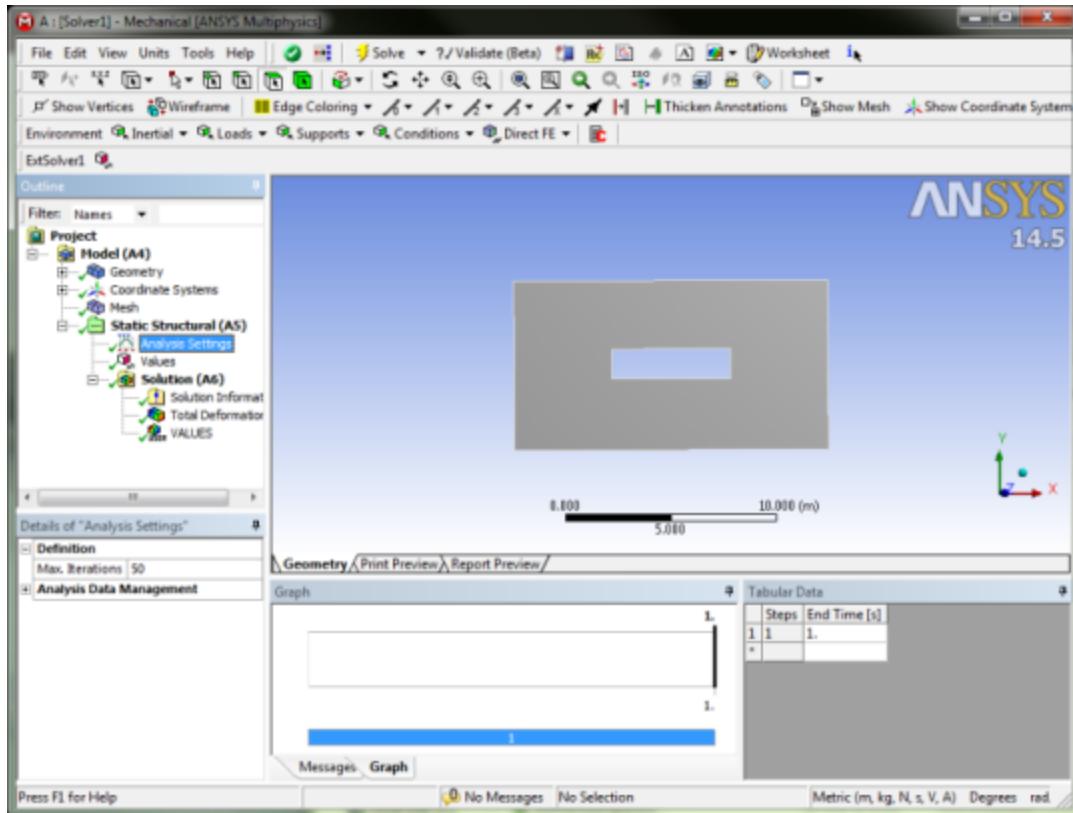
Load and Save Data

It is possible to use the callbacks `<onload>` and `<onsave>` in the `<interface>` block to load and save data associated with the third-party solver. As previously discussed, the extension `Extsolver1` uses these callbacks to save and load the computed results. This means that the results are still available if the user closes and reopens the system **Solver1**.

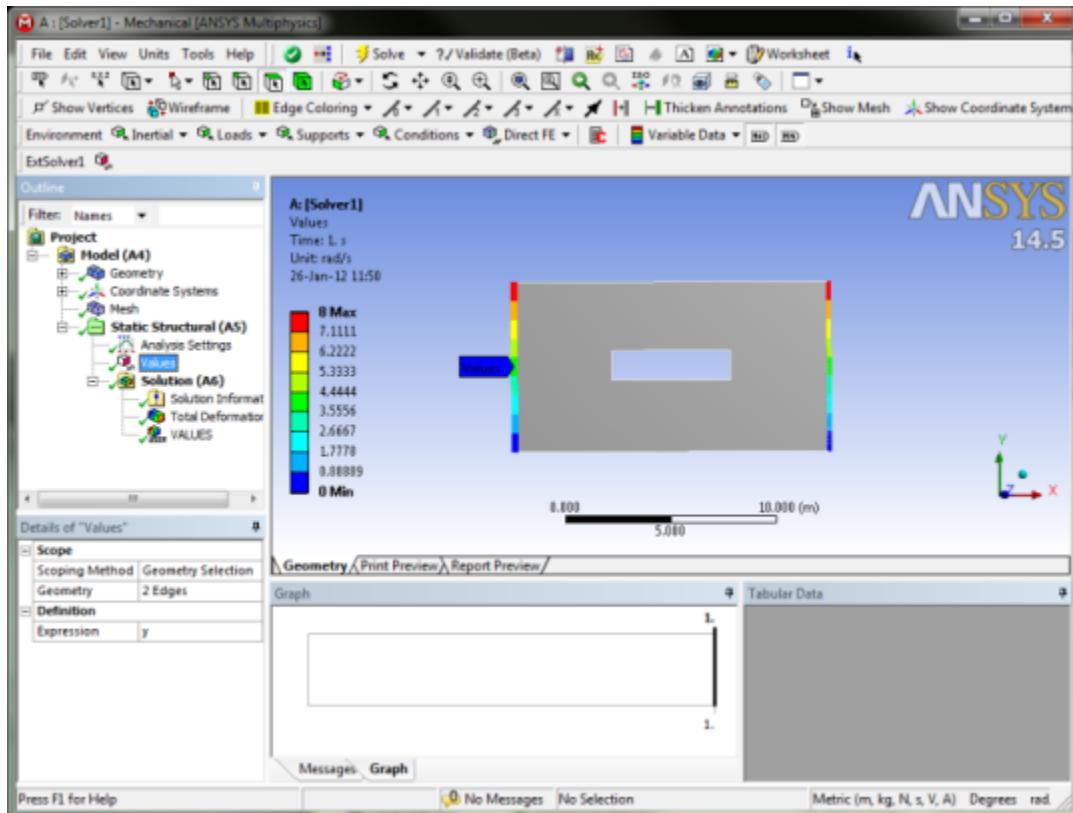
The callbacks `<onload>` and `<onsave>` take in argument the name of the working directory.

The following figure shows the analysis settings object associated with the external solver.

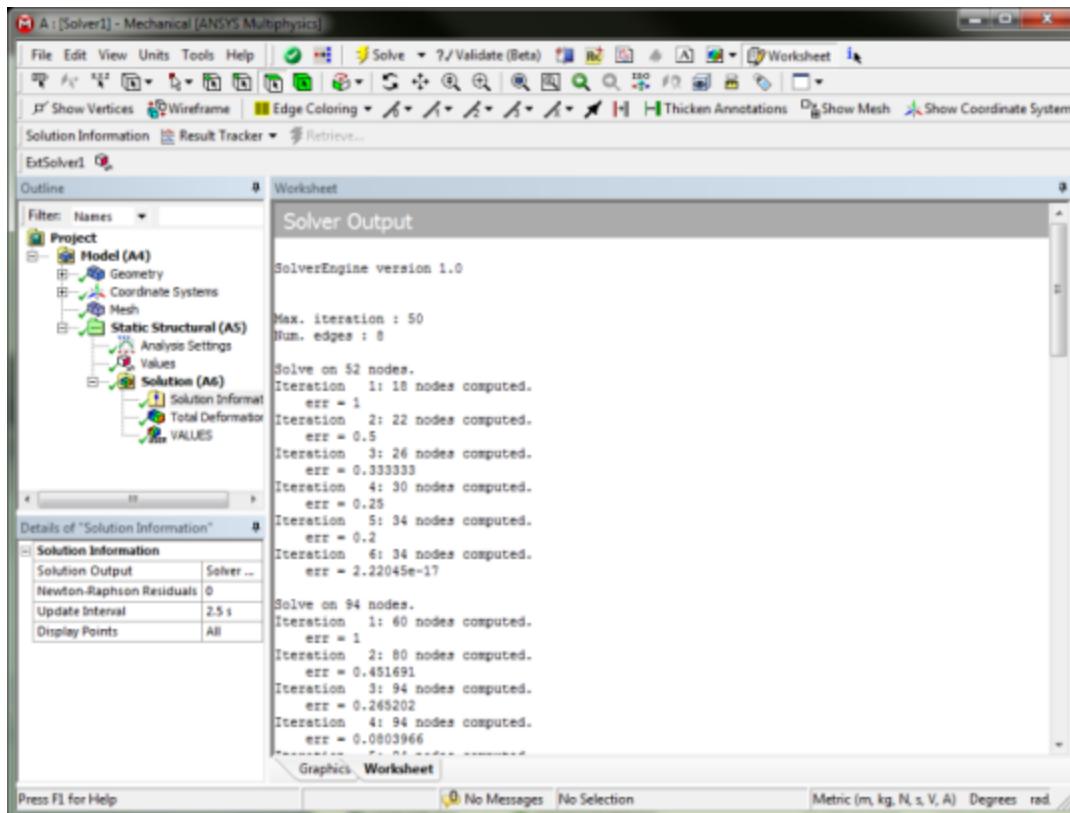
Feature Creation Capabilities



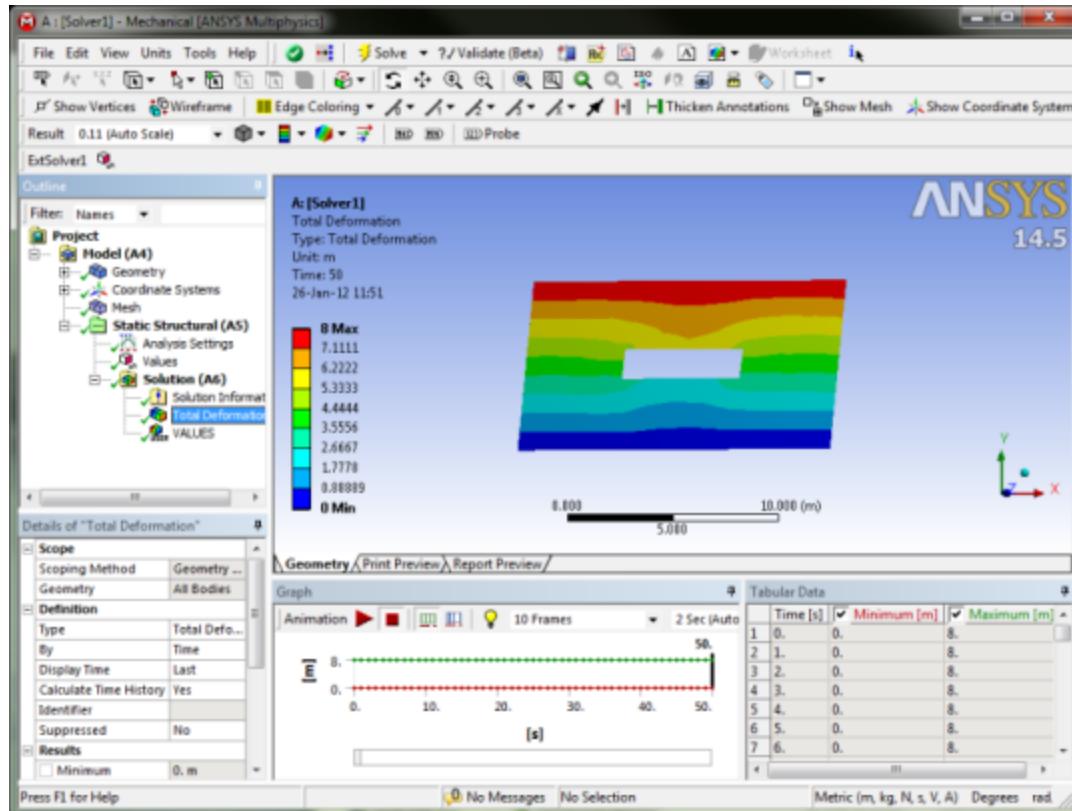
The next figure shows the boundary condition object associated with the external solver.



During the resolution, solution information is interactively displayed. Two types of output are derived from the resolution. The following figure shows the solution information associated with the external solver.



The next figure shows the postprocessing associated with the external solver.



Capabilities for ANSYS DesignModeler

This section discusses customization of ANSYS DesignModeler. While ACT is used to work with loads and results in ANSYS Mechanical, ACT is used to work with geometries in DesignModeler. Once you have used ACT to add custom toolbars and menus to the DesignModeler user interface, you can use those entities to generate geometries by creating and using a primitive generator, defining properties, and applying various operations to the resulting geometry.

The extension `GeometryFeature` described in this section is not included in the package **ACT Developer's Guide Examples**. This theoretical example for adding a geometry to DesignModeler is for demonstration purposes only.

The extension `GeometryFeature` creates a geometry based on an existing part. As part of this process, the extension draws a circle, defines a disc based on the circle, and extrudes the disc into a cylinder.

The following topics are discussed:

[Geometry Definition in the XML Extension Definition File](#)

[Geometry Definition in the IronPython Script](#)

Geometry Definition in the XML Extension Definition File

The XML extension definition file `GeometryFeature.xml` follows.

```
<extension version="1" name="GeometryFeature">
<script src="main.py" />
<interface context="DesignModeler">
  <images>images</images>
  <toolbar name="ACTtoolbar" caption="ACTtoolbar">
    <entry name="MyFeature" icon="construction">
      <callbacks>
        <onclick>createMyFeature</onclick>
```

```

</callbacks>
</entry>
</toolbar>
</interface>
<simdata context="DesignModeler">
  <geometry name="MyFeature" caption="MyFeature" icon="construction" version="1">
    <callbacks>
      <ongenerate>generateMyFeature</ongenerate>
      <onaftergenerate>afterGenerateMyFeature</onaftergenerate>
    </callbacks>
    <property name="Face" caption="Face" control="scoping">
      <attributes selection_filter="face"/>
    </property>
    <property name="Length" caption="Length" control="float" unit="Length" default="1.2 [m]"></property>
    <property name="Minimum Volume" caption="Minimum Volume" control="float" unit="Volume" >
    <attributes readonlyaftergenerate="true" />
    </property>
  </geometry>
</simdata>
</extension>

```

As in previous examples, this file first defines the extension using the attributes **version** and **name**. The path to the IronPython script `main.py` is specified by the **<script>** block. The ACT toolbar is defined in the **<interface>** block, which has a context of DesignModeler. The callback function **createMyFeature** is used to create and add the geometry to the DesignModeler environment.

The definition of the geometry is encapsulated in the **<simdata>** block. The attributes in the child **<geometry>** block provide the name, caption, icon, and version that apply to the geometry. The geometry **<callbacks>** block includes two callbacks, **<ongenerate>** and **<onaftergenerate>**.

- The callback **<ongenerate>** specifies the name of the function to invoke when the geometry is generated by the product. The product provides the geometry type (type **IDesignModelerGeoFeature**) as the callback argument.

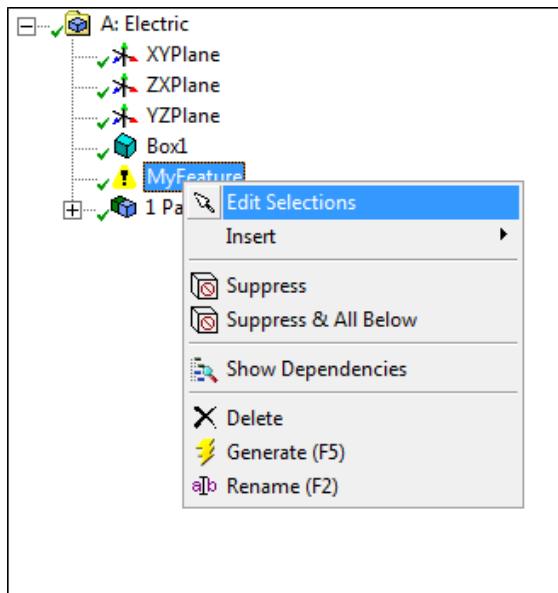
This callback must return a Boolean value to indicate whether the generation has been successful. It returns **true** if the generation has succeeded and **false** if it has failed.

- The callback **<onaftergenerate>** specifies the name of the function to invoke after the geometry is generated by the product. The product provides the geometry type (type **IDesignModelerGeoFeature**) as the callback argument.

To specify additional details about the geometry, you can include additional callbacks, such as **<canadd>**, **<onadd>**, **<oninit>**, and **<onmigrate>**, in the **<callbacks>** block.

In the callback definition, you can define the properties to apply to the actual definition of the geometry. In this example, **Face** and **Length** properties are applied. These properties display in the **Details View** pane of DesignModeler, where the user provides the necessary values to complete the geometry definition. A subsequent section reviews the IronPython script to show how geometry properties can be retrieved and modified.

The following figure shows that each defined property appears in the **Details View** pane with the corresponding names and types of interface control.



The two geometry properties, **Face** and **Length**, use different control types.

- The property **Face** uses a **scoping** control type with the attribute **selection_filter** set to **face**. This specifies that the value for this property is one or more faces. The faces are defined in the file `main.py` and then used to generate the geometry.
- The property **Length** uses a **float** control type and does not require the definition of one specific callback. Instead, the XML file introduces a physical quantity dependency with the attribute **unit**, which specifies that this property is consistent with a length. The attribute **default** specifies a default value of `1.2[m]`. This default value is exposed in the **Details View** pane each time a new load is created.

Geometry Definition in the IronPython Script

The IronPython script for this example is named `main.py`. This section discusses the functions associated with the callbacks `<ongenerate>` and `<onaftergenerate>`.

```
import units
import math

def createMyFeature(feature):
    ExtAPI.CreateFeature("MyFeature")

def vectorProduct(v1, v2):
    return [v1[1]*v2[2]-v1[2]*v2[1], -v1[0]*v2[2]+v1[2]*v2[0], v1[0]*v2[1]-v1[1]*v2[0]]

def scalarProduct(v1, v2):
    return v1[0]*v2[0]+v1[1]*v2[1]+v1[2]*v2[2]

def norm(v):
    return math.sqrt(scalarProduct(v,v))

def generateMyFeature(feature,function):

    length = feature.Properties["Length"].Value
    length = units.ConvertUnit(length, ExtAPI.DataModel.CurrentUnitFromQuantityName("Length"), "m")
    faces = feature.Properties["Face"].Value.Entities
    bodies = []
    builder = ExtAPI.DataModel.GeometryBuilder

    for face in faces:
        centroid = face.Centroid
        uv = face.ParamAtPoint(centroid)
```

```

normal = face.NormalAtParam(uv[0], uv[1])
radius = math.sqrt(face.Area/(math.pi*2))

xdir = [1., 0., 0.]
vector = vectorProduct(xdir, normal)
if norm(vector)<1.e-12:
    xdir = [0., 1., 1.]
s = scalarProduct(xdir, normal)
xdir = [xdir[0]-s*normal[0],xdir[1]-s*normal[1],xdir[2]-s*normal[2]]
n = norm(xdir)
xdir[0] /= n
xdir[1] /= n
xdir[2] /= n

arc_generator = builder.Primitives.Wire.CreateArc(radius, centroid, xdir, normal)
arc_generated = arc_generator.Generate()

disc_generator = builder.Operations.Tools.WireToSheetBody(arc_generated)

normal[0] *= -1
normal[1] *= -1
normal[2] *= -1
extrude = builder.Operations.CreateExtrudeOperation(normal,length)
cylinder_generator = extrude.ApplyTo(disc_generator)[0]

bodies.Add(cylinder_generator)

feature.Bodies = bodies
feature.MaterialType = MaterialTypeEnum.Add
return True

def afterGenerateMyFeature(feature):
    edges = []
    minimum_volume = System.Double.MaxValue
    for body in feature.Bodies:
        body_volume = 0
        if str(body.BodyType) == "GeoBodySolid":
            body_volume = body.Volume
            if body_volume <= minimum_volume:
                minimum_volume = body_volume

        for edge in body.Edges:
            edges.Add(edge)
        else:
            ExtAPI.Log.WriteMessage("Part: "+body.Name)

    feature.Properties["Minimum Volume"].Value = minimum_volume

ExtAPI.SelectionManager.NewSelection(edges)
named_selection = ExtAPI.DataModel.FeatureManager.CreateNamedSelection()
ExtAPI.SelectionManager.ClearSelection()

```

Functions Associated with the Callback <ongenerate>

Descriptions follow of the functions associated with the callback **<ongenerate>**.

- The function **createMyFeature(feature)** creates the geometry in the DesignModeler tree. It is displayed as **My Feature** in the **Tree Outline** pane.
- The mathematical functions **vectorProduct** and **scalarProduct** are used later in the function **generateMyFeature**.

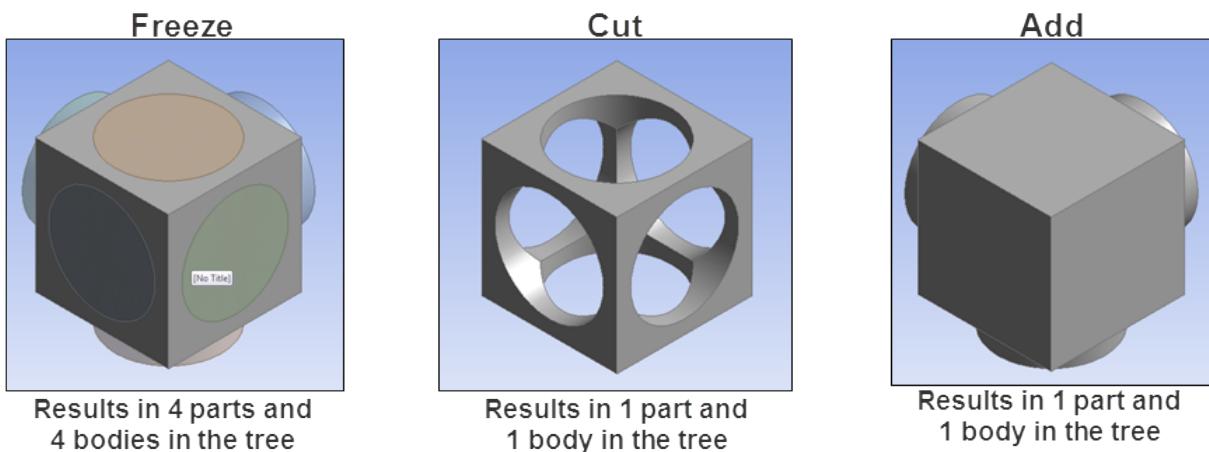
The first several functions configure details that are used later in the creation of the geometry.

In this example, the function **generateMyFeature** is referenced in **GeometryFeature.xml** by the callback **<ongenerate>**. When called, it generates the geometry feature. Within this function, the following details about the geometry are defined:

- The property **length**, which is used later to create the primitive. The conversion of the length unit ensures that you are working with the expected metric unit.
- The scoped property **face**, which in this example is three faces.
- The list **bodies**, where the geometric features to create are added.
- The **builder**, which serves as the ACT gateway in DesignModeler. All features and operations are accessed from here.
- The variable **faces**, which is used as the argument to create the circle. Under faces:
 - Define the **centroid** of the scoped face.
 - Evaluate the **normal** to this face.
 - Evaluate the **radius** that is to be used for the arc wire.
- Calculate the **xdir**, which is the principle direction of the arc wire defined later and is required to draw the arc.
- The objects **vectorProduct** and **scalarProduct** are used to specify the location of the geometry's primitives and operations.
- The next several objects generate an arc wire.
 - With the object **arc_generator**, the extension uses the primitive generator **builder.Primitives.Wire.CreateArc**. The method **CreateArc()** uses arguments from faces to draw the circle. This generator can be used one or more times to build the primitive body.
 - With the object **arc_generated**, the extension uses the method **Generate()** to generate the primitive body.
 - With the object **disc_generator**, the extension uses the operations generator **builder.Operations.Tools.WireToSheetBody** to define a disc based on the circle.
- The next several lines extrude the disc into a cylinder.
 - With the object **extrude**, the extension uses the operations generator **builder.Operations.CreateExtrudeOperation** to specify that the resulting extrusion is equal to the value of the property **Length**.
 - With the object **cylinder_generator**, the extension uses the method **ApplyTo** to define the geometry to which the extrude operation is applied. This method returns a list of bodies to which the operation is applied.
- Bodies added to the list **feature.Bodies** are added to DesignModeler after the generation. All other bodies used in the generation process are lost.

The list **feature.Bodies** can contain both bodies and parts. You can create a part with the command **builder.Operations.Tools.CreatePart(list_of_bodies)**.

- The list **material.Type** allows you to enter different properties such as **Freeze**, **Cut**, and **Add**. The following figure illustrates the resulting geometry given selection of the different properties. You can see the effect of different material type properties on the geometry.



Functions Associated with the Callback <onaftergenerate>

A description follows of the IronPython function associated with the callback `<onaftergenerate>`. This callback specifies the name of the function that is invoked when the callback `<ongenerate>` is complete.

- The callback `<onaftergenerate>` allows you to access and work with the part or bodies created previously by the callback `<ongenerate>`.

For example, you can use it rename a part or body, suppress it, create a named selection, create a new selection via the Selection Manager, or access one or more of the geometric features of a body as its area.

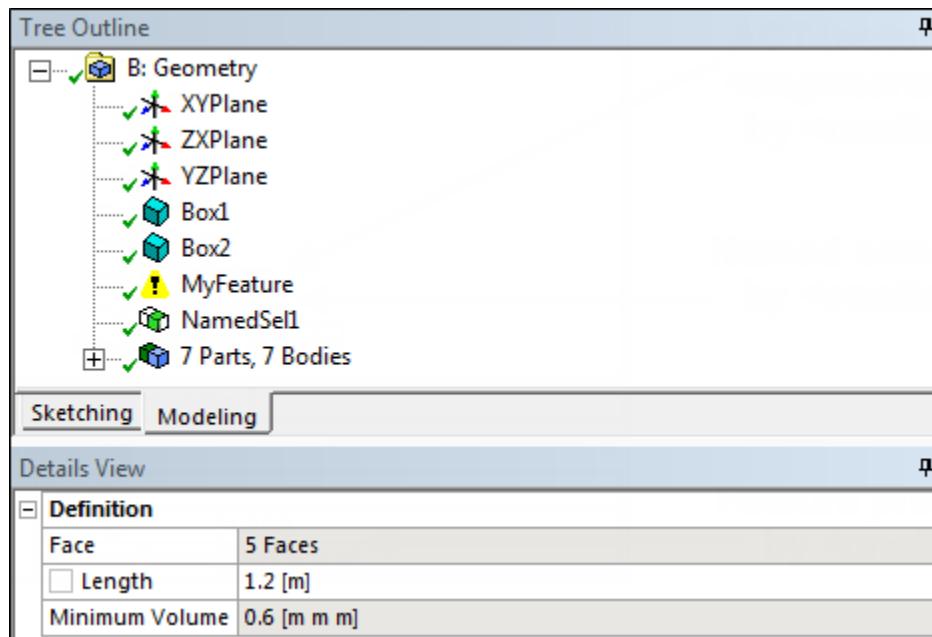
- Unlike the callback `<ongenerate>`, which expects a return value of `True` or `False`, the callback `<onaftergenerate>` does not expect a return value.
- The callback `<onaftergenerate>` allows you to define properties for the feature to be created.

In this example, the function `afterGenerateMyFeature` is referenced in `GeometryFeature.xml` by the callback `<ongenerate>`. It accepts only the feature itself as an argument. When called, it specifies the properties and attributes of the feature and selects edges of the bodies created:

- The list `edges`, where you add all the edges of the bodies to create for the feature.
- The property `Minimum Volume`, which in this example is the volume of the body with the smallest volume of all the bodies created for the feature. It uses a `float` control type and uses the attribute `unit` to specify a value consistent with a volume. Because this property has the special attribute `readonlyaftergenerate` set to `true`, the value becomes non-editable after generation of the feature.
- The variable `edge`, which is used to find the edges of all the bodies created for the feature.
- The entry `feature.Properties["Minimum Volume"].Value` sets the property `Minimum Volume` in DesignModeler to the value `minimum_volume`.
- The command `ExtAPI.SelectionManager.NewSelection(edges)` creates a new selection, which includes all edges previously selected.

- The command `named_selection = ExtAPI.DataModel.FeatureManager.CreateNamedSelection()` creates a named selection that is defined by the current selection containing all edges previously selected.

The following figures shows the feature, named selection, and properties in DesignModeler.



Capabilities for ANSYS DesignXplorer

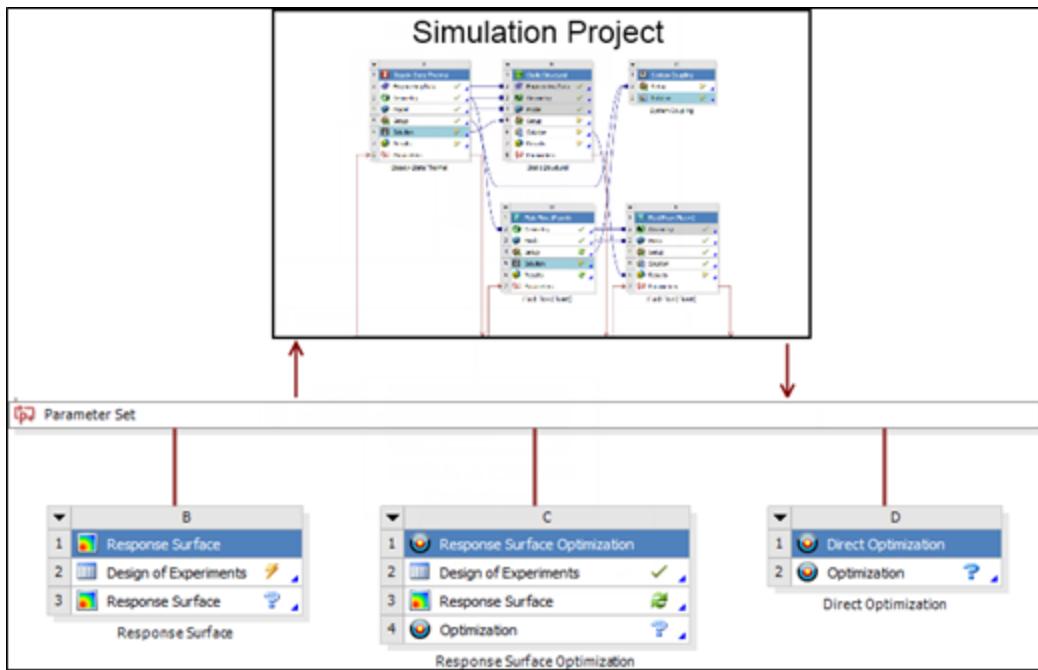
This section discusses how to define functionality for ANSYS DesignXplorer. This ANSYS Workbench product offers design exploration and optimization features.

You can create an ACT extension to integrate external functionality with DesignXplorer. A **DOE Extension** integrates one or more external sampling methods, while an **Optimization Extension** integrates one or more external optimization methods.

From a user-interface point of view, the sampling or optimization method, along with specific properties to control its behavior, is hosted and exposed in DesignXplorer as an additional method. Through its API, DesignXplorer provides the services required the calculation of points, report progress, and return results. The end-user experience is the same for both built-in and external methods of optimization or sampling.

From a development point of view, a sampling or optimizer is defined as a standard ACT extension that includes one or more specific elements for the DesignXplorer context. A required callback, which allows for the instantiation of the external method, is a class generally coded in C#, C/C++ or Fortran by the extension developer. The callback class must implement the method interface (**ISamplingMethod** or **IOptimizationMethod**) for the sampling or optimizer to be recognized as valid and interact with DesignXplorer.

DesignXplorer takes advantage of the ANSYS Workbench platform to allow parametric studies in a generic way for any simulation project. Regardless of the number of solvers and physics or the complexity of the workflow involved to execute the simulation, DesignXplorer sees the same simplified representation of the project and interacts with it only in terms of parameters, design points, and the associated services.



This generic architecture makes it possible to implement a variety of algorithms without needing to worry about the communication with solvers. Because DesignXplorer hosts the external methods, they also take advantage of the ANSYS Workbench platform architecture to work with any kind of simulation project defined in ANSYS Workbench, with no additional implementation effort.

The following topics are discussed:

[The Design Exploration Process](#)

[Implementing a DesignXplorer Extension](#)

[Notes on Method Class Implementation](#)

The Design Exploration Process

The external functionality defined by the extension is responsible for carrying out relevant analysis tasks as defined by the DesignXplorer user. The external optimizer is responsible for solving the optimization study, and the external sampling is responsible for generating the DOE. The external functionality is not required to support all the features available in DesignXplorer. However, because the sampling or optimizer is hosted in the DesignXplorer environment, it is subject to the environment's limitations. Consequently, some of the external features might not be available in DesignXplorer.

The DOE and optimization study are defined as follows:

Design of Experiments	Optimization Study
Variables	Variables
DOE domain	Optimization domain
	Parameter relationships
	Objectives
	Constraints

Variables are the parameters of the Workbench project. As there are input and output parameters, there are input and output variables for the sampling or optimization. Because the user can disable input

parameters, the number of input variables transferred to the external functionality is smaller than or equal to the number of input parameters defined in the Workbench project.

The **Domain** is the multidimensional space that the sampling or optimizer can explore to generate the DOE or solve the optimization problem. It is defined by the input variables and their range of variation or as a list of possible values. There are different types of input variables:

- **Double Variable:**

- Continuous variable defined by a range of variation, from its lower to its upper bound, where the bounds are real values
- Exposed to the user as a continuous input parameter

- **Double List Variable:**

- Defined by a list of sorted real values
- Exposed to the user as a continuous input parameter with manufacturable values

- **Integer List Variable:**

- Defined by a list of integer values
- Exposed to the user as a discrete input parameter

The optimization **Parameter Relationships** define relationships between input variables such as **P1+2*P2 <= 12.5[mm]**. They allow the user to restrict the definition of the optimization domain.

Optimization **Objectives** are defined on variables, such as **Minimize P3**. The following objective types are supported:

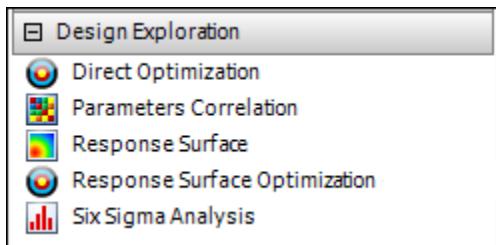
- **Minimize** a variable
- **Maximize** a variable
- **Seek Target** for a variable, where the target is a user-specified constant value

The optimization **Constraints** are defined on variables as well, such as **P3 <= 80**. Both an objective and a constraint can be defined for the same variable. The following constraint types are supported:

- **Less Than**, where the user defines the lower bound (constant value)
- **Greater Than**, where the user defines the upper bound (constant value)
- **Equal To**, where the user defines the target value (constant value)
- **Inside Bounds**, where the user defines the targeted interval (constant values)

DesignXplorer Systems and Component

DesignXplorer exposes several types of systems in Workbench, with each system corresponding to a type of parametric analysis. DesignXplorer systems can be found under **Design Exploration** in the **Toolbox**.



Design of Experiments Component

The **Design of Experiments** component appears in two different kinds of design exploration systems: **Response Surface** and **Response Surface Optimization**. It provides a selection of built-in sampling methods. Each sampling method comes with its own specific standard and advanced properties, capabilities, and limitations.

Optimization Components

The **Optimization** component appears in two different kinds of design exploration systems: **Direct Optimization** and **Response Surface Optimization**. It provides a selection of built-in optimization methods. Each optimization method comes with its own specific standard and advanced properties, capabilities, and limitations.

- **Direct Optimization** system:

- The system is composed only of an **Optimization** component.
- Each point requested by the optimizer is calculated by updating a design point (real solve).
- With this approach:
 - The optimizer manipulates accurate output parameter values (directly returned by the solver).
 - The number of input parameters is not limited. However, the cost is proportional to the number of points.

- **Response Surface Optimization** system:

- The system is composed of three components (**Design of Experiments**, **Response Surface**, and **Optimization**).
- Each point requested by the optimizer is calculated by evaluating the response surface.
- With this approach:
 - The optimizer manipulates approximated output parameter values.
 - The number of input parameters is limited by the ability to generate the response surface. However, the cost is extremely low because the resources needed to evaluate the response surface are negligible. The optimization is performed as a postprocessing operation based on the response surface, so several alternative optimization studies can be performed very quickly on the same surface.

DesignXplorer Extensions

External DOEs and optimizers can be packaged and installed as Workbench extensions. When the extension is installed and loaded, DesignXplorer detects the external DOEs or optimizers and exposes them as additional custom methods. If the user selects one of these custom methods, DesignXplorer delegates to the selected external option. It delegates sampling resolution to the external sampling and optimization resolution to the external optimizer. When an external option is used, the Workbench session is journaled and can be replayed as usual.

The extension is a container for one or more DOEs or optimizers. The DOEs and optimizers themselves are generally provided as compiled libraries included in a WBEX file for the extension.

Implementing a DesignXplorer Extension

A DesignXplorer extension has three main parts:

- XML extension definition file that declares, defines, and configures the extension
- IronPython scripts that implement callbacks
- **ISamplingMethod** or **IOptimizationMethod** implementation, which is the adapter to your existing implementation. This is typically achieved in C# or IronPython.

This section describes each of these pieces and includes references to several examples of functional DesignXplorer extensions.

Implementation Requirements

The implementation of a DesignXplorer extension requires:

- **ANSYS Workbench** installation
- **ANSYS Customization Suite** license
- **IronPython**, which is provided as part of ANSYS Workbench in the directory `<installdir>\common-files\IronPython`
- **Microsoft Visual Studio 2010 SP1** for C# implementation
- **Microsoft Visual Studio 2010 SP1** or equivalent for C/C++ implementation

DesignXplorer Extension Definition and Configuration

As with any other type of extension, a DesignXplorer extension is defined and configured by an XML extension definition file that identifies the extension (version, name, author, description) and provides the following main entry points:

- Filename of the main IronPython script (as specified by the **<script>** block)
- Name of the function to invoke when initializing the extension (as specified by the callback **<oninit>**)

In a DesignXplorer extension, the callback **<oninit>** is the correct place to load the DLL containing the external sampling. Another callback named **<onterminate>** is invoked when the extension is unloaded and is the correct place to perform cleanup operations and unload the DLL.

A DesignXplorer extension specifically defines external sampling or optimization methods that DesignXplorer can use. The definitions are encapsulated in `<simdata>` blocks, where the attribute `context` is set to `DesignXplorer`.

The following figure shows the main blocks that you use to declare and define DOEs.

```

<extension version="1" name="PythonSampling"
    |   description="Full Factorial DOE supporting 10 continuous input parameters">
<guid shortid="PythonSampling">F4F4A543-F84E-4791-BD0D-754A7395AA95</guid>
<script src="[ext.Folder]\main.py" />
<interface context="DesignXplorer">
    <callbacks>
        |   <oninit>onInit</oninit>
    </callbacks>
</interface>

<simdata context="DesignXplorer">
    <sampling name="FullFactorial" caption="Full Factorial" icon="construction" version="1" MaximumNumberOfInputParameters="10"
        |  LogFile="true" CustomTable="false">

        <callbacks>
            <oncreate>createSampling</oncreate>
            <canrun>canRunSampling</canrun>
        </callbacks>

        <property name="NumberOfLevels" caption="Number of Levels" control="integer" default="3">
            <attributes min="1" max="100"/>
        </property>

        <propertygroup name="Group4" caption="Sampling Status" display="caption">
            <property name="NumberOfPoints" caption="Number of Points" control="integer" readonly="true"></property>
        </propertygroup>
    </sampling>

    </simdata>
</extension>

```

The DOE defined is called **FullFactorial**. The two attributes framed in red are examples of capabilities being defined for this DOE. Here you can see that for **FullFactorial**:

- The attribute `LogFile` is set to `true` so that the sampling is capable of using the DesignXplorer API to generate a log file in the Workbench project.
- The attribute `CustomTable` is set to `false` so that the DOE is not capable of handling the custom table available in DesignXplorer.
- All other capabilities are configured per the default values. For more information, see [DesignXplorer Extension Capabilities \(p. 97\)](#).

The callbacks to hook up with DesignXplorer are framed in blue. The callback `<oncreate>`, which is invoked to obtain an `ISamplingMethod` instance, is mandatory. All other callbacks are optional.

Finally, all of the properties declared for the DOE are framed in orange. These properties allow the user to control specific settings of the algorithm, access relevant output information (such as number of levels or status), or whatever else the DOE needs to expose as a result of the sampling run.

The next figure shows the main blocks used to declare and define the optimizers.

```

<extension version="1" name="MyOptimizer">

  <script src="[ext.Folder]\main.py" />

  <interface context="DesignXplorer">
    <callbacks>
      <oninit>onInit</oninit>
    </callbacks>
  </interface>

  <simdata context="DesignXplorer">

    <optimizer name="MyFirstOptimizer" caption="Random Optimizer" version="1"
      SeekObjective="false"LogFile="true" >

      <callbacks>
        <oncreate>createOptimizer</oncreate>
        <canrun>canRunOptimizer</canrun>
        <description>getDescription</description>
        <configuration>getConfiguration</configuration>
        <quickhelp>getQuickHelp</quickhelp>
        <InputParametersEdited>InputParametersEdited</InputParametersEdited>
      </callbacks>

      <property name="MyNumberOfSamples" caption="Number of Samples" control="integer" default="50"
      ></property>
      <property name="MyMaximumNumberOfCandidates" caption="Maximum Number of Candidates" control=
      "integer" default="2">
        <attributes min="1"/>
      </property>
      <property name="MySeedValue" caption="Seed Value" control="integer" default="0"></property>

    </optimizer>

    <optimizer name="MySecondOptimizer" caption="Another algorithm of mine" version="1"
    </optimizer>

  </simdata>

</extension>

```

The extension is named **MyOptimizer**. It defines two different external optimizers. The declaration for each optimizer is marked with a green bracket. DesignXplorer manages each of these optimizers as an independent optimization method.

The first optimizer defined is called **MyOptimizer**. The two attributes framed in red are examples of capabilities being defined for this optimizer. Here you can see that for **MyOptimizer**:

- The attribute **SeekObjective** is set to **false**, so the optimizer does not have the capability to handle the **Seek Target** objective available in DesignXplorer.
- The attribute **LogFile** is set to **true**, so the optimizer has the capability to use the DesignXplorer API to generate a log file in the Workbench project.
- All other capabilities are configured per the default values. For more information, see [Optimization Example \(p. 97\)](#).

The callbacks to hook up with DesignXplorer are framed in blue. The callback **OnCreate**, which is invoked to obtain an **IOptimizationMethod** instance, is mandatory. All other callbacks are optional.

Finally, all of the properties declared for the optimizer are framed in orange. These properties allow the user to control specific settings of the algorithm, access relevant output information (such as the number of iterations, final convergence metrics, or status), or whatever else the optimizer needs to expose as a result of the optimization run.

DesignXplorer Extension Capabilities

A DesignXplorer extension is designed to solve a specific range of problems, and its implementation is characterized by complementary features and limitations. To allow DesignXplorer to determine when a given sampling or optimization method is applicable, depending on the current DOE or optimizer definition, it is necessary to declare the main capabilities of each external method. There are also optional capabilities that DesignXplorer uses to adjust the user interface according to what complementary features are supported by the external method.

For external DOEs, the capabilities are specified in the XML extension definition file as attributes of the **<sampling>** block. For external optimizers, the capabilities are specified in the XML extension definition file as attributes of the **<optimizer>** block.

Main Capabilities

The main capabilities are used to determine if the external sampling or optimization method can be applied to the problem as it is currently defined. If the external method has the required capabilities, it is available as a menu option in the DesignXplorer **Properties** view (applicable external samplings are available in the **Design of Experiments Type** menu, and applicable optimizers are available in the **Method Name** menu). If an external method is not applicable, it is not listed as a menu option. Each modification to the DOE or optimization study (such as modifying an existing input parameter or enabling or disabling an input parameter) triggers an evaluation of the relevant capabilities to reassess whether the external method is applicable and is to be made available to the user.

DOE Example

In the definition of the external DOE, the external sampling **FullFactorial** has **MaximumNumberOfInputParameters** set to 10. As a consequence, as soon as the user defines more than 10 input parameters, **FullFactorial** is removed from the list of available sampling methods. If the user then changes the DOE so that the number of input parameters is less than or equal to 10, **FullFactorial** is immediately restored as an available option.

If the user selects **FullFactorial** first and then defines more than 10 input parameters afterward, the sampling method is retained. Given the incompatibility of the sampling method and the number of input parameters, however, the state of the DOE changes to **Edit Required**. A Quick Help message is provided to explain why the sampling cannot be launched as currently configured.

Optimization Example

In the definition of the external optimizer, **MyFirstOptimizer** does not have the **SeekObjective** capability. As a consequence, as soon as the user defines a **Seek Target** objective, **MyFirstOptimizer** is removed from the list of available optimization methods. If the user then changes the optimization study so that the **Seek Target** objective is no longer defined, **MyFirstOptimizer** is immediately restored as an available option.

If the user selects **MyFirstOptimizer** first and then defines the unsupported **Seek Target** objective afterward, the optimization method is retained. Given the incompatibility of the optimization method and the objective, however, the state of the optimization changes to **Edit Required**. A

Quick Help message is provided to explain why the optimization cannot be launched as currently configured.

The following table lists the main capabilities controlling the availability of external methods.

DOE	Optimizer
<code>MaximumNumberOfInputParameters</code>	<code>MaximumNumberOfInputParameters</code>
<code>MaximumNumberOfDoubleParameters</code>	<code>MaximumNumberOfDoubleParameters</code>
<code>MaximumNumberOfDoubleListParameters</code>	<code>MaximumNumberOfDoubleListParameters</code>
<code>MaximumNumberOfIntegerListParameters</code>	<code>MaximumNumberOfIntegerListParameters</code>
	<code>ParameterRelationship</code>
	<code>ObjectiveOnInputParameter</code>
	<code>ConstraintOnInputParameter</code>
	<code>MinimizeObjective</code>
	<code>MaximizeObjective</code>
	<code>SeekObjective</code>
	<code>LessThanConstraint</code>
	<code>GreaterThanConstraint</code>
	<code>EqualToConstraint</code>
	<code>InsideBoundsConstraint</code>
	<code>MinimumNumberOfObjectives</code>
	<code>MaximumNumberOfObjectives</code>
	<code>MinimumNumberOfConstraints</code>
	<code>MaximumNumberOfConstraints</code>
	<code>BasedOnResponseSurfaceOnly</code>
	<code>BasedOnDirectOptimizationOnly</code>

For a comprehensive list of capabilities, see the "Sampling" and "Optimizer" sections in the [ANSYS ACT API Reference Guide](#) and [ANSYS ACT XML Reference Guide](#).

Optional Capabilities

The optional capabilities are used to enable or disable specific options or features of DesignXplorer according to the level of support provided by the selected external method. After the main capabilities have established that a method is applicable to a given DOE or optimization study, the optional capabilities determine which features to expose for that method in the DesignXplorer user interface.

For example, the `LogFile` capability is declared for the sampling defined for the external DOE, **Full-Factorial**, and also for the external optimizer defined in **MyFirstOptimizer**. As a result, the corresponding features for managing and exposing the log file are enabled in DesignXplorer and exposed in the user interface. If this capability was not declared or was set to `false`, DesignXplorer would adjust its user interface to hide the access to the log file because it is not supported by this external method.

As additional examples:

- If the selected sampling does not support the **Custom Table** capability, **Custom Table** is not available in the user interface.
- If the selected optimizer does not support the **Maximize** objective type, the objective types available in the user interface are limited to **Minimize** and **Seek Target**.
- If the optimizer does not support importance levels on objectives, the property **Objective Importance** is not available in the **Property** view of the optimization criterion.

For a comprehensive list of capabilities, see the "Sampling" and "Optimizer" sections in the [ANSYS ACT API Reference Guide](#) and [ANSYS ACT XML Reference Guide](#).

Notes on Method Class Implementation

An implementation of the main interface method class (**ISamplingMethod** for DOE and **IOptimizationMethod** for optimization) is mandatory for a DesignXplorer extension to be valid. This method class is the core of the extension. It is the object that is delegated the responsibility to generate the sampling or solve the optimization study.

If you start a completely new implementation of the sampling or optimization algorithm, you might consider writing a C# class that derives directly from the method class, as provided by the **PublicAPIs** assembly.

If you start from an existing code, either in C/C++ or FORTRAN code, you should consider implementing your method class in IronPython as an adapter to your algorithm implementation, wrapping your existing classes and routines.

Both approaches are illustrated by the extension examples listed in [DesignXplorer Extension Examples \(p. 282\)](#).

For full reference documentation of the API, see the "API Description" section in the [ANSYS ACT API Reference Guide](#).

Notes on Monitoring

Notes on Sampling Monitoring

DesignXplorer provides user interface blocks that allow the user to monitor the progress of a sampling, in addition to the progress and log messages.

Notes on Optimization Monitoring

DesignXplorer provides user interface blocks that allow the user to monitor the progress of an optimization. In addition to the progress and log messages, the API allows the external optimizer to optionally provide DesignXplorer with progress data such as history and convergence values.

The **history values** are values of the input and output variables and show how the optimizer explores the parametric space. In the user interface, history values are rendered graphically, as a 2D chart per objective and input parameter, where the X axis represents points or iterations and the Y axis represents the parameter values. For more information, see "Using the History Chart" in the *DesignXplorer User's Guide*.

The X axis can be configured in the XML extension definition file to represent points or iterations using the attribute **HistoryChartXAxisType** for the optimizer block (**byPoint** or **byIteration**).

To provide history values during the execution of `IOptimizationMethod.Run()`, the optimization method can call `IOptimizationServices.PushHistoryPoint(IOptimizationPoint point)`, where point contains a value for some or all enabled input and output variables.

The **convergence values** are independent from the variables. They are the values of one or more convergence metrics, specific to your optimizer and showing its convergence during the process. In the user interface, convergence values are rendered graphically, as a 2D chart with one or several curves. For more information, see "Using the Convergence Criteria Chart" in the *ANSYS DesignXplorer User's Guide*.

To provide convergence values during the execution of `IOptimizationMethod.Run()`, the optimization method can call `IOptimizationServices.PushConvergenceData (IConvergenceData data)`, where data contains the values for one or several convergence criteria.

Notes on Results

Notes on Sampling Results

The DOE postprocessing is similar for all DOE methods. Once the DOE is generated, DesignXplorer extracts all results provided by the sampling to generate postprocessing tables and charts:

- Sample points from the property `ISamplingMethod.Samples`

Notes on Optimization Results

The optimization postprocessing is similar for all optimization methods. Once the optimization is solved, DesignXplorer checks `IOptimizationMethod.PostProcessingTypes` and extracts all results provided by the optimizer to generate postprocessing tables and charts:

- Candidate points from the property `IOptimizationMethod.Candidates`
- Sample points from the property `IOptimizationMethod.Samples`
- Pareto fronts from the property `IOptimizationMethod.ParetoFrontIndex`

In some cases, DesignXplorer builds missing results if one of these result types is not supported. For example, if the optimization method provides the sample points without the candidate points, DesignXplorer applies its own sorting logic to generate the candidate points based on objectives and constraints definitions. If the optimizer provides the sample points without the Pareto fronts indices, DesignXplorer builds missing data based on objectives and constraints definitions.

Capabilities for ANSYS AIM

This section discusses customization of ANSYS AIM. Using ACT, you can customize AIM by adding a custom load for a structural analysis (similar to the functionality available in ANSYS Mechanical) and create a full CFD system encapsulating all available fluid flow conditions.

The following topics are discussed:

[Adding a Preprocessing Feature in ANSYS AIM Structural](#)

[Creating a Custom Object to Merge Existing AFD Features \(Process Compression\)](#)

[Using SetScriptVersion Within ACT Extensions for ANSYS AIM](#)

Adding a Preprocessing Feature in ANSYS AIM Structural

This section discusses customization of ANSYS AIM for the native exposure of a preprocessing feature—specifically, how to add a custom load to a structural analysis. The example used in the discussion is defined in the extension `CustomPressure`, which was written to create a custom pressure as a load. This example is for demonstration purposes only.

The XML extension definition file and IronPython script are addressed separately.

Custom Load Definition in the XML Extension Definition File

The XML extension definition file `CustomPressure.xml` follows. This code adds the custom pressure load to the project.

```
<extension version="1" name="CustomPressure">
  <guid shortid="CustomPressure">314bd00a-2f64-4b62-8196-bab9206c2c6b</guid>
  <script src="main.py" />

  <simdata context="Study">
    <load name="CustomPressure" version="1" caption="CustomPressure" icon="tload" issupport="false"
      isload="true" color="#0000FF" >

      <callbacks>
        <getsolvecommands order="1">writeNodeId</getsolvecommands>
      </callbacks>

      <property name="Geometry" caption="Geometry" control="scoping">
        <attributes selection_filter="face" />
      </property>
      <property name="Expression" caption="Expression" datatype="double" control="float"
        unit="Pressure" isparameter="true"></property>

    </load>
  </simdata>
</extension>
```

As in previous examples, this file first defines the extension using the attributes `version` and `name`. The `<script>` block specifies the IronPython script `main.py`.

The `<simdata>` block encapsulates the definition of the load. The attribute `context` is set to `Study` to indicate that the extension is for AIM. The attributes in the `<load>` block provide the name, version, caption, icon, support status, and color that apply to this load. The attribute `color` is defined in a hexadecimal format. This color is used to contrast the load when it is displayed on the model. The `<callbacks>` block encapsulates the function `<getsolvecommands>`. This IronPython function is called when the solver input file is generated by AIM. Consequently, the related IronPython function is responsible for generating the commands that describe the load within the ANSYS input file.

In the callback definition, the properties to apply to the actual definition of the load are specified. These properties are displayed in the specific ACT object created by the extension, where the user provides the necessary values to complete the load definition.

Neither of these properties requires you to define a callback.

- The first property, `Geometry`, uses a `scoping` control type and has the attribute `selection_filter` set to `face`. This property allows the user to select the face of the geometry to which to apply the custom pressure.
- The second property, `Expression`, uses a `float` control type, which provides the property with access to AIM expressions functionality. Consequently, you can enter an expression into the `Expression` field and

expect a float. Setting **datatype** to **double** introduces a physical quantity dependency with the **unit** option, which is set to **Pressure**. This combination specifies that this property is consistent with a pressure. Finally, setting **isparameter** to **true** specifies that this expression can be parameterized.

Custom Load Definition in the IronPython Script

The IronPython script `main.py` for the extension `CustomPressure` follows.

```
def writeNodeId(load, stream):
    ExtAPI.Log.WriteMessage("writeNodeId... ")
    stream.WriteLine("/com, GetSolveCommands")

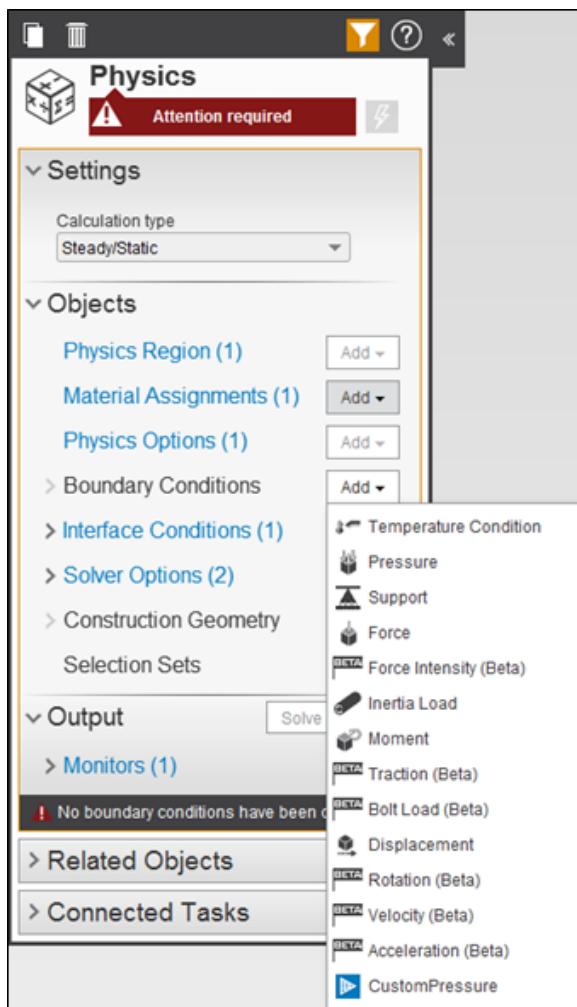
    property_expression = load.PropertyByName("Expression")
    expression = property_expression.Value
    ExtAPI.Log.WriteMessage(expression.ToString())
    if expression=="":
        return None

    property_geometry = load.PropertyByName("Geometry")
    refIds = property_geometry.Value.Ids
    ExtAPI.Log.WriteMessage(refIds.ToString())
    mesh = ExtAPI.DataModel.MeshDataByName("Mesh 1")
    for refId in refIds:
        meshRegion = mesh.MeshRegionById(refId)
        ExtAPI.Log.WriteMessage(meshRegion.ToString())
        nodeIds = meshRegion.NodeIds
        for nodeId in nodeIds:
            node = mesh.NodeById(nodeId)

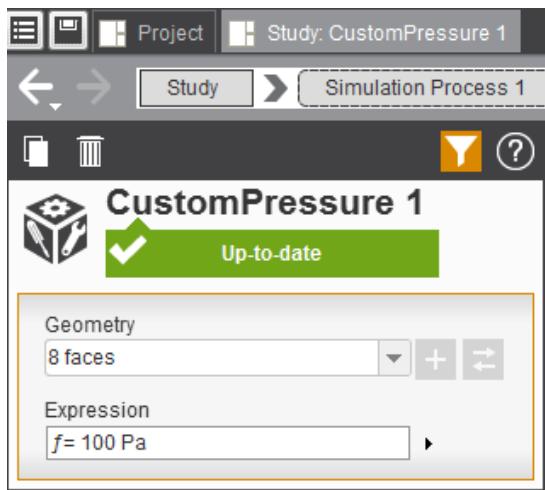
        stream.WriteLine("F,"+nodeId.ToString()+" , FX,"+expression.ToString()+"\n")
```

The script `main.py` defines the function `writeNodeId`, which writes the value of the property `Expression` on the geometry nodes scoped for the load.

The following figure shows how the load `CustomPressure` that you created in ACT is now available as a boundary condition in AIM.



The following figure shows the status of the load once all of the properties are defined.



Creating a Custom Object to Merge Existing AFD Features (Process Compression)

By using ACT extensions in AIM, you have the ability to create a complete CFD system encapsulating existing boundary conditions. This section discusses the customization of AIM to expose a fluids system with access to all available boundary conditions.

You can have one load per boundary condition and have access to all fluid boundary conditions (such as inlet, outlet, wall, and so on). The boundary conditions appear as scoping properties. In some instances, you can also add an inlet velocity and an outlet pressure to standard boundary conditions.

Fluids extensions developed in AIM are very similar in construction to other extensions developed in AIM or Workbench. As with other extensions, each AFD extension developed in AIM has an XML extension definition file and an IronPython script.

- The XML file has a block for defining buttons and toolbars and a block containing callbacks and properties for defining the load.
- The script defines the functions invoked by the events and callbacks in the XML file to implement the behavior of the extension.

This section describes a theoretical extension named `AFDLoad`. This example is for demonstration purposes only.

CFD System Definition in the XML Extension Definition File

The XML extension definition file `AFDLoad.xml` follows. This extension is used to create a CFD system with full access to available boundary conditions.

```
<extension version="1" name="AFDLoad">

    <script src="main.py" />

    <interface context="Study">
        <images>images</images>
    </interface>

    <simdata context="Study">
        <load name="InOut" version="1" caption="InOut" icon="support" issupport="true" color="#0000FF">
            <callbacks>
                <getsolvecommands>getcmds</getsolvecommands>
            </callbacks>

            <property name="Inlet" caption="Inlet" control="scoping" />
            <property name="Outlet" caption="Outlet" control="scoping" />
            <property name="InletVelocity" caption="Inlet Velocity" control="float" unit="Velocity"
            default="1 [m s^-1]" />
            <property name="OutletPressure" caption="Outlet Pressure" control="float" unit="Pressure"
            default="0 [Pa]" />
        </load>
    </simdata>
</extension>
```

This XML file has the standard ACT construction, with the `<extension>` block defining the attributes `version` and `name`, and the `<script>` block specifying the IronPython script `main.py`. The `<interface>` block sets the attribute `context` to `Study` to indicate that the extension is for AIM.

As in the AIM preprocessing example, the `<simdata>` block encapsulates the definition of the load. The attribute `context` is set to `Study`, and the attributes in the `<load>` block provide the name, version, caption, icon, support status, and color that apply to the load that contains the boundary condition. In this example, both the load name and caption are `InOut`. As before, the `<callbacks>` block encapsulates the function `<getsolvecommands>`, which creates the boundary conditions for the load.

In the callbacks definition, the load properties to use for the boundary conditions are defined: `Inlet`, `Outlet`, `Inlet Velocity`, `Velocity`, `Outlet Pressure`, and `Pressure`. The attributes `controlUnit`, and `default` function as described in previous examples. These properties are displayed in the specific ACT object created when the user provides the necessary values to complete the load definition.

CFD System Definition in the IronPython Script

The IronPython script `main.py` for the extension `AFDLoad` follows.

Note

In future releases, the IronPython commands used to define fluid dynamics features might be modified to ensure consistency with AIM journaling and scripting functionality. For information on whether a given command has been modified or deprecated, refer to the [ANSYS ACT API Reference Guide](#).

```
import clr
clr.AddReference("Ansys.Study.Proxies.Customization")
from Ansys.Study.Proxies.Customization.Analysis import AFDOBJECT

def getcmds(load,stream):
    velocity = load.Properties["InletVelocity"]
    pressure = load.Properties["OutletPressure"]

    faces = []
    faces += load.Properties["Inlet"].Value.Ids
    faces += load.Properties["Outlet"].Value.Ids
    geo = load.Analysis.GeoData
    wallFaces = []
    for part in geo.Assemblies[0].Parts:
        for body in part.Bodies:
            for face in body.Faces:
                if not faces.Contains(face.Id) and not wallFaces.Contains(face.Id):
                    wallFaces.Add(face.Id)

    boundary = AFDOBJECT.CreateBoundary()
    boundary SetProperty("BoundaryType", "Inlet")
    boundary SetProperty("Location", load.Properties["Inlet"].Value.Ids)
    boundary SetProperty("Flow.Option", "Velocity")
    boundary SetProperty("Flow.Velocity.Magnitude", velocity.Value.ToString() + ['+velocity.UnitString+'])
    boundary.Send(stream)
    boundary = AFDOBJECT.CreateBoundary()
    boundary SetProperty("BoundaryType", "Outlet")
    boundary SetProperty("Location", load.Properties["Outlet"].Value.Ids)
    boundary SetProperty("Flow.Option", "Pressure")
    boundary SetProperty("Flow.Pressure.GaugeStaticPressure", pressure.Value.ToString() + [
        '+pressure.UnitString+'])
    boundary.Send(stream)
```

```
boundary = AFDOBJECT.CreateBoundary()
boundary SetProperty("BoundaryType", "Wall")
boundary SetProperty("Location", wallFaces)
boundary.Send(stream)
```

The function **getcmds** is defined in the script `main.py`. This function writes commands to the solver input file. The required input arguments are **load** and **stream**, where **load** is the load object and **stream** is the solver input file. In the XML file, the callback `<getsolvecommands>` references the function **getcmds**. When called, this function creates the boundaries for **AFDOBJECT**, defines boundary properties, and specifies the faces to which to apply the boundary. Within this function:

- The method **CreateBoundary()** is applied to **AFDOBJECT** to create a boundary. (**AFDOBJECT** was imported by the command **import** at the top of the script.)

```
boundary = AFDOBJECT.CreateBoundary()
```

- In the section **faces**, you can specify:
 - The faces to use in the **inlet** and **outlet** properties.
 - The faces belonging to the **wall** (which are all faces not already used for the **inlet** and **outlet** properties).
- With **boundary SetProperty**, you specify the boundary type, such as inlet, outlet, wall, and so on. It can also be applied to the boundary object to add a property. As part of the argument, you must specify both the type of property to add and the value to assign.

For example, the following code segment specifies the creation of an inlet boundary.

```
boundary SetProperty("BoundaryType", "Inlet")
```

- With **boundary SetProperty**, you can also specify the location (faces, bodies, and so on) to which the boundary applies. To do so, you can use the scoping properties previously defined in the extension. The property **location** expects a list of identifiers.

```
boundary SetProperty("Location", load.Properties["Inlet"].Value.Ids)
```

- To the property just added, you can add an option representing a velocity (for an inlet) or a pressure (for an outlet).

```
boundary SetProperty("Flow.Option", "Velocity")
```

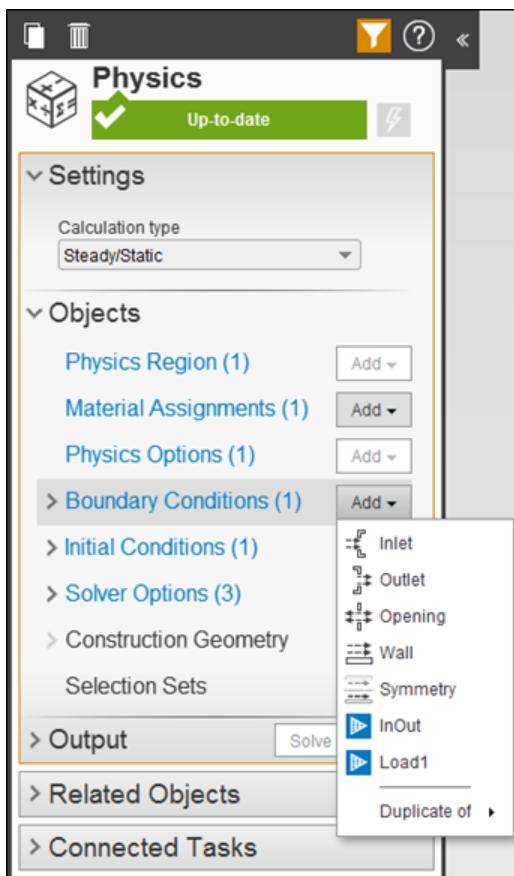
- To the option just added, you can add a float value and specify a unit. The float value can come from a property defined in the XML file.

```
boundary SetProperty("Flow.Velocity.Magnitude", velocity.Value.ToString() + '[' + velocity.UnitString + ']')
```

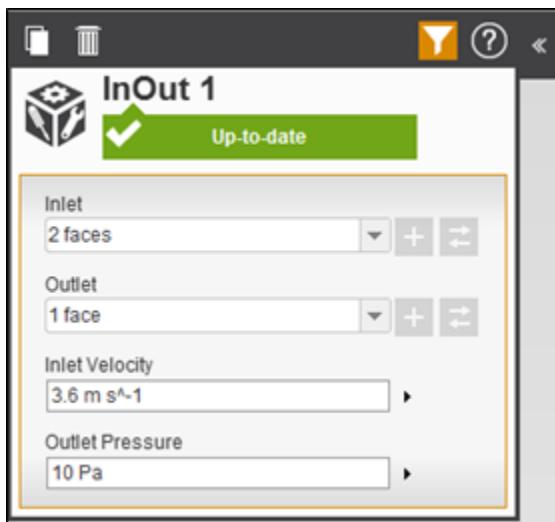
- Finally, you write the command generated by creation of the boundary.

```
boundary.Send(stream)
```

The following figure shows that the load **InOut** that you created in ACT is now available as a fluids boundary condition in AIM. This load includes all of the properties defined in the extension.



The following figure shows the status of the load once all of the properties are defined.



Using SetScriptVersion Within ACT Extensions for ANSYS AIM

When creating an ACT callback, to ensure that any migration can occur in a subsequent release, the best practice is to use the following pattern for setting and resetting script versions. This ensures that the commands being used today are correctly executed and that the product version is kept at the version where the ACT extension is used.

```
# Store current Script Version
currentVersion = GetScriptVersion()
# Set the Script Version to the appropriate version
```

```
SetScriptVersion(Your version)
# Insert your script
...
# Reset the Script Version
SetScriptVersion(currentVersion)
```

Capabilities for ANSYS Fluent

When creating ACT extensions for ANSYS Fluent, in an extension's XML definition file, the **<simdata>** block can contain a child **<udf>** block for specifying the location of a folder containing UDF libraries. When supplying the path for a UDF folder, you use the same logic as when supplying the path for the extension's IronPython script, where **src** is the path relative to the extension folder:

```
<script src="main.py"/>
```

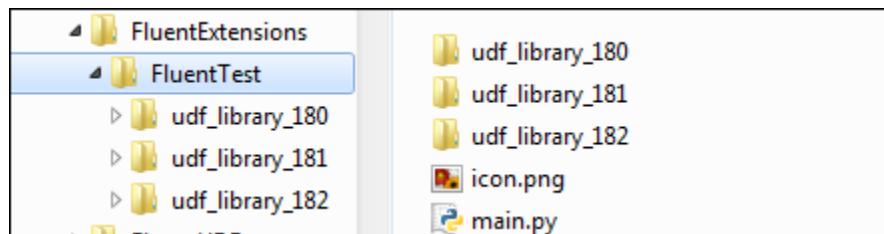
While paths for both the **<script>** block and **<udf>** block are relative to the extension folder, the **<script>** block requires a *file relative path*, and the **<udf>** block requires a *folder relative path*. An example of the XML code for an extension with a **<udf>** block follows:

```
<extension name="FluentTest" version="1" icon="icon.png">
  <guid>36DB3B25-C5E2-4E43-9A04-BA53AA7DC05A</guid>
  <author>ANSYS, Inc.</author>
  <interface context="Fluent"></interface>
  <description>A sample extension for Fluent with UDF encapsulation</description>
  <script src="main.py" />
  <simdata context="Fluent">
    <udf folder="udf_library"></udf>
  </simdata>
</extension>
```

Because the UDF libraries in the specified folder are packaged with the extension, when this extension is installed, ACT can load these libraries.

When you specify the UDF folder name in the **<udf>** block, you do not specify a Fluent version. The ACT extension automatically appends the Fluent version in which the extension is running to the name of the folder. However, because UDF libraries are version-dependent, you must name each of the real UDF folders with the specified name followed by an underscore and the version.

In the previous example, the name specified for the UDF folder is **udf_library**. However, the real folders are named **udf_library_180**, **udf_library_181**, and **udf_library_182**. The extension searches for the name given in the **<udf>** block followed by the Fluent version in which you are running the extension.



Thus, the naming convention of the UDF folder in the XML file is **filename**, and the naming convention for the real folder is **filename_version**.

Simulation Workflow Integration

With ACT, you can take your custom external applications and processes and integrate them into the ANSYS Workbench workflow. Features exposed by ACT also enable you to perform automation and customization activities, such as creating new systems to facilitate interaction with the Workbench **Project Schematic**. The ACT API delivers functionality previously available only through the External Connection add-in or the SDK in an improved and streamlined offering.

The following sections describe how to use custom ACT workflows in Workbench:

[Exposing Custom Task Groups and Tasks on the Project Schematic](#)

[Using Global Callbacks](#)

[Progress Monitoring](#)

[Using Process Utilities](#)

[The Custom Workflow Creation Process](#)

[Creating the XML Definition File](#)

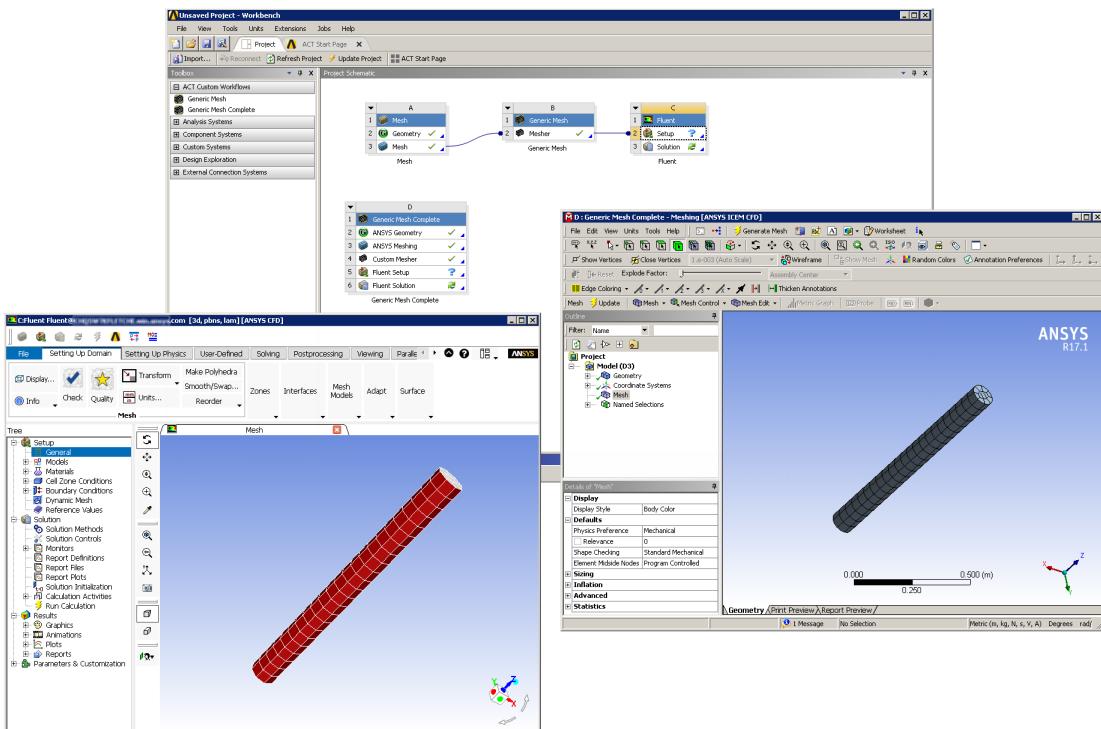
[Creating the IronPython Script](#)

Exposing Custom Task Groups and Tasks on the Project Schematic

You can use ACT to create custom *task groups* and custom *tasks* on the **Project Schematic**. Workbench refers to a task group as a *system* and to a task as a *cell*.

- *Task groups* contain tasks. In Workbench, task groups are exposed as custom systems. Once defined and added to the Workbench **Toolbox**, an ACT-defined task group can be added to the **Project Schematic** workflow in the same way as an installed (Workbench-defined) system.
- *Tasks* are the cells contained in a task group. In Workbench, ACT-defined tasks can be exposed as custom components. Both custom tasks and external tasks (Workbench components, which are defined outside the extension) can be added to a custom task group.

The following figure shows a Workbench Mesh Transfer task group that takes an upstream mesh and passes it to a downstream Fluent task group. For more information, see the [Generic Mesh Transfer \(p. 302\)](#) example.



Using Global Callbacks

You can use ACT to create callbacks that invoke custom actions within the Workbench project workflow. Global callbacks are available for both schematic-level actions and Workbench-level actions. Schematic-level actions are executed on the **Project Schematic**. Workbench-level actions are executed elsewhere in the project, such as on the **Parameter Set** bar.

Related Topics:

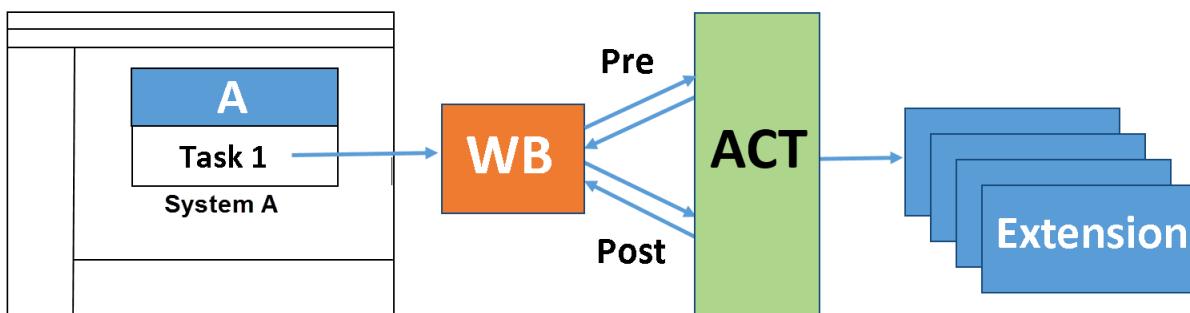
- [Invoking Custom Schematic Actions](#)
- [Invoking Custom Workbench Actions](#)
- [Global Callback Observation Process](#)

Invoking Custom Schematic Actions

ACT global callbacks for schematic actions provide hooks that listen and respond to activity on the Workbench **Project Schematic**. You can use these hooks to define custom processes or actions to be executed at specific times in the Workbench schematic workflow.

Schematic callback support is accomplished with the **Action Observation** feature. This feature provides notification before and after every action processed by the schematic. This enables ACT to identify a point in the workflow so you can specify that a custom action be executed either before or after that point. Both pre- and post-action callback support has been added to eleven distinct schematic actions (such as Create, Update, Refresh, and so on.)

Global Callbacks = green



ACT does not observe schematic activities until at least one extension provides one callback for a schematic action. At that time, ACT starts observing the requested schematic action and invokes your callback. Most of the schematic callbacks receive a user task as an argument. (For more information, see the [ANSYS ACT API Reference Guide](#).)

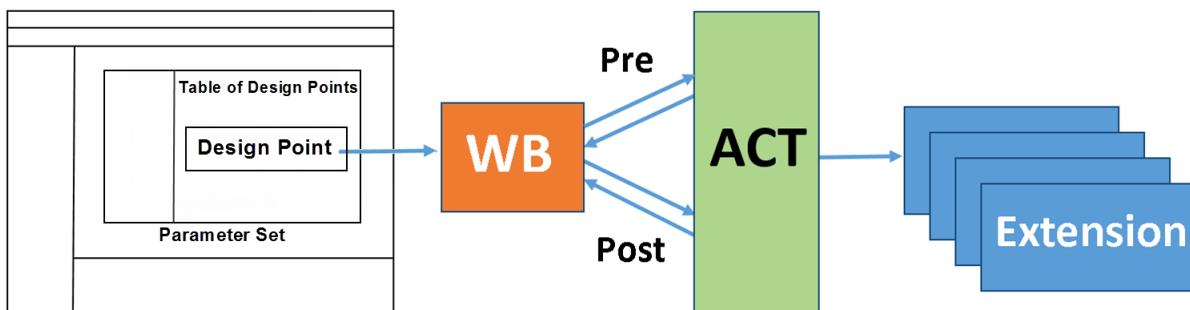
Once observations of schematic actions have begun, a user-registered schematic callback is invoked for every component that exists in the **Project Schematic**. This applies to both installed (Workbench-defined) and custom (ACT-defined) tasks. For example, if you create an analysis system and update the project, ACT invokes the update-related callbacks up to n times, where n = the number of components (tasks) in the system (task group).

Invoking Custom Workbench Actions

ACT global callbacks for Workbench actions provide hooks that listen and respond to activity that occurs within the project, but outside the **Project Schematic**. You can use these hooks to define custom processes or actions to be executed at specific times in the Workbench project workflow.

Currently, pre- and post-execution callbacks are available for changes to project design points, which is executed in the **Table of Design Points** in the **Parameter Set** bar.

Global Workbench Callbacks = Green



Global Callback Observation Process

For both schematic and Workbench actions, you can manipulate the observation process.

Accessing Information

For task-related callbacks, from the user task, you can access name information, template information, properties, and the task's underlying container via the property **InternalObject**.

Querying

For queries, you can alter the action's return value in the post-execution callback.

Aborting an Action

To abort an action, you can throw an exception from the pre-execution callback.

Filtering Components

To process only the callback for a specific task or template, you must filter the components by performing a check inside the callback implementation based on the supplied arguments.

For example, for schematic actions, you can access the task names or containers to decide whether or not to take action. If the action must apply only to all **Fluent Setup** components, you would create an update callback that contains the following code:

```
def myPostUpdateCallback(task):
    container = task.InternalObject
    if container != None and container.Type.Equals('Ansys.Fluent.Addin:SetupContainer'):
        #perform your post-update action on the fluent setup container
    Stopping Observations:
```

When you unload all extensions that register global callbacks, ACT stops observing the specified actions. This sets Workbench back to its prior state and can be used as a fix if either schematic or Workbench observations cause unexpected behavior.

Progress Monitoring

ACT provides progress monitoring for custom workflows via the property **ExtAPI.UserInterface.ProgressMonitor**. This property returns an object of type **IProgressMonitor**.

The following APIs are available:

Class	Member	Description
IProgressMonitor	WorkName	The name of the current work under progress monitoring.
	WorkStatus	The current status for the work under progress monitoring.
	WorkDetails	The current details for the work under progress monitoring.
	TotalWorkUnits	The total number of work units allocated to the work under progress monitoring.
	CompletedWorkUnits	The completed number of work units.
	Parent	The parent progress monitor, if this is a child monitor. Otherwise, null.
	PropagateWorkStatus	Indicates whether to propagate a child's work status up to the parent.

Class	Member	Description
	PropagateWorkDetails	Indicates whether to propagate a child's work details up to the parent.
	CanAbort	Indicates whether the user can cancel the current operation.
	CanInterrupt	Indicates whether the user can interrupt the current operation.
	BeginWork(name, units)	Starts work engaged with progress monitoring.
	UpdateWork(units)	Updates the current work units.
	EndWork()	Marks the current work as complete (progress at 100%).
	CreateChildMonitor(units)	Creates a child progress monitor with the current monitor as its parent.
	Status	The current status of this progress monitor. ProgressStatus Enum Values: <ul style="list-style-type: none"> • NotStarted • Running • Aborted • Interrupted • Completed

A sample implementation of progress monitoring functionality follows.

```
def update(task):

    monitor = ExtAPI.UserInterface.ProgressMonitor
    monitor.BeginWork("Testing 123...", 3)
    LogProgress(monitor)
    monitor.WorkStatus = "Test Status!"
    monitor.WorkDetails = "Test Details!"
    System.Threading.Thread.Sleep(2000)
    monitor.UpdateWork(1)
    LogProgress(monitor)
    monitor.WorkDetails = "More details..."
    System.Threading.Thread.Sleep(2000)
    monitor.UpdateWork(1)
    LogProgress(monitor)
    monitor.WorkDetails = "Even more details..."
    System.Threading.Thread.Sleep(2000)
    monitor.UpdateWork(1)
    LogProgress(monitor)
```

```

monitor.WorkDetails = "Final details..."
System.Threading.Thread.Sleep(2000)
monitor.EndWork()
def LogProgress(monitor):
    ExtAPI.Log.WriteMessage("Progress: " + str(monitor.CompletedWorkUnits)
                           + "/" + str(monitor.TotalWorkUnits))

```

Using Process Utilities

ACT process utility capabilities enable you to execute any outside program with a single method call. To do this, you call the method **Start** exposed on the object **ExtAPI.ProcessUtils**.

The following APIs are available:

Class	Member	Description
ExtAPI.ProcessUtils	Start(target, args)	Starts an application, file, or other target. Returns an integer error code from the resulting process. Waits for process to exit.
	Start(target, useShell, args)	Starts an application, file, or other target. useShell indicates whether or not to use the running OS's shell to execute the target.

A sample implementation of process utilities follows.

```

import System

def update(task):
    activeDir = task.ActiveDirectory
    extensionDir = task.Extension.InstallDir
    exeName = "ExampleAddinExternalSolver.exe"
    solverPath = System.IO.Path.Combine(extensionDir, exeName)

    inputValue = task.Properties["Inputs"].Properties["Input"].Value

    inputFileName = "input.txt"
    outputFileName = "output.txt"
    dpInputFile = System.IO.Path.Combine(activeDir, inputFileName)
    dpOutputFile = System.IO.Path.Combine(activeDir, outputFileName)

    #write input file
    f = open(dpInputFile, "w")
    f.write('input=' + inputValue.ToString(
        System.Globalization.NumberFormatInfo.InvariantInfo))
    f.close()

    exitCode = ExtAPI.ProcessUtils.RunApplication(solverPath,
                                                dpInputFile, dpOutputFile)
    if exitCode != 0:
        raise Exception ('External solver failed!')

    #read output file

    outputValue = None
    f = open(dpOutputFile, "r")
    currLine = f.readline()
    while currLine != "":
        valuePair = currLine.split('=')
        outputValue = System.Double.Parse(valuePair[1],
                                         System.Globalization.NumberFormatInfo.InvariantInfo)
        currLine = f.readline()
    f.close()

```

```

if outputValue == None:
    raise Exception("Error in update - no output value detected!")
else:
    task.Properties["Outputs"].Properties["Output"].Value = outputValue

```

The Custom Workflow Creation Process

As with any other type of extension, creating task groups and tasks to be exposed in a Workbench workflow involves the creation, installation, and loading of the custom workflow extension.

The steps to create a custom workflow extension are the same as for any ACT extension. At minimum, you must create the XML definition file, the IronPython script, and any support files (such as task group and task images and custom application executables). The IronPython script and support files should be placed in the extension directory, which is at the same level as the XML file.

- For general information on creating extensions, see [Creating a Scripted Extension \(p. 11\)](#).
- For information on how to place your files within the extension, see [Setting Up the Directory Structure \(p. 13\)](#).

To use the extension you've created, you must install it as described in [Installing a Scripted Extension \(p. 33\)](#).

Once you've installed your extension, you can load it into Workbench via the **Extension Manager**. On loading your extensions, the functionality defined in the extension becomes available.

- If you have defined global callbacks, they are invoked within the **Project Schematic** workflow or other Workbench project location as defined. Each global callback is invoked each time the operation associated with it is performed.
- If you have defined custom task groups or tasks, the custom task groups are exposed as systems in a new group in the Workbench **Toolbar**.



These custom task groups can be added to the **Project Schematic**. When you invoke an **Update** operation on a custom task within the task group, the extension can:

- Obtain the inputs
- Prepare the inputs
- Write input files

- Run the external solver or performs calculations
- Read output files
- Set the parameters or properties to the calculated values

Creating the XML Definition File

The extension's XML definition file contains information for defining a workflow within Workbench, including task groups (systems) designed for a particular simulation objective. All analyses performed within Workbench begin by referencing a task group. The XML file describes the task groups and declares the contained tasks.

At the extension level, the XML file must specify, at minimum:

- Extension name
- Interface context (must be **Project**)
- Interface **<images>** block (needed to specify task group or task icons)
- Workflow definition

At the workflow level, the XML file can specify:

- Workflow context (required and must be **Project**)
- Tasks (container for individual task definitions)
- Task groups (container for task group definitions)
- Global callbacks

At the global callback level, the XML file can specify pre- and post-operation callbacks for schematic and Workbench operations. For more information, see [Defining a Global Callback \(p. 118\)](#) and [Using Global Callbacks \(p. 110\)](#).

At the task level, the XML file can specify:

- Task name (required)
- Task version (required)
- Display text
- Image name
- Context menus
- Callbacks
- Property groups and properties
- Inputs and outputs
- Parameters

- Remote job execution specifications

For more information, see [Defining a Task \(p. 124\)](#).

At the task group level, the XML file can specify:

- Task group name (required)
- Task group version (required)
- Header text
- Toolbox category
- Parametric support
- Image name
- Abbreviation

For more information, see [Defining a Task Group \(p. 141\)](#).

Basic Structure of the XML Definition File

The following example of an XML definition file shows you its basic structure:

```
<extension version="1" name="">
    <guid shortid="">69d0095b-e138-4841-a13a-de12238c83f3</guid>
    <script src="" />
    <interface context="Project">
        <images>images</images>
    </interface>
    <workflow name="" context="Project" version="1">
<callbacks>
    <onbeforetaskreset></onbeforetaskreset>
    <onaftertaskreset></onaftertaskreset>
    <onbeforetaskrefresh></onbeforetaskrefresh>
    <onaftertaskrefresh></onaftertaskrefresh>
    <onbeforetaskupdate></onbeforetaskupdate>
    <onaftertaskupdate></onaftertaskupdate>
    <onbeforetaskduplicate></onbeforetaskduplicate>
    <onaftertaskduplicate></onaftertaskduplicate>
    <onbeforetasksourceschanged></onbeforetasksourceschanged>
    <onaftertasksourceschanged></onaftertasksourceschanged>
    <onbeforetaskcreation></onbeforetaskcreation>
    <onaftertaskcreation></onaftertaskcreation>
    <onbeforetaskdeletion></onbeforetaskdeletion>
    <onaftertaskdeletion></onaftertaskdeletion>
    <onbeforetaskcanusetransfer></onbeforetaskcanusetransfer>
    <onaftertaskcanusetransfer></onaftertaskcanusetransfer>
    <onbeforetaskcanduplicate></onbeforetaskcanduplicate>
    <onaftertaskcanduplicate></onaftertaskcanduplicate>
    <onbeforetaskstatus></onbeforetaskstatus>
    <onaftertaskstatus></onaftertaskstatus>
    <onbeforetaskpropertyretrieval></onbeforetaskpropertyretrieval>
    <onaftertaskpropertyretrieval></onaftertaskpropertyretrieval>
    <onbeforedesignpointchanged></onbeforedesignpointchanged>
    <onafterdesignpointchanged></onafterdesignpointchanged>
</callbacks>
<tasks>
    <task name="" caption="" icon="" version="1">
        <callbacks>
            <onupdate></onupdate>
            <onrefresh></onrefresh>
            <oninitialize></oninitialize>
```

```
<onedit></onedit>
<onreset></onreset>
<onstatus></onstatus>
<onreport></onreport>
</callbacks>
<contextmenus>
    <entry name="" caption="" icon="" priority="" type="">
        <callbacks>
            <onclick></onclick>
        </callbacks>
    </entry>
</contextmenus>
<propertygroup name="" caption="">
    <property name="" caption="" control="" default="" readonly="" needupdate="" visible="" persistent="" isparameter="" keytype="" valuetype="" elementtype="" />
</propertygroup>
    <property name="" caption="" control="" default="" readonly="" needupdate="" visible="" persistent="" isparameter="" keytype="" valuetype="" elementtype="" />
</propertygroup>
<parameters>
    <parameter name="" caption="" usage="" control="" version="1" />
</parameters>
<inputs>
    <input/>
    <input type="" format="" />
</inputs>
<outputs>
    <output type="" format="" />
</outputs>
<rsmjob name="" deletefiles="" version="1">
    <inputfile id="1" name="" />
    <outputfile id="1" name="" />
    <program>
        <platform name="" path="" />
        <argument name="" value="" separator="" />
    </program>
    <callbacks>
        <oncreatejobinput></oncreatejobinput>
        <onjobstatus></onjobstatus>
        <onjobcancellation></onjobcancellation>
        <onjobreconnect></onjobreconnect>
    </callbacks>
</rsmjob>
</task>
</tasks>
<taskgroups>
    <taskgroup name="" caption="" icon="" category="" abbreviation="" version="1" isparametricgroup="False">
        <includetask name="" external="" />
        <includeGroup name="" />
    </taskgroup>
</taskgroups>
</workflow>
</extension>
```

Defining a Global Callback

Within the `<workflow>` block, one or more global callbacks (either for schematic or Workbench operations) can be defined in a child `<callbacks>` block.

ACT currently provides 22 schematic callbacks. Each of the 11 operations available on the **Project Schematic** support both a pre- and post-operation callback.

ACT also provides two non-schematic, Workbench callbacks. The change design point operation supports both a pre- and post-operation callback.

The basic structure of a global callback definition follows.

```

<callbacks>
  <onbeforetaskreset></onbeforetaskreset>
  <onaftertaskreset></onaftertaskreset>
  <onbeforetaskrefresh></onbeforetaskrefresh>
  <onaftertaskrefresh></onaftertaskrefresh>
  <onbeforetaskupdate></onbeforetaskupdate>
  <onaftertaskupdate></onaftertaskupdate>
  <onbeforetaskduplicate></onbeforetaskduplicate>
  <onaftertaskduplicate></onaftertaskduplicate>
  <onbeforetasksourceschanged></onbeforetasksourceschanged>
  <onaftertasksourceschanged></onaftertasksourceschanged>
  <onbeforetaskcreation></onbeforetaskcreation>
  <onaftertaskcreation></onaftertaskcreation>
  <onbeforetaskdeletion></onbeforetaskdeletion>
  <onaftertaskdeletion></onaftertaskdeletion>
  <onbeforetaskcanusetransfer></onbeforetaskcanusetransfer>
  <onaftertaskcanusetransfer></onaftertaskcanusetransfer>
  <onbeforetaskcanduplicate></onbeforetaskcanduplicate>
  <onaftertaskcanduplicate></onaftertaskcanduplicate>
  <onbeforetaskstatus></onbeforetaskstatus>
  <onaftertaskstatus></onaftertaskstatus>
  <onbeforetaskpropertyretrieval></onbeforetaskpropertyretrieval>
  <onaftertaskpropertyretrieval></onaftertaskpropertyretrieval>
  <onbeforedesignpointchanged></onbeforedesignpointchanged>
  <onafterdesignpointchanged></onaftersignpointchanged>
</callbacks>

```

Related Topics:

[Invoking Global Callbacks](#)
[Filtering Global Callbacks](#)
[Specifying Global Callback Execution Order](#)
[Available Schematic Callbacks](#)
[Available Workbench Callbacks](#)

Invoking Global Callbacks

By default, if a global schematic callback is defined in the XML definition file, it is called for each instance of the schematic task associated with it. This applies to both pre-installed (non-ACT) and custom tasks. For example, if you have defined the callback `<onaftertaskupdate>`, it is invoked for each task that is updated. If three tasks on the **Project Schematic** are updated, the callback is invoked three times, once after each update.

Unless otherwise noted, a global schematic workflow callback receives a task as the method argument. Depending on the callback, the task is the concrete task displayed in the schematic or the task template.

Filtering Global Callbacks

To specify that a callback is processed only for a specific location in the project, such as the task or template, you can perform a check within the callback implementation based on the supplied arguments. For example, in the following IronPython code sample:

- The method `onBeforeUpdate` is unfiltered. The actions are invoked before the update of each eligible task on the schematic.

- The method **onAfterUpdate** is filtered. It produces different messages for **Engineering Data** tasks and the second **Setup** task, while printing 'ignored' for all other tasks.
-

Note

The **Setup** tasks for multiple Fluent task groups would be identified in the following sequence: **Setup**, **Setup 1**, **Setup 2**, and so on.

- The method **onBeforeDuplicate** is filtered. It prints a specific message for the **Setup** tasks for all Fluent tasks on the schematic.

```
...
def onBeforeUpdate(task):
    msg = getPrintMessage('pre-update', task)
    print msg

def onAfterUpdate(task):
    if task.Name == "Engineering Data":
        msg = getPrintMessage('post-update', task)
        print msg
    elif task.Name == "Setup 2":
        print("fluent setup 2")
    else:
        print("ignored")

def onBeforeDuplicate(task):
    if task.Container.Type == "Ansys.Fluent.Addin:SetupContainer":
        print("all fluent setups")
    else:
        print("ignored")
...

```

Specifying Global Callback Execution Order

When multiple callbacks are defined for the same pre- or post-operation schematic action, you can specify their order of execution by adding the **order** attribute to the callback declaration:

```
<workflow ...>
  <callbacks>
    <onbeforetaskreset order="1">...</onbeforetaskreset>
    ...
  </callbacks>
</workflow>
```

ACT executes the callbacks from lowest order to highest order. This means that a callback with an order of 2 executes after a callback with an order of 1. When values are identical, the order in which the extension is loaded dictates the collision resolution.

Available Schematic Callbacks

This section describes the global callback and corresponding methods and arguments for each schematic operation.

Note

Any hyphenation within long table entries can be ignored. Callbacks and methods do not include hyphens.

Reset

Resets a component back to its pristine, new state.

Global Callbacks	Methods	Arguments
<onbeforetaskreset>	onBeforeReset	task
<onaftertaskreset>	onAfterReset	

Refresh

Consumes all upstream data and prepares any local data for an ensuing update.

Global Callbacks	Methods	Arguments
<onbeforetaskrefresh>	onBeforeRefresh	task
<onaftertaskrefresh>	onAfterRefresh	

Update

Generates all broadcast output types that render the component fully solved.

Global Callbacks	Methods	Arguments
<onbeforetaskupdate>	onBeforeUpdate	task
<onaftertaskupdate>	onAfterUpdate	

Create

Creates a task based on an underlying template.

Global Callbacks	Methods	Arguments
<onbeforetaskcreate>	onBeforeCreate	templateTask Note: This is a task representing the template. Some components do not include the template as a parameter to the command create . As a result, this could be null in some situations.
<onaftertaskcreate>	onAfterCreate	task

Delete

Removes a task from a task group.

Global Callbacks	Methods	Arguments
<onbeforetaskdelete>	onBeforeDelete	task
<onaftertaskdelete>	onAfterDelete	

Duplicate

Creates an identical, yet independent, clone of the task.

Global Callbacks	Methods	Arguments
<onbeforetaskduplicate>	onBeforeDuplicate	task
<onaftertaskduplicate>	onAfterDuplicate	

SourcesChanged

Processes a change in upstream sources.

Global Callbacks	Methods	Arguments
<onbeforetasksourceschanged>	onBeforeSourcesChanged	task
<onaftertasksourceschanged>	onAfterSourcesChanged	

CanUseTransfer

Checks whether the task can consume data from a specific upstream task.

Callback receives two tasks as arguments: the source (upstream producer) task and target (downstream consumer) task.

Global Callbacks	Methods	Arguments
<onbeforetaskcanusetransfer>	onBeforeCanUseTransfer	sourceTask, targetTask
<onaftertaskcanusetransfer>	onAfterCanUseTransfer	sourceTask, targetTask, systemCalculatedCanUse

Note: The argument **systemCalculatedCanUse** provides the result that the original task determined. The return value of this callback (Boolean) alters the originally calculated value.

CanDuplicate

Checks whether the task permits duplication.

Callback receives no arguments. The schematic does not provide any workable information to construct or provide tasks (either container-based or template-based). Callback only serve as a general “processing” hook if exposed.

Global Callbacks	Methods	Arguments
<onbeforetaskcanduplicate>	onBeforeCanDuplicate	None
<onaftertaskcanduplicate>	onAfterCanDuplicate	systemCalculatedCanUse

Note: The argument **systemCalculatedCanDuplicate** provides the result that the original task determined. The return value

Global Callbacks	Methods	Arguments
		of this callback (Boolean) alters the originally calculated value.

Status

Calculates the task's current state.

Global Callbacks	Methods	Arguments
<onbeforetaskstatus>	onBeforeStatus	task
<onaftertaskstatus>	onAfterStatus	task, systemCalculatedStatus Note: The argument systemCalculatedStatus provides the result that the original task determined. The return value of this callback component state alters the originally calculated value.

PropertyRetrieval

Determines the visibility of property-containing objects.

Global Callbacks	Methods	Arguments
<onbeforetaskpropertyretrieval>	onBeforePropertyRetrieval	task
<onaftertaskpropertyretrieval>	onAfterPropertyRetrieval	task, systemCalculatedPropertyObject Note: The argument systemCalculatedPropertyObjects provides the result that the original task determined. The return value of the callback (list of data references) alters the originally calculated value.

Available Workbench Callbacks

This section describes the global callback and corresponding methods for each non-schematic Workbench operation.

designpointchanged

Signals to workflow that a design point change is about to or has occurred.

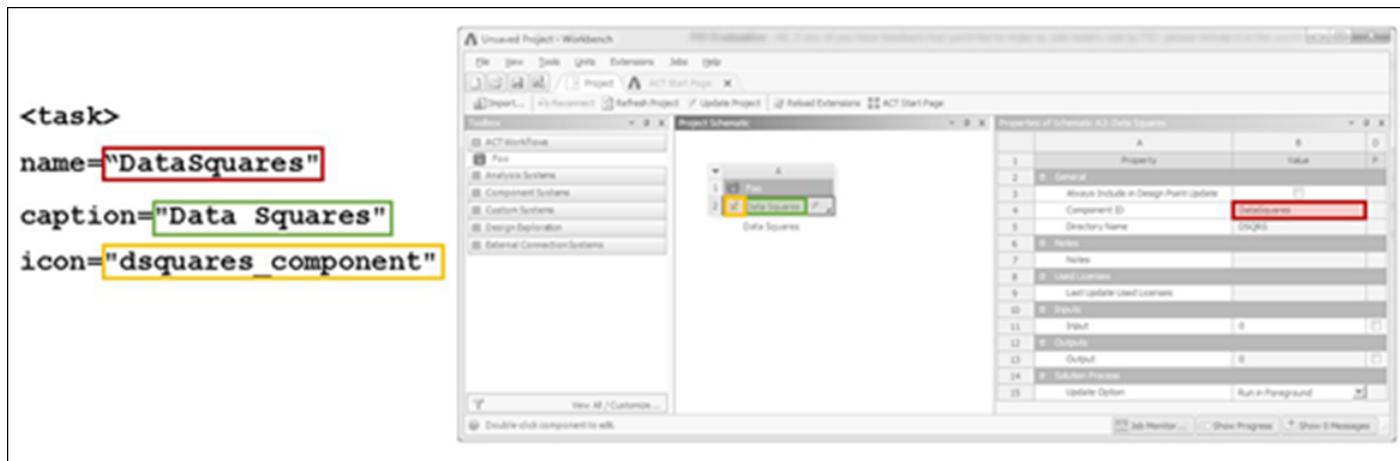
Global Callbacks	Methods
<onbeforedesignpointchanged>	onBeforeDesignPointChanged

Global Callbacks	Methods
<onafterdesignpointchanged>	onAfterDesignPointChanged

Defining a Task

One or more custom tasks can be defined in the `<tasks>` block. Individual tasks are defined in `<task>` child blocks. The base `<task>` class represents an extension-based workflow task that serves as a template from which ACT creates real instances.

In each task definition, you must include the attributes `name` and `version`. The following figure maps the task **DataSquares** to the Workbench UI.



Child blocks are used to specify callbacks, inputs and outputs, properties, parameters, context menus, and remote execution. The basic structure of a task definition follows:

```

<tasks>
  <task name="" caption="" icon="" version="1" enableGenericTransfer="">
    <callbacks>
      <IsVisible></IsVisible>
      <IsValid></IsValid>
      <GetValue></GetValue>
      <SetValue></SetValue>
      <OnShow></OnShow>
      <OnHide></OnHide>
      <OnInit></OnInit>
      <OnAdd></OnAdd>
      <OnRemove></OnRemove>
      <OnDuplicate></OnDuplicate>
      <OnActivate></OnActivate>
      <OnCancel></OnCancel>
      <OnApply></OnApply>
      <Value2String></Value2String>
      <String2Value></String2Value>
      <OnValidate></OnValidate>
      <OnMigrate></OnMigrate>
    </callbacks>
    <contextmenus>
      <entry name="" caption="" icon="" priority="" type="">
        <callbacks>
          <onclick></onclick>
        </callbacks>
      </entry>
    </contextmenus>
    <propertygroup name="" caption="">
      <property name="" caption="" control="" default="" options="" readonly="" needupdate="" visible="" persistent="" isparameter="" keytype="" valuetype="" elementtype="" />
    </propertygroup>
  </task>
</tasks>

```

```

<property name="" caption="" control="" default="" readonly="" needupdate="" visible=""
persistent="" isparameter="" keytype="" valuetype="" elementtype="">
  <callbacks>
    <IsVisible></IsVisible>
    <IsValid></IsValid>
    <GetValue></GetValue>
    <SetValue></SetValue>
    <OnShow></OnShow>
    <OnHide></OnHide>
    <OnInit></OnInit>
    <OnAdd></OnAdd>
    <OnRemove></OnRemove>
    <OnDuplicate></OnDuplicate>
    <OnActivate></OnActivate>
    <OnCancel></OnCancel>
    <OnApply></OnApply>
    <Value2String></Value2String>
    <String2Value></String2Value>
    <OnValidate></OnValidate>
    <OnMigrate></OnMigrate>
  </callbacks>
</property>
<parameters>
  <parameter name="" caption="" usage="" control="" version="1"/>
</parameters>
<inputs>
  <input/>
  <input type="" count="" format="" />
</inputs>
<outputs>
  <output type="" format="" />
</outputs>
<rsmjob name="" deletefiles="" version="1">
  <inputfile id="1" name="" />
  <outputfile id="1" name="" />
<program>
  <platform name="" path="" />
  <argument name="" value="" separator="" />
</program>
<callbacks>
  <oncreatejobinput></oncreatejobinput>
  <onjobstatus></onjobstatus>
  <onjobcancellation></onjobcancellation>
  <onjobreconnect></onjobreconnect>
</callbacks>
</rsmjob>
</task>
</tasks>

```

Defining Task-Level Attributes

All attributes set on ACT-defined tasks are persisted and resumed within a project.

```

...
task = ExtAPI.DataModel.Tasks[0]
task.SetAttributeValue('MyAttribute', 'MyValue')
Save(FilePath='...', Overwrite=True)
Reset()
Open(FilePath='...')
task = ExtAPI.DataModel.Tasks[0]
task.GetAttributeValue('MyAttribute')

```

Defining Task-Level General Data Transfer

ACT provides you with a method of transferring data within the **Project Schematic**. All tasks contain a general transfer object of type **GeneralTransfer**. This object contains one property, **TransferData**, of type **Dictionary<string, object>**. When the type is fully created for each task, ACT

automatically exposes the type as an output by default. If you do not want to engage in all transfer possibilities enabled through the general transfer, you can declare this at the task level:

```
<workflow name="MyWorkflow" context="Project" version="1">
  <tasks>
    <task name="MyTask" ... enableGenericTransfer="False">
      ...
    </task>
    ...
  </tasks>
</workflow>
```

Once a general transfer-enabled task exists in the schematic, the transfer data can be accessed to push and pull data within the *data store*. A simple property **TransferData** is exposed off of the task argument of all task callbacks or tasks retrieved from **ExtAPI** or as a callback argument:

```
def producer_update(task):
    data = task.TransferData
```

Once obtained, the dictionary **TransferData** acts like any other dictionary within ACT. You can both set and retrieve values through string *keys*, and you can perform other collection-based calls such as **Add()** and **Remove()**. The custom **IDictionary** is coded in such a way that if you try to access a non-existent key, such as when setting a new value, it automatically adds the key and allows the set to take place.

```
task.TransferData["Test"] = "Sample text"
```

On the consumer side, you can easily access the transfer data:

```
def consumer_update(task):
    container = task.InternalObject
    upstreamData = container.GetInputDataByType(
        InputType="GeneralTransfer")
    for upstreamDataEntity in upstreamData:
        task = ACT.GetTaskForContainer(
            upstreamDataEntity.Container)
        data = task.TransferData["Test"]
```

The transfer data approach eliminates dependency on file and file path transfers. However, if desired, you can still opt to pass file paths via the new transfer data. No further actions or implementations are required to fulfill the basic framework transfer connections.

Note

Downstream processing might require additional actions implemented via ACT workflow callbacks.

In some circumstances, you might want to interrogate a task's source tasks. For example, this can ease the access of transfer data from upstream sources. The property **SourceTasks** is available on all objects of the type **UserTask**:

```
def myTaskUpdate(task):
    sources = task.SourceTasks
    for sourceTask in sources:
        data = sourceTask.TransferData["Test"]
```

Tasks can filter or block their source connections by implementing a task-level callback **canconsumedata**.

XML Definition File:

```

<task name="Consumer">...
  <callbacks>
    <canconsumedata>taskCanUse</canconsumedata>
    ...
  </callbacks>
  ...
</task>

```

IronPython Script:

```

def taskCanUse(task, sourceTask):
    if sourceTask.Name == "Foo":
        return True
    else:
        return False

```

For a list of pre-installed tasks that support general data transfer, see [Appendix E: Pre-Installed Custom Workflow Support \(p. 497\)](#).

Custom solver-based systems also support general data transfer.

Defining Task-Level Data Transfer Access

The properties **InputData** and **OutputData** are exposed on the object **UserTask**. Additionally, **UserTask** exposes a property **TargetTasks** that returns all downstream tasks that consume (hold a connection to) the current task.

Class	Member
UserTask	InputData
	OutputData
	TargetTasks

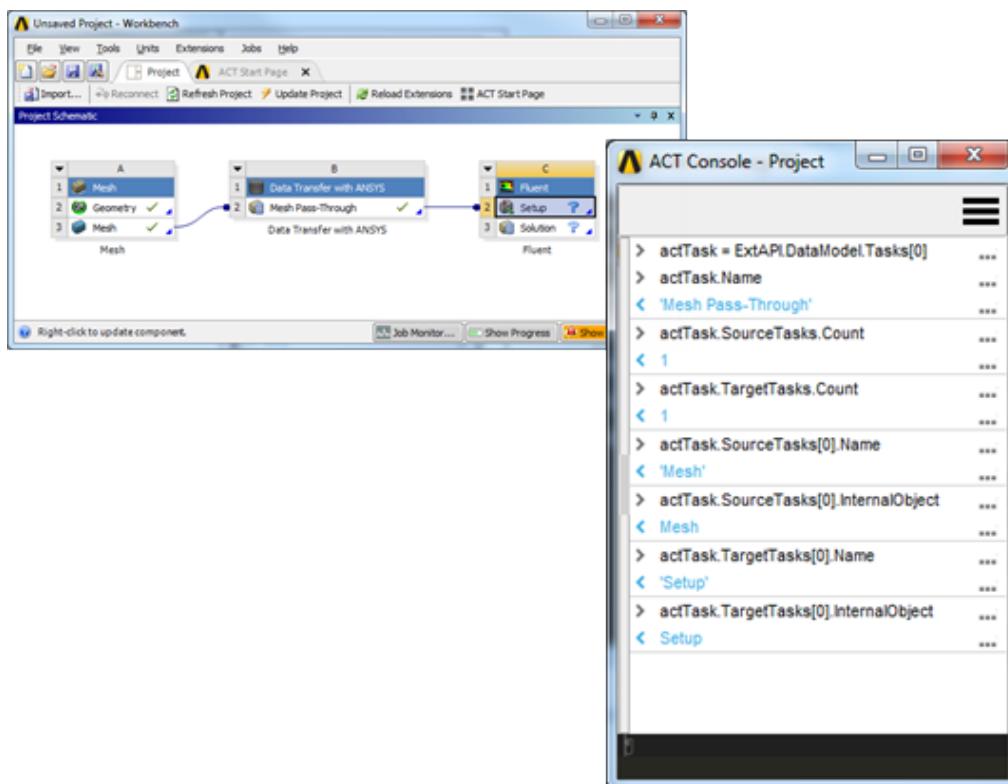
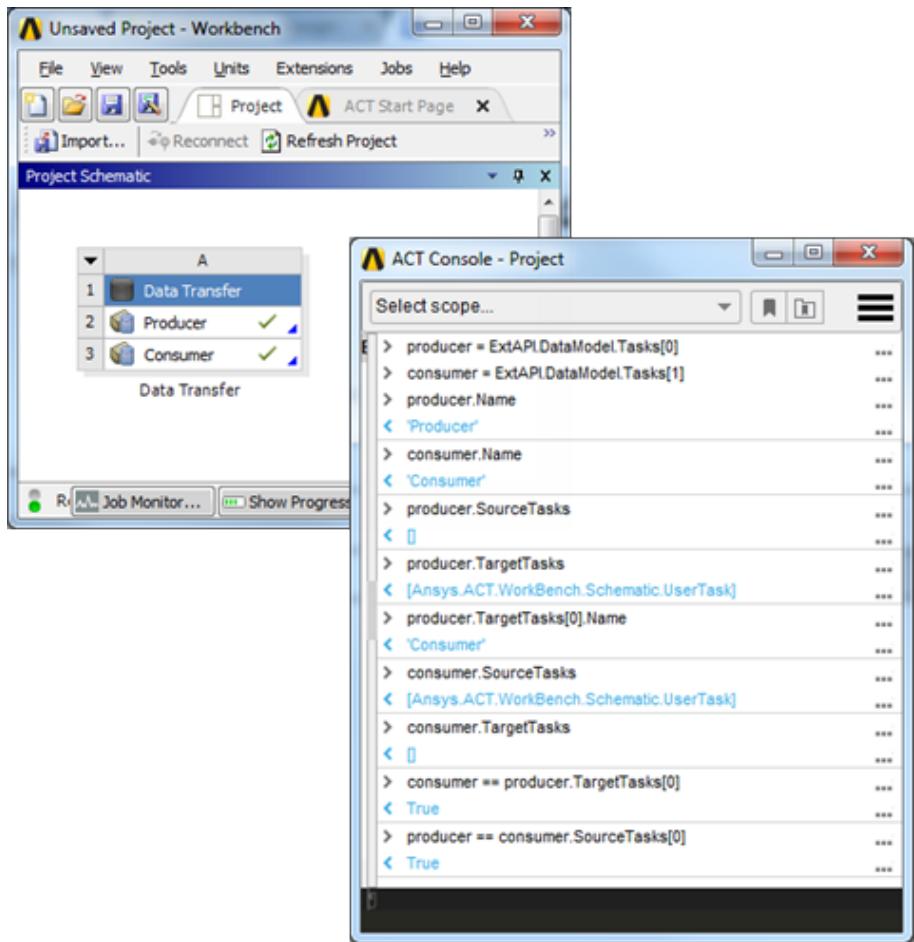
An example follows.

```

import System

def consumer_update(task):
    upstreamData = task.InputData["MyData"]
    fileRef = None
    upstreamDataCount = upstreamData.Count
    if upstreamDataCount > 0:
        fileRef = upstreamData[0]
        task.UnregisterFile(fileRef)
        task.RegisterFile(fileRef.Location)
def producer_update(task):
    extensionDir = task.Extension.InstallDir
    activeDir = task.ActiveDirectory
    filePath = System.IO.Path.Combine(extensionDir, "Sample_Materials.xml")
    activeFilePath = System.IO.Path.Combine(activeDir,
                                             "Sample_Materials.xml")
    task.UnregisterFile(activeFilePath)
    if System.IO.File.Exists(activeFilePath) == False:
        System.IO.File.Copy(filePath, activeFilePath)
    fileRef = task.RegisterFile(activeFilePath)
    myData = task.OutputData["MyData"][0]
    myData.TransferFile = fileRef
def mesh_update(task):
    upstreamData = task.InputData["MeshingMesh"]
    meshFileRef = None
    upstreamDataCount = upstreamData.Count
    if upstreamDataCount > 0:
        meshFileRef = upstreamData[0]
    meshOutput = task.OutputData["SimulationGeneratedMesh"][0]
    meshOutput.TransferFile = meshFileRef

```



Task-Level File Management Capabilities

The class **UserTask** includes APIs to facilitate easier management of task-level files. The following APIs are available:

Class	Member	Description
UserTask	ActiveDirectory	Obtains the active design point directory for the task.
	RegisteredFiles	A collection of all registered files associated with the task.
	RegisterFile(file-path)	Registers and associates a file with the task.
	UnregisterFile(file-Path)	Unregisters and disassociates a file from the task.
	UnregisterFile(file-Path, delete)	Unregisters and disassociates a file from the task.
	Unregister-File(fileReference)	Unregisters and disassociates a file from the task.
	Unregister-File(fileReference, delete)	Unregisters and disassociates a file from the task.

A sample implementation of task-level file management capabilities follows.

```
import System

def update(task):
    activeDir = task.ActiveDirectory
    installDir = task.Extension.InstallDir
    srcImageFilePath = System.IO.Path.Combine(installDir,
                                              System.IO.Path.Combine("images", "logo-ansys.jpg"))
    destImageFilePath = System.IO.Path.Combine(activeDir, "logo-ansys.jpg")
    task.UnregisterFile(destImageFilePath)
    System.IO.File.Copy(srcImageFilePath, destImageFilePath)
    task.RegisterFile(destImageFilePath)
    ExtAPI.Log.WriteMessage('task "'+task.Name+'" Registered File Count =
                           '+str(task.RegisteredFiles.Count))
```

Note

The **UnRegister(...)** members fully unregister the files only if they have been associated with the task (through the **task.Register(...)** members). Otherwise, the files remain registered.

Defining Task-Level Callbacks

Callbacks to IronPython functions are specified in a **<callbacks>** block. All callbacks receive, at minimum, a task object as an argument. The following table lists the available callbacks and their arguments.

Callback	Arguments
oninitialize	task
onupdate	task

Callback	Arguments
onrefresh	task
onreset	task
onedit*	task
onstatus**	task
onreport***	task, report
onDelete	task
canconsumedata	sourceTask, targetTask

*Definition of the callback **<onedit>** automatically creates a default **Edit** context menu for the task.

The callback **<onstatus> invokes the method **status** when the application asks the task for its current state. For more information, see [Accessing State-Handling APIs \(p. 184\)](#).

***The callback **<onreport>** invokes the method **report** when you generate a project report. It allows the task to access the report object and add its own task-specific reporting content. For more information, see [Accessing Project Reporting APIs \(p. 187\)](#).

In the event of task-level callback errors, exceptions are provided to the Workbench framework, allowing the framework to detect failures and correctly manage task states. Exception details are also reported to the Log.

Defining Task-Level Context Menus

If you specify the callback **<onedit>** in the XML definition file, a default **Edit** context menu option is automatically created for the task. However, it is possible to define additional GUI operations for each task.

Custom GUI operations are specified in the optional **<contextmenus>** block. At minimum, the context menu entry definition must include the attribute **name**. When the optional attribute **type** is defined, it must be set to **ContextMenuEntry**.

Each entry in the **<contextmenus>** block must have a callback **<onclick>**, which receives a task object as its argument.

The basic structure of the **<contextmenus>** block follows.

```
<contextmenus>
  <entry name="" caption="" icon="" priority="" type="">
    <callbacks>
      <onclick></onclick>
    </callbacks>
  </entry>
</contextmenus>
```

Defining Task-Level Property Groups and Properties

Instead of using parameters to drive your simulation, you can work through the data model, simplifying data access by defining custom properties.

Properties can be defined in the optional **<propertygroup>** block. At minimum, each property group definition must include the attribute **name**. The attribute **caption** is used as the group category name in the Workbench **Property** view if the child properties are visible.

Individual properties can be defined in the `<property>` block, which can be either a child to a `<propertygroup>` block or a standalone definition at the same level. At minimum, each property definition must include the attributes `name` and `control`.

The basic structure of a property definition follows.

```
<propertygroup name="" caption="">
  <property name="" caption="" control="" default="" readonly="" needupdate=""
    visible="" persistent="" isparameter="" keytype="" valuetype="" elementtype="" />
</propertygroup>
<property name="" caption="" control="" default="" readonly="" needupdate=""
  visible="" persistent="" isparameter="" keytype="" valuetype="" elementtype="" />
```

ACT supports the following property control types:

- `string/text`
- `double`
- `float`
- `integer`
- `DataReference`
- `boolean`
- `object`
- `quantity`
- `option/select`
- `fileopen`
- `folderopen`
- `list`
- `dictionary`
- `DataContainerReference`

Defining Task-Level Property Callbacks

Callbacks to IronPython functions can be specified in a task-level property definition. All property callbacks receive, at minimum, the entity (task) and property as arguments. Available property callbacks are listed and described.

`IsVisible`

You can define the callback `isvisible` on task-exposed properties. This callback can dynamically control whether or not to display a property. You can key off of any desired condition or other task property to determine the property's visibility, returning `True` or `False` as a result.

XML Definition File:

```
...
<task ...>
```

```

...
<property name="Input" caption="Input" control="double" .../>
<property name="HiddenString" caption="Test String"
          control="string" ...>
    <callbacks>
        <isvisible>stringVisibilityCheck</isvisible>
    </callbacks>
</property>
...
</task>
...

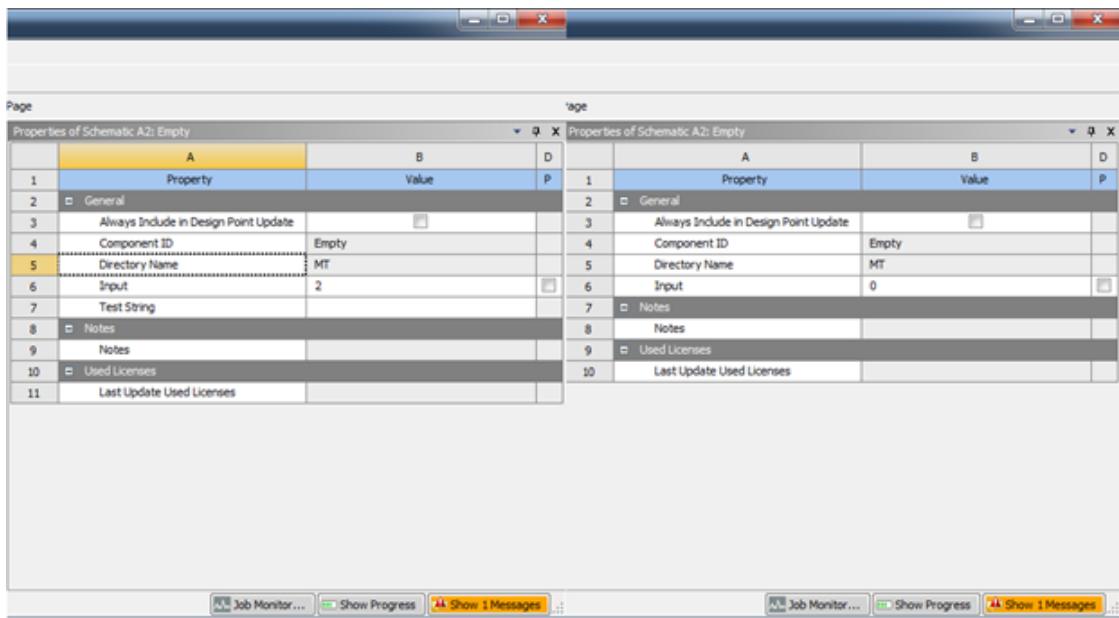
```

IronPython Script:

```

def stringVisibilityCheck(entity, property):
    inputVal = entity.Properties["Input"]
    if inputVal.Value == 2.0:
        return True
    else:
        return False

```



IsValid

You can define the callback **isValid** on task-exposed properties. This callback can dynamically control whether or not a property is valid. You can key off of any desired condition or other task property to determine the property's validity, returning **True** or **False** as a result.

XML Definition File:

```

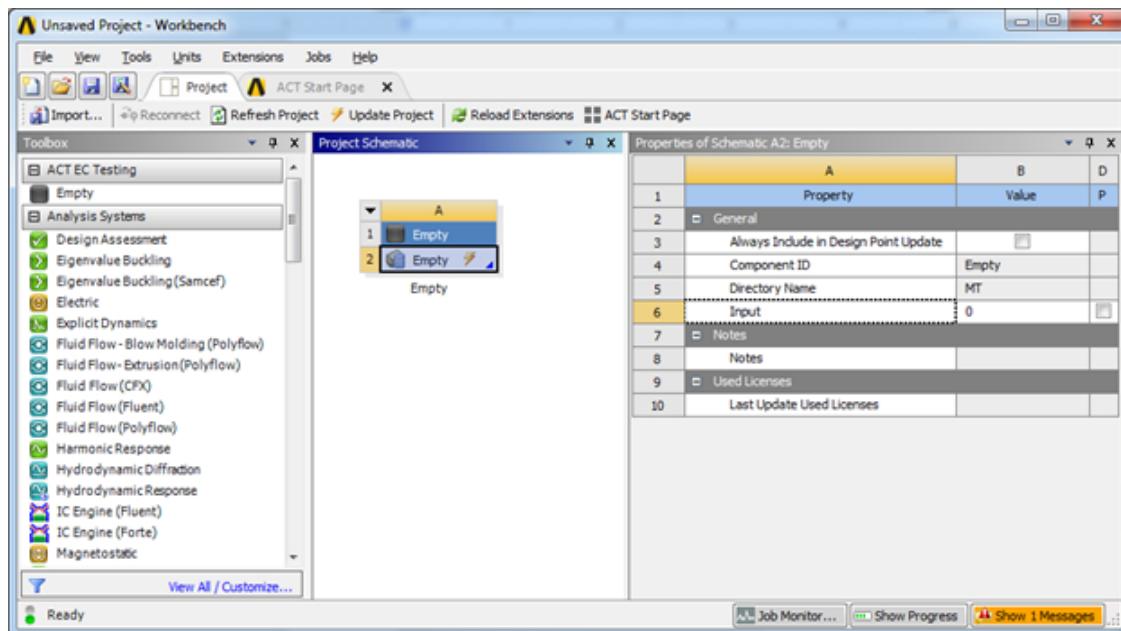
...
<task ...>
...
<property name="Input" caption="Input" control="double"
          default="0.0" readonly="False" needupdate="True"
          visible="True" persistent="True"
          isparameter="True">
    <callbacks>
        <isValid>validityCheck</isValid>
    </callbacks>
</property>
...
</task>
...

```

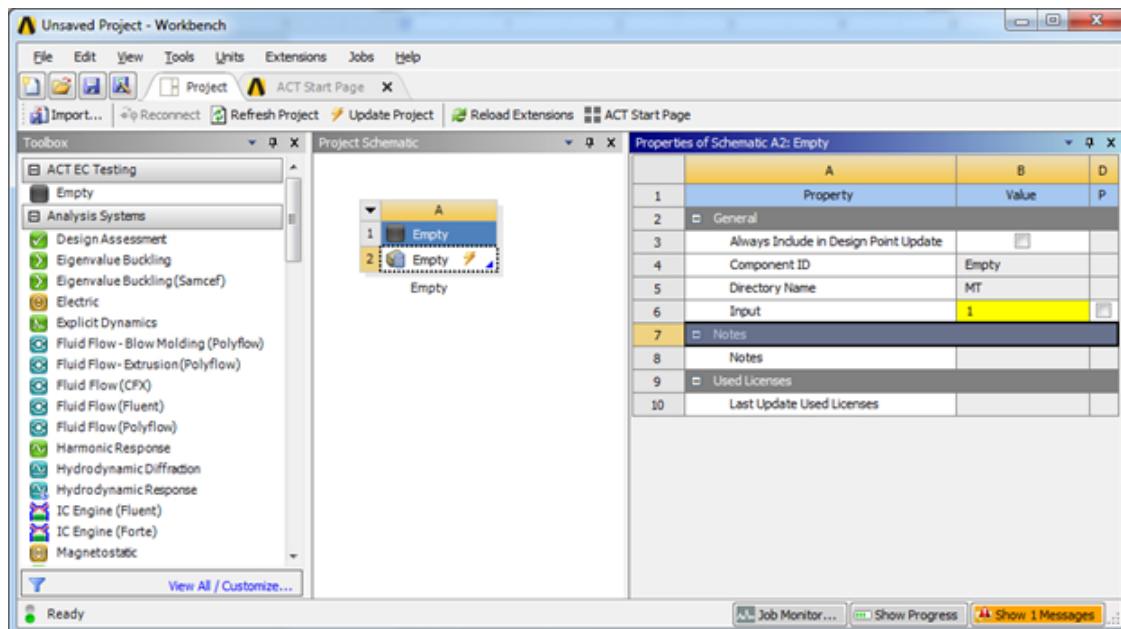
IronPython Script:

```
def validityCheck(entity, property):
    if property.Value == 1.0:
        return False
    else:
        return True
```

In the following figure, the property is valid.



In the following figure, the property is invalid.



GetValue

You can define the callback **getvalue**, which is invoked when the user obtains a value either from the command line, a script, or through the project itself.

SetValue

You can define the callback **setvalue**, which is called when the user sets a value from either the command line, a script, or through the project itself.

OnShow

You can define the callback **onshow**, which is called when the **Project** page is about to display the property.

OnHide

You can define the callback **onhide**, which is called when the **Project** page is about to hide the property.

OnInit

You can define the callback **oninit**, which is called when the property is first created on the task object.

****This is called at the same location as the callback **onadd**.****

OnAdd

You can define the callback **onadd**, which is called when the property is first added on the task object.

****This is called at the same location as the callback **oninit**.****

OnRemove

You can define the callback **onremove**, which is called when the property is deleted from the task object.

OnDuplicate

You can define the callback **onduplicate**, which is called when the property is duplicated to another task.

OnActivate

You can define the callback **onactivate**, which is called when the property is activated in the user interface.

****This is called at the same location as the callback **onshow**.****

OnCancel

You can define the callback **oncancel**, which is called when a property is deactivated.

****This is called at the same location as the callback **onhide**.****

OnApply

You can define the callback **onapply**, which is called immediately after a property value change but before it is actually processed.

Value2String

You can define the callback **value2string**, which is called when the property **ValueString** is retrieved.

String2Value

You can define the callback **string2value**, which is called when the property **ValueString** is set.

OnValidate

You can define the callback **onvalidate**, which is called during a property value change.

OnMigrate

As of 17.1, this callback is not supported.

Using Standardized Property Interaction

You can get and set property values using standard ACT syntax, as defined through **SimEntity**. At the task-level callbacks, the provided task argument provides the entry point:

```
propA = task.Properties["PropertyA"].Value
task.Properties["PropertyA"].Value = "Foo"
```

Access is available at the master **ExtAPI** level. Assuming that only one task resides in the schematic:

```
task = ExtAPI.DataModel.Tasks[0]
propA = task.Properties["PropertyA"].Value
task.Properties["PropertyA"].Value = "Foo"
```

Property groups are treated the same as elsewhere in ACT:

```
task = ExtAPI.DataModel.Tasks[0]
group1 = task.Properties["GroupName"]
subProp1 = group1.Properties["SubPropertyA"]
subProp1Val = subProp1.Value
```

When you must perform "advanced" schematic interactions, you can access **task.InternalObject** to receive the task's underlying **DataContainerReference**. This might be needed to execute Workbench-specific scripting commands. For more information, see the *Workbench Journaling and Scripting Guide*.

```
task = ExtAPI.DataModel.Tasks[0]
container = task.InternalObject
container.SendCommand(Language="Python", Command="print 'test'")
```

Interaction with pre-installed tasks operates in the same way. For example, if you create a Fluent analysis system in the schematic, you could set the property **RunParallel** as follows:

```
task = ExtAPI.DataModel.Task[0]
task.Properties["RunParallel"].Value = True
```

All "set"-based actions performed via **Property.Value** calls are recorded in the journal. Using the last example with a Fluent system, the resulting journal would contain the following code:

```
# encoding: utf-8
# Release 17.1
SetScriptVersion(Version="17.1.21")
Reset()
template1 = GetTemplateTemplateName="FLUENT")
system1 = template1.CreateSystem()
setup1 = system1.GetFluentLauncherSettings()
fluentLauncherSettings1.RunParallel = True
```

Using Path-Based Property Control Support

You can specify the controls **fileopen** and **folderopen** on task properties. On task creation and property display, a folder icon appears in the property value cell. Selecting this icon displays a dialog box for selecting a file or folder. File path and folder path access remains the same as if the property was string-based.

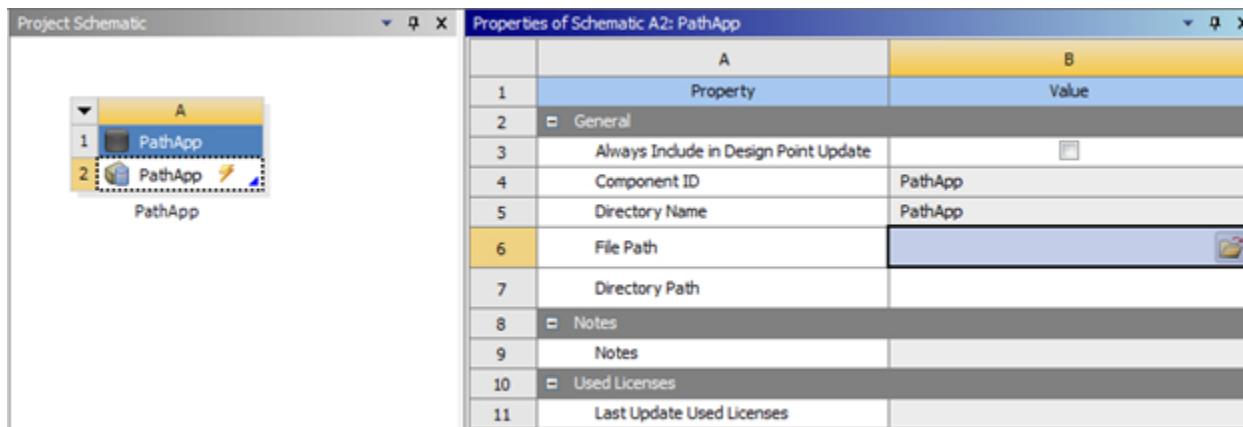
XML Definition File:

```
<task ...>
  ...
  <property name="Prop1" caption="File Path" control="fileopen"
  .../>
  <property name="Prop2" caption="Directory Path"
  control="folderopen" .../>
```

```
...
```

IronPython Script:

```
def update(task):
    filePath = task.Properties["Prop1"].Value
    dirPath = task.Properties["Prop2"].Value
```



Using String-Based and Option-Based Task Properties

For string-based task properties, specify **text** as the **control** type.

```
...
<task ...>
    ...
        <property name="StringTest" caption="String" control="text" ...>
    </property>
    ...
</task>
...
```

For option-based task properties, specify the standard **select** control type and use the **options** attribute to provide the items to be populated to the control.

```
...
<task ...>
    ...
        <property name="ComboBoxTest" caption="My Prop" control="select" ...>
    ...
<attributes options="one,two,three" ...>
    ...
</task>
...
```

Specifying Property Default Values

In the XML definition file, you can specify default property values to be used in the **Project Schematic**.

```
...
<task ...>
    ...
        <property name="MyProperty" caption="My Property" control="text" default="Foo" ...>
    ...
</task>
...
```

Defining Task-Level Parameters

You can integrate an external application with the **Project Schematic** by defining parameters that are created dynamically at initialization.

Parameters are defined in the optional `<parameters>` block. At minimum, each parameter definition must include the following attributes: `name`, `usage`, `control`, and `version`.

The basic structure of a parameter definition follows.

```
<parameters>
  <parameter name="" caption="" usage="" control="" version="1"/>
</parameters>
```

Defining Task-Level Inputs and Outputs

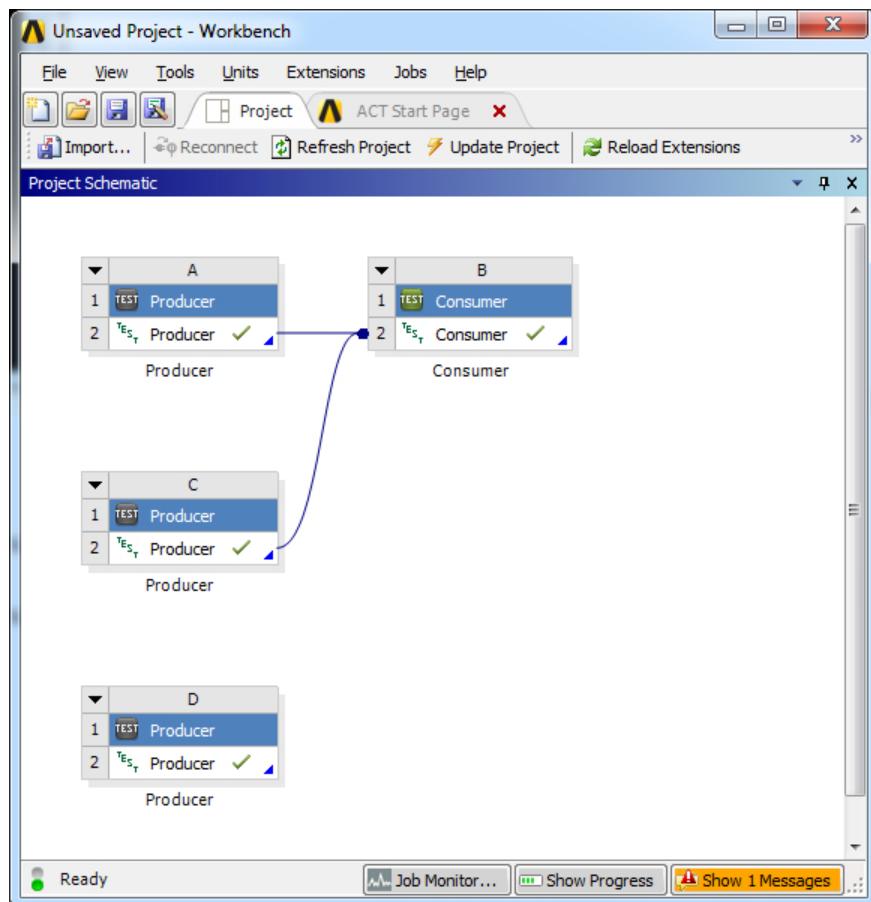
In a Workbench workflow, the **Project Schematic** connections serve as visual representations of data flow between tasks. These connections depend on input and output coordination. Workbench can establish connections only if an upstream (providing) task exposes outputs whose types also match the inputs for a downstream (consuming) task.

Inputs and outputs are defined in `<input>` and `<output>` blocks.

You can use the attribute `type` to specify the data type expected by the connection.

By default, ACT applies an unlimited count to the number of input specifications. You can specify a count according to your own requirements:

```
...
<task ...>
  ...
  <inputs>
    <input/>
    <input format="" type="GeneralTransfer" count=2/>
  </inputs>
  ...
</task>
...
```



Certain Workbench types require the use of both the attributes **type** and **format**. For example, a **Mesh** task that consumes a mesh and then passes a mesh to another task group would use these attributes to specify the mesh type and file format of the input and output files.

This example defines the inputs and outputs for a task within a Fluent meshing workflow. The meshing-based type values and **FluentMesh** format values instruct an upstream mesh task to output the Fluent mesh file format (MSH).

```
...
<task name="Mesher" caption="Mesher" icon="GenericMesh_cell" version="1">
  <callbacks>
    <onupdate>update</onupdate>
    <onedit>edit</onedit>
  </callbacks>
  <inputs>
    <input format="FluentMesh" type="MeshingMesh" count="1"/>
    <input/>
  </inputs>
  <outputs>
    <output format="FluentMesh" type="SimulationGeneratedMesh" />
  </outputs>
</task>
...
```

For a list of supported transfer types and their corresponding transfer properties, see [Appendix C: Data Transfer Types \(p. 489\)](#).

For a list of the data types and data transfer formats for each addin, see [Appendix D: Addin Data Types and Data Transfer Formats \(p. 493\)](#).

Defining a Remote Job Execution Specification

ANSYS Remote Solve Manager (RSM) enables the remote execution of applications and processes. By leveraging high-performance clusters and efficient job scheduling, users submit jobs either as background solve tasks (local machine) or as true remote cluster and scheduler submissions. Workbench orchestrates the packaging, submission, tracking, and reintegration of job artifacts.

ACT enables you to leverage RSM capabilities, specifying the remote execution of a task's processes via an `<rsmJob>` child block. This block allows you to specify all relevant RSM-related information, including supported platforms, input and output files, process arguments, and the callbacks needed to execute the remote jobs.

This example shows the basic definition of a remote execution specification. It defines the name and version of the job, specifies that job files should be deleted after execution, and defines job input and output files and callbacks.

```
...
<task name="MyTask" caption="My Task" icon="my_task" version="1">
...
  <rsmjob name="squares" deletefiles="True" version="1">
    <inputfile id="1" name="input1.txt"/>
    <outputfile id="1" name="output1.txt"/>
    <program>
      <platform name="Win64" path="%AWP_ROOT171%\path_to.exe"/>
      <platform name="Linux64" path="%AWP_ROOT171%\path_to.exe"/>
      <argument name="" value="inputFile:1" separator="/" />
      <argument name="" value="outputFile:1" separator="/" />
    </program>
    <callbacks>
      <oncreatejobinput>createJobInput</oncreatejobinput>
      <onjobstatus>getJobStatus</onjobstatus>
      <onjobcancellation>cancelJob</onjobcancellation>
      <onjobreconnect>reconnectJob</onjobreconnect>
    </callbacks>
  </rsmjob>
</task>
...
```

For a more detailed discussion of how to access RSM capabilities, see the example [Custom, Lightweight, External Application Integration with Custom Data and Remote Job Execution \(p. 291\)](#).

Accessing and Invoking Task-Related Workbench Data

From a task, you can access and manipulate objects above the task level, such as container-level commands, Workbench-level components or parameters, and the task-containing task group.

Executing Container-Level Commands on a Task

ACT provides the ability to invoke journaling and scripting-defined, container-level commands on a task. The following API is available:

Class	Member	Description
UserTask	<code>ExecuteCommand(commandName, args)</code>	Invokes a schematic method defined on the task.

The following code sample shows an implementation of the method `ExecuteCommand`.

```
#assume the first task is a mechanical-based system
task = ExtAPI.DataModel.Tasks[0]
task.ExecuteCommand("Edit", [])
```

Accessing Workbench Parameters from a Task

ACT enables you to access task-defined Workbench parameters directly from a task. The following API is available:

Class	Member	Description
UserTask	Parameters	Obtains the parameters defined on this task.

The following code sample shows an implementation of the method **Parameters**.

```
task = ExtAPI.DataModel.Tasks[0]
myParameters = task.Parameters
myParam = myParameters[0]
myParamValue = myParam.Value
```

Accessing a Workbench-Based Component from a Task

ACT enables you to access the Workbench-based component object directly from a task. The following API is available:

Class	Member	Description
UserTask	Component	Obtains the underlying Project Schematic component associated with this task.

The following code sample shows an implementation of the method **Component**.

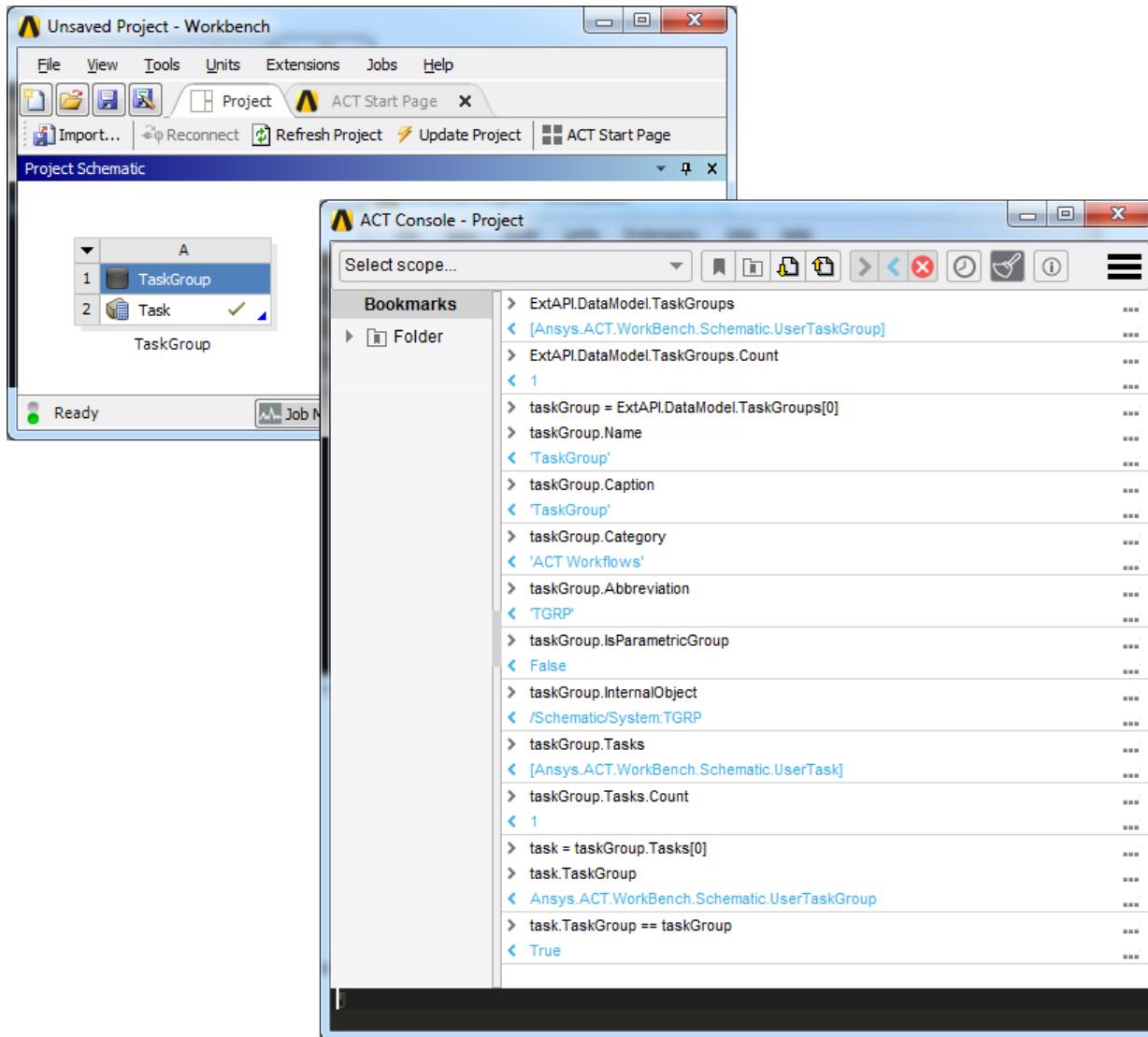
```
#assume the first task is a mechanical-based system
task = ExtAPI.DataModel.Tasks[0]
myTaskComponent = task.Component
```

Accessing the Owning Task Group from a Task

ACT enables you to access specific **taskgroup** instances that have been created within the **Project Schematic**. Because a **task** is owned by a **TaskGroup**, you can access the owning **TaskGroup** directly from a new **Task**. The following APIs are available:

Class	Member	Description
Task (UserTask)	TaskGroup (UserTaskGroup)	Obtains the task's task group owner.
ExtAPI.DataModel	TaskGroups	The list of all task groups currently within the schematic.
TaskGroup (User-TaskGroup)	Tasks (User-Tasks)	The list of all tasks owned by this task group.
	Name	The task group name.
	Caption	The task group display name.
	Category	The task group toolbox category.
	Abbreviation	The task group abbreviation.
	IsParametric-Group	Indicates whether the task group falls under the Parameter Set bar,

Class	Member	Description
		operating only on parameter and design point data.
	InternalObject	(System data reference)



Defining a Task Group

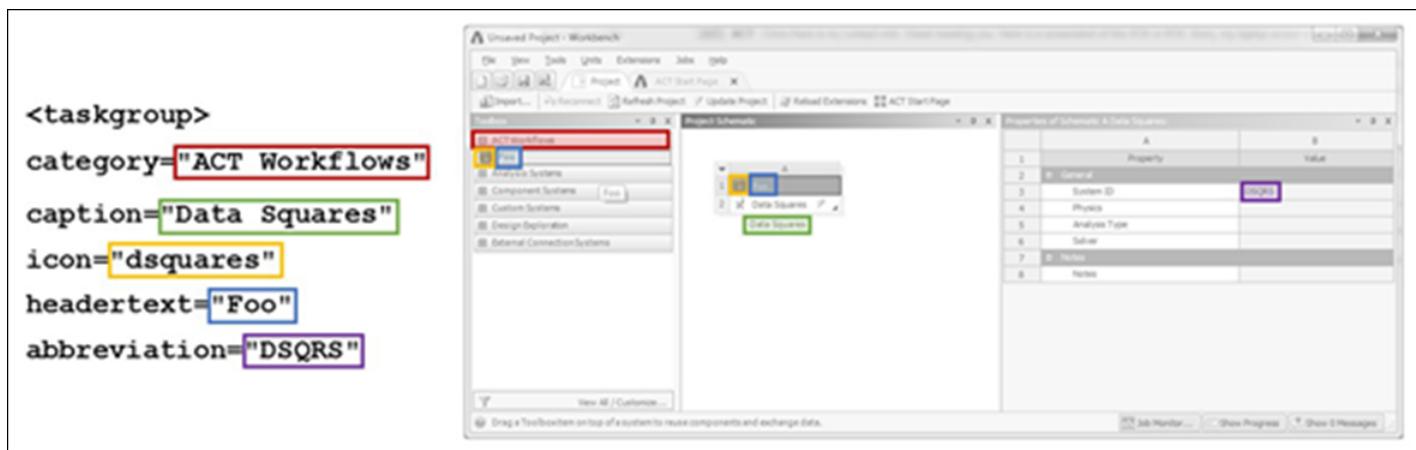
A *task group* is a collection of tasks defined in a `<taskgroups>` block. Individual task groups are defined in child `<taskgroup>` blocks. At minimum, each task group definition must include the attributes **name** and **version**.

```

<extension>
  <workflow name="MyWorkflow" context="Project" version="1">
    <taskgroups>
      <taskgroup name="DataSquares" ... headertext="Foo" />
    </taskgroups>
  </workflow>
</extension>

```

The following figure maps the preceding example to the Workbench UI.



Starting in ANSYS 18.0, task group entries support standard ACT attributes. When creating custom task groups with installed ANSYS-defined mechanical tasks, you can use attributes to specify additional properties.

```
<extension>
  <workflow name="MyWorkflow" context="Project" version="1">
    <taskgroups>
      <taskgroup name="CustomStructural" caption="Custom Structural Group"
                 icon="custom_structural" category="ACT Workflows"
                 abbreviation="CSTRUCT" version="1">
        ...
        <attributes SolverType="ANSYS"
                   SystemName="Static Structural (ANSYS)"/>
      </taskgroup>
    </taskgroups>
  </workflow>
</extension>
```

For the following tasks, you find the solver type by referring to `$(AttributeCombination)`, where the second part of the Physics Type and Solver Name combination is the Solver Type.

- `SimulationModelCellTemplate`
- `SimulationSetupCellTemplate_$(AttributeCombination)`
- `SimulationSolutionCellTemplate_$(AttributeCombination)`
- `SimulationResultsCellTemplate_$(AttributeCombination)`

For a static structural ANSYS-based task, the `StructuralStaticANSYS` combination shows `ANSYS` as the Solver Type. You can obtain more combinations from `%AWP_ROOT182%\Addins\Simulation\AnalysisTemplates.xml`.

You use the `<includetask>` and `<includeGroup>` child blocks to specify the one or more tasks or nested task groups to include. In the `<includetask>` child block, you can reference both the custom tasks defined in your extension and "external" tasks (Workbench-defined components). Setting the optional attribute `external` to `true` specifies that the referenced task is defined externally, originating outside of the extension.

For an external task, the attribute `name` must be set to the internally defined template name for the given component. Optionally, you can set the attribute `caption` to use your own display text. For a list of internally defined systems with their components and component display names, see [Appendix B: ANSYS Workbench Internally Defined System Template and Component Names \(p. 471\)](#). To

review examples of task groups referencing external tasks, see [Generic Material Transfer \(p. 298\)](#) or [Generic Mesh Transfer \(p. 302\)](#).

If you define a caption, it overrides the task-level caption. Otherwise, the task group-level caption is used.

The basic structure of a task group definition follows.

```
...
<taskgroups>
  <taskgroup name="" caption="" icon="" category="" abbreviation="" version="1"
  isparametricgroup="False">
    <includetask name="" external="" />
    <includeGroup name="" />
  </taskgroup>
</taskgroups>
...
```

Note

At present, ACT-defined tasks cannot be added to installed task groups. Also, nested task groups within a task group are not recognized by Workbench at 17.1.

Creating the IronPython Script

IronPython scripts define the actions to be executed during a callback invocation. These scripts execute within the Workbench Python interpreter. As a result, scripts have full access to the scriptable Workbench API. For more information, see [Using Journals and Scripts in the Workbench User's Guide](#).

For example, the script is used to create update instructions for producing and consuming data. If any task produces or consumes data, you must supply an **Update** routine that processes input and output types as declared by the workflow definition. For a table of supported task inputs and outputs, see [Appendix A: Component Input and Output Tables \(p. 411\)](#).

Related Topics:

- [Upstream Data Consumption \(Input\)](#)
- [Data Generation \(Output\)](#)
- [Convenience APIs](#)

Upstream Data Consumption (Input)

Typically, tasks need to implement complex source handling logic, connection tracking routines, and a refresh procedure to consume data. ACT abstracts these complexities by automatically performing these actions during refresh. It obtains upstream data and stores it in a dictionary accessible by the user during the task Update. The task can obtain this data by calling [Convenience APIs \(p. 144\)](#).

Data Generation (Output)

Tasks that produce output data (for example, declare output types in the task group definition) must ensure that their custom Update routine assigns output data.

The XML definition file prepares empty data objects representing task outputs. You must only set the correct transfer properties that downstream consumers interrogate. Refer to [Appendix C: Data Transfer Types \(p. 489\)](#) to determine which properties you must set.

For example, a material transfer to a downstream **Engineering Data** task must set the property **DataReference TransferFile** on a MatML31 data object to the file reference of a registered matml-formatted XML file, all completed during the update routine.

Convenience APIs

Convenience APIs are IronPython queries that provide simple access to task-stored input and output data. The available convenience APIs are:

GetInputDataByType

Returns a **List<object>** containing upstream data for a given type. For example:

```
upstreamData =
    container.GetInputDataByType(InputType="MeshingMesh")
meshFileRef = None
upstreamDataCount = upstreamData.Count
if upstreamDataCount > 0:
    meshFileRef = upstreamData[0]
```

GetOutputData

Returns a **Dictionary<string, List<DataReference>>** holding the task's output types. For example:

```
outputRefs = container.GetOutputData()
meshOutputSet = outputRefs["SimulationGeneratedMesh"]
meshOutput = meshOutputSet[0]
meshOutput.TransferFile = meshFileRef
```

GetCustomEntity

Obtains the custom data entity from a container based on the entity's name and type. If the name and type are not specified, the default ACT object is returned. This object contains the properties defined by the workflow task from the **<propertygroup>** and **<property>** blocks.

```
entity = ACT.GetCustomEntity(container)
```

GetCustomEntityPropertyValue

This query returns the value of a property defined on a custom data entity.

```
inputValue = ACT.GetCustomEntityPropertyValue(entity, "Input")
```

Users can also directly access properties off of a task object.

SetCustomEntityPropertyValue

This command handles the setting of a custom entity's property value.

```
ACT.SetCustomEntityPropertyValue(entity, "Output", inputValue)
```

Users can also directly set properties off of a task object.

GetTaskByName

This command accesses the data model to facilitate task searches. A usage example appears in the figure at the end of this section.

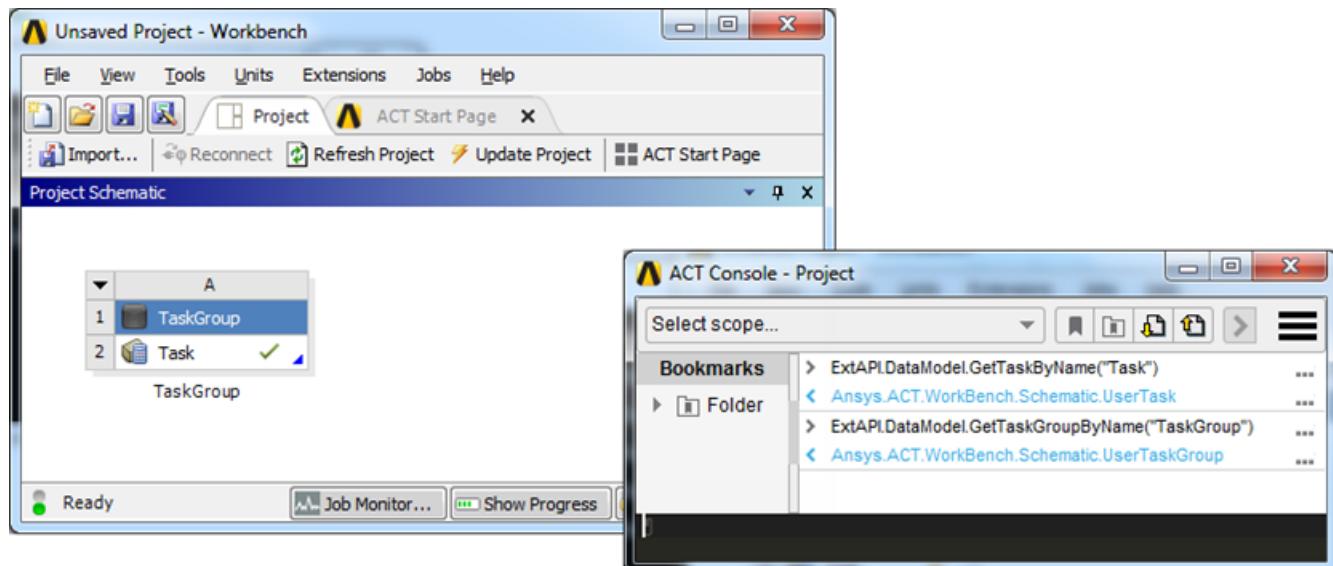
```
ExtAPI.DataModel.GetTaskByName(taskName)
```

GetTaskGroupByName

This command accesses the data model to facilitate task group searches. A usage example appears in the figure at the end of this section.

```
ExtAPI.DataModel.GetTaskGroupByName(taskGroupName)
```

The following figure shows convenience queries for tasks and task groups.



Simulation Wizards

ANSYS ACT allows you to automate the simulation process with the creation of custom simulation wizards. A simulation wizard is part of an ACT extension. In addition to accessing the functionality defined in that extension, it can leverage the API scripting capabilities available with Workbench, AIM, and all ANSYS products that support ACT wizards. ACT wizards enable users to benefit from ANSYS simulation capabilities while minimizing interactions with the standard ANSYS product interface.

Wizards can be created for Workbench, AIM, and any ANSYS product providing scripting capabilities and ACT support. This includes Mechanical, DesignModeler, SpaceClaim, Electronics Desktop, and Fluent. A wizard walks non-expert end-users step-by-step through a simulation.

The following topics are addressed:

[Types of Wizards](#)

[Creating Wizards](#)

[Installing and Loading Wizards](#)

[Using Wizards](#)

Types of Wizards

You can use ACT to create three different types of wizards: project wizards, target product wizards, and mixed wizards.

Project Wizards

A *project wizard* is executed from the **Project** tab in either Workbench or AIM.

- When launched from the **Project** tab in Workbench, a project wizard can engage any data-integrated ANSYS product with Workbench journaling and scripting capabilities (such as Mechanical, DesignXplorer, and SpaceClaim) and incorporate them into the project workflow.
- When launched from the **Project** tab in AIM, a project wizard can be used to automate an AIM simulation.

Target Product Wizards

A *target product wizard* is executed wholly in a specified ANSYS product that provides both scripting capabilities and ACT support. This includes AIM, Mechanical, DesignModeler, SpaceClaim, Electronics Desktop, and Fluent.

A target product wizard utilizes the functionality provided by the target product and provides simulation guidance within the product's workflow. Target product wizards for some ANSYS products, such as native products like Mechanical and DesignModeler, are run only within the Workbench environment. Target product wizards for other ANSYS products, such as SpaceClaim, Electronics Desktop, and Fluent, can be run either within the Workbench environment or in a stand-alone instance of the product.

The following table shows where different target product wizards can be launched

ANSYS Product	Launched in Workbench Environment	Launched in Standalone Instance of the Product
AIM	X	
Mechanical	X	
DesignModeler	X	
SpaceClaim	X	X
Electronics Desktop	X	X
Fluent	X	X

Mixed Wizard

A *mixed wizard* is launched from the Workbench **Project** tab and then engages one or more supported ANSYS products to automate the entire simulation process. A mixed wizard is used when interactive selection on the model is required at a given step in one given product. It provides native workflow guidance in both the **Project** tab and the included products.

Creating Wizards

A wizard is part of an ACT extension. To create an extension, you must have an ACT license. Once the scripted version of an extension is completed, you can generate a binary version. No license is needed to execute a binary version.

To start creating wizards, you can download template packages from the [ACT Resources](#) page on the [ANSYS Customer Portal](#) to use as examples. To display the page, select **Downloads > ACT Resources**. To download template packages, expand the **ACT Templates** section and click the links for the desired packages.

The following topics are addressed:

- [Parts of a Wizard](#)
- [XML Extension Definition File](#)
- [IronPython Script](#)
- [Custom Help for Wizards](#)
- [Creating Conditional Steps](#)
- [Customizing the Layout of a Wizard](#)

Parts of a Wizard

As part of an ACT extension, an ACT wizard makes use of the following files:

- **XML extension definition file and referenced IronPython script (required)**

These files are the same as those required for a standard ACT extension and use the same XML and IronPython syntax. Specific elements have to be used in the XML file to define the wizard. To create a wizard, you can begin with an existing extension and modify it. For syntax information, see the [ANSYS ACT XML Reference Guide](#) and the [ANSYS ACT API Reference Guide](#).

- **Custom help files (optional)**

HTML files containing text, images, charts, or other control types can be used to provide instructions or details for the wizard.

XML Extension Definition File

To create a wizard, you add a `<wizard>` block in the XML extension definition file. The existing requirements of a standard XML extension definition file apply. No limit exists on the number of wizards that can be added.

Within a `<wizard>` block, child `<step>` blocks define wizard steps. Each `<step>` block then has child blocks that define callbacks and properties for the step. All blocks have a set of required and optional attributes. For example, the `<wizard>` block requires the following attributes: `name`, `version`, and `context`.

You can use an optional `<uidefinition>` block to customize the appearance of a wizard that is not for AIM. In a `<uidefinition>` block, you can specify layouts both for the interface as a whole and for individual interface components. For more information on customizing the interface of a non-AIM wizard, see [Customizing the Layout of a Wizard \(p. 162\)](#) and [Wizard Custom Layout Example \(p. 391\)](#).

Example: XML Extension Definition File

To better understand how wizards are defined, look at the following excerpt from the XML definition file for a project wizard that is executed from the Workbench **Project** tab. In this XML file for the extension `WizardDemos`, the wizard `Project Wizard` is defined.

Note

The XML definition syntax is the same for all project wizards, regardless of whether they are run from the Workbench or AIM **Project** tab.

```

<extension version="2" minorversion="1" name="WizardDemos">
  <guid shortid="WizardDemos">7fdb141e-3383-433a-a5af-32cb19971771</guid>
  <author>ANSYS Inc.</author>
  <description>Simple extension to test wizards in different contexts.</description>

  <script src="main.py" />
  <script src="ds.py" />
  <script src="dm.py" />
  <script src="sc.py" />

  <interface context="Project|Mechanical|SpaceClaim">
    <images>images</images>
  </interface>

  <interface context="DesignModeler">
    <images>images</images>

    <toolbar name="Deck" caption="Deck">
      <entry name="Deck" icon="deck">
        <callbacks>
          <onclick>CreateDeck</onclick>
        </callbacks>
      </entry>
      <entry name="Support" icon="Support">
        <callbacks>
          <onclick>CreateSupport</onclick>
        </callbacks>
      </entry>
    </toolbar>
  </interface>
  ...

```

```

<wizard name="ProjectWizard" version="1" context="Project" icon="wizard_icon">
  <description>Simple wizard for demonstration in Project page.</description>

  <step name="Geometry" caption="Geometry" version="1" HelpFile="help/geometry.html"
    layout="DefaultTabularDataLayout">
    <description>Create a geometry component.</description>

    <componentStyle component="Submit">
      <background-color>#b6d7a8</background-color>
    </componentStyle>

    <componentData component="TabularData">
      <CanAddDelete>false</CanAddDelete>
    </componentData>

    <callbacks>
      <onupdate>EmptyAction</onupdate>
      <onreset>DeleteGeometry</onreset>
      <oninit>InitTabularData</oninit>
    </callbacks>

    <propertygroup display="caption" name="definition" caption="Basic properties" >
      <property name="filename" caption="Geometry file name" control="fileopen" />
      <property name="myint" caption="Integer value" control="integer" />
      <property name="mytext" caption="Text value" control="text" />
      <property name="myquantity" caption="Quantity value" control="float" unit="Pressure" />
      <property name="myreadonly" caption="Readonly value" control="text" readonly="true"
        default="My value" />
      <propertygroup display="property" name="myselect" caption="List of choice" control="select"
        default="Option1">
        <attributes options="Option1,Option2" />
        <property name="option1" caption="Option1 value" control="text" visibleon="Option1" />
        <property name="option2first" caption="Option2 first value" control="float" unit="Pressure"
          visibleon="Option2" />
        <property name="option2second" caption="Option2 second value" control="float" unit="Length"
          visibleon="Option2" />
      </propertygroup>
    </propertygroup>

    <propertytable name="Table" caption="Table" control="custom" display="worksheet"
      class="Worksheet.TabularDataEditor.TabularDataEditor">
      <property name="Frequency" caption="Frequency" unit="Frequency" control="float"
        isparameter="true"></property>
      <property name="Damping" caption="Damping" control="float" isparameter="true"></property>
      <property name="TestFileopen" caption="fileopen" control="fileopen"></property>
    </propertytable>

    <propertytable name="TableB" caption="Table B" control="tabulardata" display="worksheet">
      <property name="Integer" caption="Integer" control="integer"></property>
      <property name="FileOpen" caption="Fileopen" control="fileopen"></property>
      <property name="Number" caption="Number A" unit="Pressure" control="float"></property>
      <property name="ReadOnly" caption="ReadOnly" unit="Pressure" control="float" readonly="true"
        default="4 [Pa]" />
      <propertygroup display="property" name="myselect" caption="List of choice" control="select"
        default="Option1">
        <attributes options="Option1,Option2" />
        <property name="TestStr" caption="Text" control="text" visibleon="Option1"></property>
        <property name="Number B" caption="Number B" unit="Temperature" control="float"></property>
      </propertygroup>
    </propertytable>

  </step>

  <step name="Mechanical" caption="Mechanical" enabled="true" version="1" HelpFile="help/mechanical.html">
    <description>Create a mechanical component.</description>

    <callbacks>
      <onupdate>EmptyAction</onupdate>
      <onreset>DeleteMechanical</onreset>
    </callbacks>

    <property name="name" caption="System name" control="text" />

```

```

</step>

<step name="Fluent" caption="Fluent" version="1" HelpFile="help/fluent.html">
  <description>Create a fluent component.</description>

  <callbacks>
    <onrefresh>CreateDialog</onrefresh>
    <onupdate>EmptyAction</onupdate>
    <onreset>EmptyReset</onreset>
  </callbacks>

  <property name="name" caption="System name" control="text" />
  <property name="dialog" caption="Dialog" control="text">
    <callbacks>
      <onvalidate>ValidateDialog</onvalidate>
    </callbacks>
  </property>
  <property name="dialog2" caption="DialogProgress" control="text">
    <callbacks>
      <onvalidate>ValidateDialogProgress</onvalidate>
    </callbacks>
  </property>
  <property name="nextstep" caption="Next Step" control="select" >
    <callbacks>
      <onvalidate>ValidateNextStep</onvalidate>
    </callbacks>
  </property>
</step>

<step name="ReportView" caption="ReportView" version="1" layout="ReportView@WizardDemos">
  <description>Simple example to demonstrate how report can be displayed.</description>

  <callbacks>
    <onrefresh>RefreshReport</onrefresh>
    <onreset>EmptyReset</onreset>
  </callbacks>

</step>

<step name="CustomStep" caption="CustomStep" enabled="true" version="1" layout="MyLayout@WizardDemos">
  <description>Create a mechanical component.</description>

  <callbacks>
    <onrefresh>RefreshMechanical</onrefresh>
    <onupdate>LogReport</onupdate>
    <onreset>EmptyReset</onreset>
  </callbacks>

  <property name="name" caption="System name" control="text" />
</step>

<step name="Charts" caption="Charts" enabled="true" version="1" layout="GraphLayout@WizardDemos">
  <description>Demonstrate all chart capabilities.</description>

  <callbacks>
    <onrefresh>RefreshCharts</onrefresh>
  </callbacks>

</step>

</wizard>

...
</extension>

```

Extension definition:

The top-most `<extension>` block specifies the extension name and version.

Guid definition:

The `<guid>` block specifies a unique identifier for the extension.

Script definition:

The `<script>` blocks define the IronPython scripts that are referenced by the wizards in the extension.

Interface definition:

The `<interface>` block defines the user interface for one or more wizards in the extension. For this example, the attribute `context` for the `<interface>` block is set to **Project | Mechanical | Space-Claim**.

Wizard definition:

The `<wizard>` block specifies the required attributes `name`, `version`, and `context`. In this example, the attribute `context` is set to **Project** because this wizard is executed from the Workbench **Project** tab.

Optional attributes `icon` and `description` are also defined. Icon files are stored in the folder `images` in the extension directory. You specify the location of the folder `images` in the `<interface>` block.

Child `<step>` blocks define the wizard steps.

Note

- For a mixed wizard, the attribute `context` for the wizard is still set to **Project** because the wizard is launched from the Workbench **Project** tab. Each step in a mixed wizard has its `context` set to the ANSYS product from which the step is executed.
 - For a wizard targeted for AIM, the attribute `context` is not used for step definitions. For more information, see [Using an AIM Target Product Wizard \(p. 168\)](#).
-

Step definition:

A `<step>` block exists for each step in the wizard. In a `<step>` block, the required attributes `name` and `version` are defined. The optional attribute `caption` sets the display text for the step. You must set the attribute `context` for a step only if it differs from the attribute `context` for the wizard. All steps in this particular wizard are executed from the Workbench **Project** tab.

Note

For each step in a mixed wizard, the `context` is set to the ANSYS product from which the step is executed.

Additional optional attributes can be defined for steps. For example, the attribute `HelpFile` indicates the file containing custom help content to display in the user interface. In the IronPython script, you can also programmatically change the attribute `enabled` for a step to make its display conditional. For more information, see [Creating Conditional Steps \(p. 161\)](#).

Child blocks define callbacks and properties.

Callback definition:

A `<callbacks>` block executes a function that has been defined in the IronPython script for the wizard. Each callback receives the current step as a method argument.

- The required callback `<onupdate>` is invoked when you click the **Next** button for a Workbench wizard step. In the wizard example, the callback `<onupdate>` for the first three steps executes the function `EmptyAction`.
- The callback `<onrefresh>` is invoked when the next step is initialized. It initiates changes to the interface, so you can use this callback to change the appearance of the wizard, initialize a value to be used in the current step, instantiate variables, access data of an existing project, and so on. In the wizard example, the callback `<onrefresh>` for the step **Fluent** executes the function `CreateDialog`.
- The callback `<onreset>` resets the state of the step so that the user can remedy a situation where the step was not completed correctly. In the first two steps of the example, it executes the functions `DeleteGeometry` and `DeleteMechanical`.

Note

For wizards targeted at AIM, only the callbacks `<onupdate>`, `<onrefresh>`, and `<onreset>` are supported. For more information, see [Using an AIM Target Product Wizard \(p. 168\)](#).

Property definition:

A `<propertygroup>` block specifies groupings of properties within the step, and a `<property>` block specifies properties and property attributes. For a `<property>` block, the attributes `control` and `name` are required. The attribute `control` specifies the control type for the property. The standard property types of `select`, `float`, `integer`, `scoping`, `text`, and `fileopen` are supported. Advanced types such as `applycancel` and `custom` are also supported.

Optional attributes can also be defined for various properties.

- The attribute `caption` specifies the display text for the property.
- The attribute `unit` specifies the units for the property value.
- The attribute `readonly` specifies whether the property value can be edited.
- The attribute `default` specifies the default property value.
- The attribute `visibleon` specifies whether the property is to display. In this example, the property `visibleon` is used to control the display of conditional options in a property group.

The `<propertygroup>` block is used to create groupings of properties.

- For the first property group in the example, no control is defined, so that the group is a list of individual properties inside the `<propertygroup>` block.
- The second property group is embedded within the first.
 - The attribute `control` is set to `select`, which provides a drop-down list.
 - The attribute `options` defines the available options.

- The attribute **visibleon** indicates that the properties in the group are conditional based on selection from the drop-down.
- The callback **<onvalidate>** within a **<property>** block is used to validate the value entered for a property.

The **<propertytable>** block is used to create a table for the entry of property values in tabular format. The properties **Temperature** and **Pressure** nested inside the **<propertytable>** blocks create columns for data entry.

IronPython Script

The IronPython script is referenced by the XML extension definition file. It defines the functions that are used by the extension and the actions that are performed during each step of the wizard. The XML file can reference multiple scripts.

ANSYS journaling and scripting commands can be used inside callbacks. You can write the commands manually or edit commands obtained from a session journal file, either from the temporary journal stored in your folder **%TEMP%** or from a journal generated by using the **Record Journal** menu option.

Example: IronPython Script for a Project Wizard

To demonstrate how an IronPython script is used in a wizard, this topic uses the same **Project Wizard** extension as the previous section.

The XML extension definition file references the script `main.py`:

```
geoSystem = None
dsSystem = None
fluentSystem = None

def EmptyAction(step):
    pass

def InitTabularData(step):
    table = step.Properties["TableB"]
    for i in range(1, 10):
        table.AddRow()
        table.Properties["Integer"].Value = i
        table.Properties["FileOpen"].Value = "super"
        table.Properties["Number"].Value = 45.21
        table.Properties["ReadOnly"].Value = 777
        table.Properties["myselect"].Properties["TestStr"].Value = 777
        table.Properties["myselect"].Properties["Number B"].Value = 777
    table.SaveActiveRow()

def CreateGeometry(step):
    global geoSystem
    template1 = GetTemplate(TemplateName="Geometry")
    geoSystem = template1.CreateSystem()
    geometry1 = geoSystem.GetContainer(ComponentName="Geometry")
    geometry1.SetFile(FilePath=step.Properties["definition/filename"].Value)

def DeleteGeometry(step):
    global geoSystem
    geoSystem.Delete()

def RefreshMechanical(step):
    tree = step.UserInterface.GetComponent("Tree")
    root = tree.CreateTreeNode("Root")
    node1 = tree.CreateTreeNode("Node1")
    node2 = tree.CreateTreeNode("Node2")
    node3 = tree.CreateTreeNode("Node3")
```

```

root.Values.Add(node1)
root.Values.Add(node2)
node2.Values.Add(node1)
node2.Values.Add(node3)
root.Values.Add(node3)
tree.SetTreeRoot(root)
chart = step.UserInterface.GetComponent("Chart")
chart.Plot([1,2,3,4,5],[10,4,12,13,8],"b","Line1")
chart.Plot([1,2,3,4,5],[5,12,7,8,11],"r","Line2")

def CreateMechanical(step):
    global dsSystem, geoSystem
    template2 = GetTemplate(
        TemplateName="Static Structural",
        Solver="ANSYS")
    geometryComponent1 = geoSystem.GetComponent(Name="Geometry")
    dsSystem = template2.CreateSystem(
        ComponentsToShare=[geometryComponent1],
        Position="Right",
        RelativeTo=geoSystem)
    if step.Properties["name"].Value=="error":
        raise UserErrorMessageException("Invalid system name. Please try again.")
    dsSystem.DisplayText = step.Properties["name"].Value

def DeleteMechanical(step):
    global dsSystem
    dsSystem.Delete()

def CreateFluent(step):
    global dsSystem, fluentSystem
    template3 = GetTemplate(TemplateName="Fluid Flow")
    geometryComponent2 = dsSystem.GetComponent(Name="Geometry")
    solutionComponent1 = dsSystem.GetComponent(Name="Solution")
    componentTemplate1 = GetComponentTemplate(Name="CFDPostTemplate")
    fluentSystem = template3.CreateSystem(
        ComponentsToShare=[geometryComponent2],
        DataTransferFrom=[Set(FromComponent=solutionComponent1, TransferName=None,
        ToComponentTemplate=componentTemplate1)],
        Position="Right",
        RelativeTo=dsSystem)
    if step.Properties["name"].Value=="error":
        raise Exception("Invalid system name. Please try again.")
    fluentSystem.DisplayText = step.Properties["name"].Value

def CreateDialog(step):
    comp = step.UserInterface.Panel.CreateOKCancelDialog("MyDialog", "MyTitle", 400, 150)
    comp.SetOkButton("Ok")
    comp.SetMessage("My own message")
    comp.SetCallback(cbDialog)
    prop = step.Properties["nextstep"]
    prop.Options.Clear()
    s = step.NextStep
    val = s.Caption
    while s!=None:
        prop.Options.Add(s.Caption)
        s = s.NextStep
    prop.Value = val

def cbDialog(sender, args):
    dialog = CurrentWizard.CurrentStep.UserInterface.GetComponent("MyDialog").Hide()

def ValidateDialog(step, prop):
    dialog = step.UserInterface.GetComponent("MyDialog")
    dialog.Show()

def worker(step):
    progressDialog = step.UserInterface.GetComponent("Progress")
    progress = progressDialog.GetFirstOrDefaultComponent()
    progress.Reset()
    stopped = progress.UpdateProgress("Start progress...", 0, True)
    progressDialog.Show()
    for i in range(100):

```

```
System.Threading.Thread.Sleep(100)
stopped = progress.UpdateProgress("Start progress...", i+1, True)
if stopped:
    break
progressDialog.Hide()

def ValidateDialogProgress(step, prop):
    thread = System.Threading.Thread(System.Threading.ParameterizedThreadStart(worker))
    thread.Start(step)

def ValidateNextStep(step, prop):
    prop = step.Properties["nextstep"]
    s = step.NextStep
    v = False
    while s!=None:
        if prop.Value==s.Caption:
            v = True
        s.IsEnabled = v
        s = s.NextStep
    steps = step.UserInterface.GetComponent("Steps")
    steps.UpdateData()
    steps.Refresh()

def RefreshReport(step):
    report = step.UserInterface.GetComponent("Report")
    report.SetHtmlContent(System.IO.Path.Combine(ExtAPI.Extension.InstallDir,"help","report.html"))
    report.Refresh()

def EmptyReset(step):
    pass

def LogReport(step):
    ExtAPI.Log.WriteMessage("Report:")
    for s in step.Wizard.Steps.Values:
        ExtAPI.Log.WriteMessage("Step "+s.Caption)
        for prop in s.AllProperties:
            ExtAPI.Log.WriteMessage(prop.Caption+": "+prop.DisplayString)

import random
import math

def RefreshCharts(step):
    graph = step.UserInterface.GetComponent("Graph")
    graph.Title("Line Bar Graph")
    graph.ShowLegend(False)
    graph.Plot([-1, 0, 1, 2, 3, 4], [0.5, -0.5, 0.5, -0.5, 0.5, 0.5], key="Variable A", color='g')
    graph.Bar([-1, 0, 1, 2, 3, 4], [10, 20, 30, 10, 5, 20], key="Variable B")

    graphB = step.UserInterface.GetComponent("GraphB")
    graphB.Title("Plot Graph")
    graphB.YTickFormat("0.2f")

    xValues = []
    yValues = []
    for i in range(0, 100):
        xValues.append(i)
        yValues.append(abs(math.sin(i*0.2))*i/100.0)
    graphB.Plot(xValues, yValues, key="y = a*sin(bx)", color="c")
    graphB.Plot(xValues, yValues, key="y = x", color="m")

    xValues = []
    yValues = []
    for i in range(0, 100):
        xValues.append(i)
        yValues.append(i/100.0)
    graphB.Plot(xValues, yValues, key="y = x", color="m")

    graphB.Plot([0, 10, 20, 30, 100], [0.2, 0.2, 0.2, 0.3, 0.3], key="Smth", color="r")
    graphB.Plot([0, 10, 20, 30, 100], [0.2, 0.1, 0.3, 0.5, 0.7], key="Smth", color="r")
    graphB.Plot([0, 10, 20, 30, 100], [0.2, 0.2, 0.2, 0.3, 0.3])

graphC = step.UserInterface.GetComponent("GraphC")
```

```

graphC.Title("Pie Graph")
graphC.Pie([1, 2, 3])
graphC.Pie([20, 30, 5, 15, 12], [0, "Banana", 2, 3, "42"])

graphD = step.UserInterface.GetComponent("GraphD")
graphD.Title("Bar Graph")
graphD.Bar(["Banana"], [70], key="key")
graphD.Bar([0, "Banana", 2, 3, 4], [20, 30, 5, 15, 12], key="key")

graphE = step.UserInterface.GetComponent("GraphE")
graphE.Title("Bubble Graph")
graphE.XTickFormat("f")
graphE.YTickFormat("f")
keys = ["one", "two", "three", "four", "five"]
colors = ["#BB3333", "#33BB33", "#3333BB", "#BBBB33", "#BB33BB"]
for c in range(0, 5):
    xValues = []
    yValues = []
    sizeValues = []
    for i in range(0, (c+1)*20):
        rad = random.randrange(c+1, c+2) + (random.random()*2-1)
        angle = random.random() * 2 * math.pi
        xValues.append(math.cos(angle) * rad)
        yValues.append(math.sin(angle) * rad)
        sizeValues.append(random.random() * 2.0 + 0.5)
    graphE.Bubble(xValues, yValues, sizeValues, key=keys[c], color=colors[c])

def CreateStaticStructural(step):
    template1 = GetTemplate(
        TemplateName="Static Structural",
        Solver="ANSYS")
    system1 = template1.CreateSystem()
    system1.DisplayText = "toto"

    nextStep = step.NextStep
    if nextStep!=None:
        nextStep.SystemName = system1.Name
        nextStep.ComponentName = "Geometry"
    nextStep = nextStep.NextStep
    if nextStep!=None:
        nextStep.SystemName = system1.Name
        nextStep.ComponentName = "Geometry"
    nextStep = nextStep.NextStep
    if nextStep!=None:
        nextStep.SystemName = system1.Name
        nextStep.ComponentName = "Geometry"
    nextStep = nextStep.NextStep
    if nextStep!=None:
        nextStep.SystemName = system1.Name
        nextStep.ComponentName = "Model"
    nextStep = nextStep.NextStep
    if nextStep!=None:
        nextStep.SystemName = system1.Name
        nextStep.ComponentName = "Model"

def OnSelectContext(step, prop):
    firstDMStep = step.NextStep
    secondDMStep = firstDMStep.NextStep
    firstSCStep = secondDMStep.NextStep
    secondSCStep = firstSCStep.NextStep

    firstGeoStep = step.NextStep
    if prop.Value == "DesignModeler":
        firstDMStep.IsEnabled = True
        secondDMStep.IsEnabled = True
        firstSCStep.IsEnabled = False
        secondSCStep.IsEnabled = False
    elif prop.Value == "SpaceClaim":

```

```
firstDMStep.IsEnabled = False
secondDMStep.IsEnabled = False
firstSCStep.IsEnabled = True
secondSCStep.IsEnabled = True

panel = step.UserInterface.Panel.GetComponent("Steps")
panel.UpdateData()
panel.Refresh()

def RefreshResultsProject(step):
    step.Properties["Res"].Value = ExtAPI.Extension.Attributes["result"]
    panel = step.UserInterface.Panel.GetComponent("Properties")
    panel.UpdateData()
    panel.Refresh()
```

This script defines all functions executed by callbacks in the XML file. Each step defined in the XML file can include multiple actions.

- Step 1 (Geometry): The callback `<onupdate>` executes the function `EmptyAction`. The callback `<onreset>` executes the function `DeleteGeometry`. The callback `<oninit>` executes the function `InitTabularData`.
- Step 2 (Mechanical): The callback `<onupdate>` executes the function `EmptyAction`. The callback `<onreset>` executes the function `DeleteMechanical`.
- Step 3 (Fluent): The callback `<onrefresh>` executes the function `CreateDialog`. The callback `<onupdate>` executes the function `EmptyAction`. The callback `<onreset>` executes the function `EmptyReset`.
- Step 4 (ReportView): The callback `<onrefresh>` executes the function `RefreshReport`. The callback `<onreset>` executes the function `EmptyReset`.
- Step 5 (CustomStep): The callback `<onrefresh>` executes the function `RefreshMechanical`. The callback `<onupdate>` executes the function `LogReport`. The callback `<onreset>` executes the function `EmptyReset`.
- Step 6 (Charts): The callback `<onrefresh>` executes the function `RefreshCharts`.

Functions executed by callbacks within property definitions are also defined.

Note

For a **mixed wizard**, the definition of the first step executed in the **Project** tab specifies the ANSYS products and component names from which subsequent steps are executed. For instance, in the following excerpted code, multiple actions are defined. These actions are called by the steps in the XML file.

```
...
def action1(step):

    template1 = GetTemplate( TemplateName="Static Structural", Solver="ANSYS" )
    system1 = template1.CreateSystem()
    geometry1 = system1.GetContainer(ComponentName="Geometry")
    geometry1.SetFile(FilePath=step.Properties["filename"].Value)

    nextStep = step.NextStep
    if nextStep!=None:
        nextStep.SystemName = system1.Name
        nextStep.ComponentName = "Geometry"
```

```

thirdStep = step.Wizard.Steps["Step3"]
if thirdStep!=None:
    thirdStep.SystemName = system1.Name
    thirdStep.ComponentName = "Model"
...

```

Custom Help for Wizards

To provide custom help for the steps in a wizard, you create HTML files, which you then store in a folder inside the extension. In each `<step>` block in the extension's XML definition file, you use the attribute `helpFile` to reference the relative path for the HTML file to display in the step's help panel.

Custom help files can contain any valid HTML5 syntax and supported media, such as images and graphs. You simply place the supported media files in the same folder as the HTML files.

In the following example, the help file `step1.html` is located in the directory `wizardhelp` within the extension.

```
<step name="Geometry" caption="Geometry" version="1" helpFile="wizardhelp/step1.html">
```

If you give the help files for steps the same names as the steps themselves and then store these files in a folder named `help`, you do not need to use the attribute `helpFile` to reference them. The extension can automatically find and display the appropriate help file for each step.

To provide custom help for the properties that are defined within a step, you can either create HTML files or define help text directly in the extension's XML file.

Creating HTML Files for Properties

If you choose to create HTML files for properties, you must give them the same name as the step and the property, separated by an underscore. For example, assume that `MeshingStep` is the step name and `MeshResolution` is the property name. The help file for the step must be named `MeshingStep`, and the help file for the property must be named `MeshingStep_MeshResolution`. When you use these file-naming conventions, you do not need to use the attribute `helpFile` to explicitly reference the HTML file for a step. The extension can automatically find and display the custom help for steps and their properties.

Note

For HTML files for properties, the recommended practice is to avoid images and to limit the text to no more than 200 characters.

Defining Help Text for Properties Directly in the XML File

If you choose to define help text for properties directly in the extension's XML file, you use child `<help>` blocks in the property definitions. For example, assume that a step has a property named `geometry-file`. You might define custom help for it as follows:

```

<property name="geometryfile" caption="Geometry file" control="fileopen" default="">
    <help>Select a geometry source file to import into AIM. Note the geometry should contain
the flow volume in addition to the structure.</help>
    <callbacks>
        <isvisible>isVisibleGeometryFile</isvisible>
        <isValid>isValidGeometryFile</isValid>

```

```
</callbacks>
</property>
```

Viewing Wizard Examples

For non-AIM wizard examples, see:

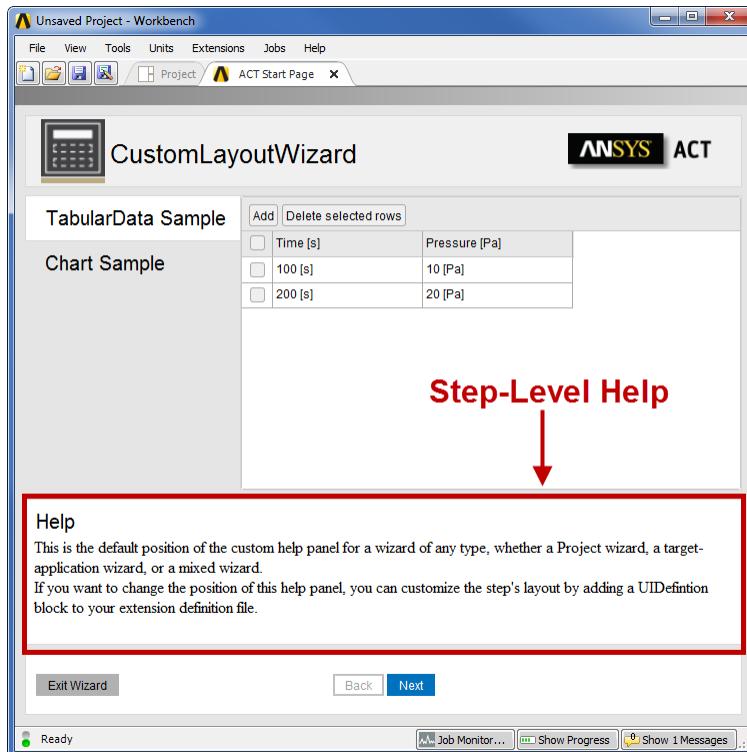
- Project Wizard (Workbench Project Tab)* (p. 308)
- Mechanical Wizard* (p. 333)
- DesignModeler Wizard* (p. 325)

For AIM wizard examples, see:

- AIM Custom Template (Single-Step) (p. 379)
- AIM Custom Template (Multiple-Step) (p. 384)

Viewing Custom Help for Non-AIM Wizards

When a non-AIM wizard is executed, the custom help for a step displays by default in a help panel at the bottom of the wizard interface.



You can change the location of this panel by customizing the layout of the wizard. For more information, see [Customizing the Layout of a Wizard](#) (p. 162).

Viewing Custom Help for AIM Wizards

When an AIM wizard is executed, the following occurs:

- Custom help for steps is shown in AIM's context-sensitive help panel. To access the help for a step, you click the question mark icon at the top of the step.

Note

After moving to a new step, you must click this icon to refresh the help panel with the new content.

- Custom help for properties (field-level help) is shown when you click the question mark icon that appears when you place the mouse cursor over a property.



Creating Conditional Steps

This section describes how to use the attribute **enabled** to create conditional steps. In the first line of a step definition, the attribute **enabled** defines the initial state of the step:

```
<step name="Mechanical" caption="Mechanical" enabled="true" version="1"
HelpFile="help/mechanical.html"><description>Create a mechanical component.</description>
```

When **enabled="true"** is the setting (default), the step initially displays in the wizard. However, in the IronPython script, you can programmatically change the state of the attribute **enabled** to create branches where certain steps are executed in only specific circumstances: **step.Enabled = True** or **step.Enabled = False**. Consequently, when running the wizard, the choices that the user makes in the step can determine which subsequent steps display.

Customizing the Layout of a Wizard

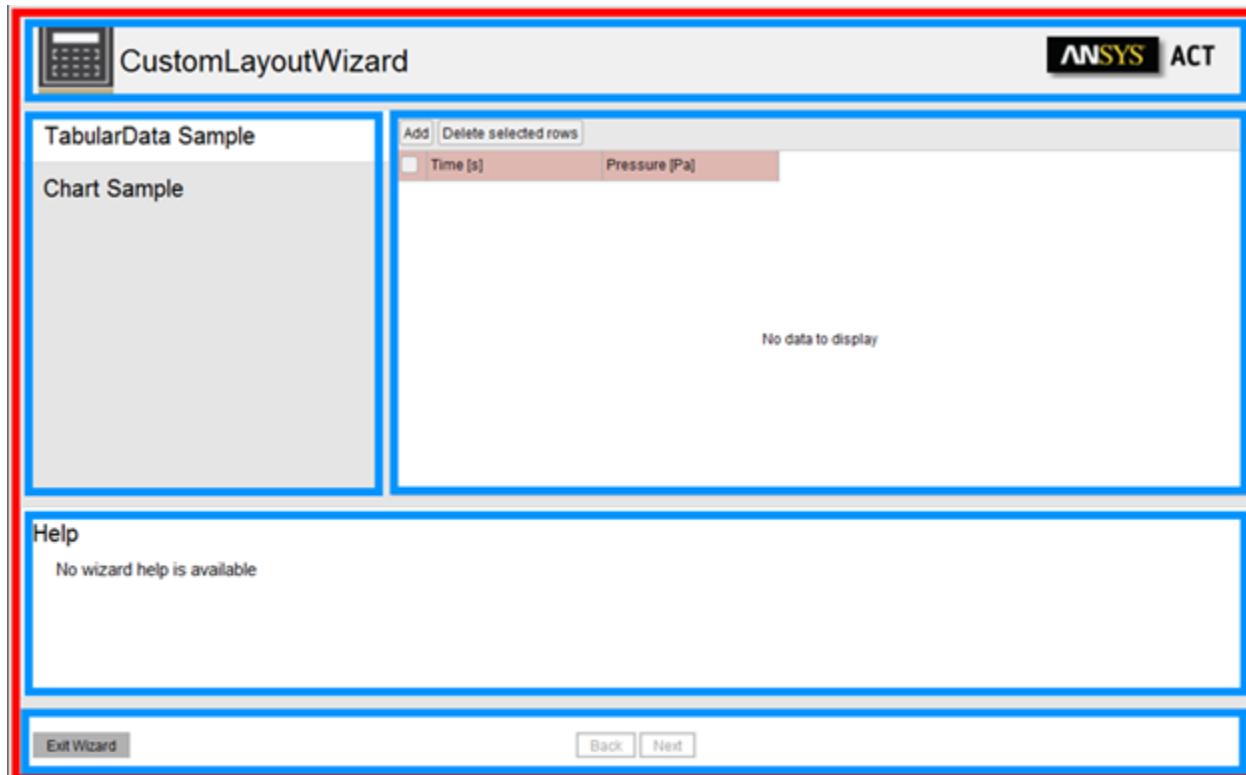
ACT provides you with the capability to customize the user interface of a non-AIM wizard, defining layouts both for the interface as a whole and for the individual interface components within the layout. Because the basic construction of an extension is described elsewhere, this section focuses solely on layout customizations.

Layout customizations are defined by the optional `<uidefinition>` block, which is placed inside the `<extension>` block in the XML definition file. The `<uidefinition>` block contains `<layout>` definitions, which contain child `<component>` blocks.

The basic definition of the `<uidefinition>` block follows:

```
<uidefinition>
  <layout>
    <component/>
    <component/>
    <component/>
    <component/>
  </layout>
</uidefinition>
```

The customizations are exposed in a non-AIM wizard as follows:



User Interface Definition

The `<uidefinition>` block defines a single user interface (UI) for the wizard. It is the top-level entry that contains all layout definitions.

Layout Definition

The `<layout>` block defines a single custom layout within the UI definition for the wizard. A separate layout can be defined for each step in the wizard. The `<layout>` block contains all component definitions.

The `<layout>` block has the following attributes:

- The attribute `name` defines the name of the layout. The name is referenced in a wizard step in the following format:

```
LayoutName@ExtensionName
```

This notation allows a wizard to reference a layout defined in a different extension.

Component Definition

The `<component>` block defines a single component of the wizard interface.

The `<component>` block has the following attributes:

- The mandatory attribute `name` defines the name for the component.
 - The component name is required to position it next to another component. To do this, use the name in conjunction with the following attributes:

→ `leftAttachment`

→ `rightAttachment`

→ `bottomAttachment`

→ `topAttachment`

- The component name is also used in the IronPython script with the method `GetComponent()`. For instance, in a step, the callback `Onrefresh` can use the following code:

```
component = step.UserInterface.GetComponent(ComponentName)
```

- The mandatory attributes `heightType`, `height`, `widthType`, and `width` define the dimensions of the component. Possible values are as follows:
 - **FitToContent**: The component is to have a default size that is set by ACT. When selected, the attributes `height` and `width` become irrelevant.
 - **Percentage**: The height and width of the component is expressed as percentages.
 - **Fixed**: The height and width of the component is expressed in pixels.
- The mandatory attribute `componentType` defines the type of component. Some examples of possible component types are:
 - `startPageHeaderComponent` (Defines banner titles)
 - `propertiesComponent`
 - `chartComponent`
 - `tabularDataComponent`

- **buttonsComponent**
- **stepsListComponent**
- The attributes **leftOffset**, **rightOffset**, **bottomOffset**, and **topOffset** specify the distance and the margin between the different components.
- The attributes **customCSSFile**, **customHTMLFile**, and **customJSFile** provide advanced capabilities to customize an entire component.

For an example of customizing a wizard interface, see [Wizard Custom Layout Example \(p. 391\)](#).

Installing and Loading Wizards

For AIM, Workbench, any target product launched through Workbench, and stand-alone Fluent, the installation process for a wizard is the same as for any other extension. For more information, see [Installing a Scripted Extension \(p. 33\)](#) and [Installing a Binary Extension \(p. 33\)](#).

To install a SpaceClaim or Electronics Desktop wizard when Workbench and AIM are not installed, the install locations are different. For more information, see [Installing a SpaceClaim Wizard \(p. 164\)](#) and [Installing an Electronics Desktop Wizard \(p. 164\)](#).

The process of loading and unloading wizards is as for any other extension. For more information, see [Loading and Unloading Extensions \(p. 35\)](#).

Installing a SpaceClaim Wizard

To install a SpaceClaim wizard, save the wizard extension and associated files to one of the following locations:

- %ANSYSversion_DIR%\scdm\Addins
- Any of the include directories specified by using the gear icon on the **Extension Manager** (accessed via the [ACT Start Page](#))

Installing an Electronics Desktop Wizard

To install an Electronics Desktop wizard, save the wizard extension and associated files to one of the following locations:

- ...\\AnsysEM\\v182\\ACT\\win64\\syslib\\ACT
- Any of the include directories specified by using the gear icon on the **Extension Manager** (accessed via the [ACT Start Page](#))

Using Wizards

This section discusses accessing and executing an installed and loaded wizard:

[Using a Workbench Project or Non-AIM Target Product Wizard](#)

[Using an AIM Target Product Wizard](#)

Using a Workbench Project or Non-AIM Target Product Wizard

This section describes how to execute an installed and loaded project wizard or non-AIM target product wizard:

[Launching a Wizard from the Wizards Page](#)

[The Wizard Interface](#)

[Entering Data in a Wizard](#)

[Exiting a Wizard or Project](#)

Launching a Wizard from the Wizards Page

To launch a wizard:

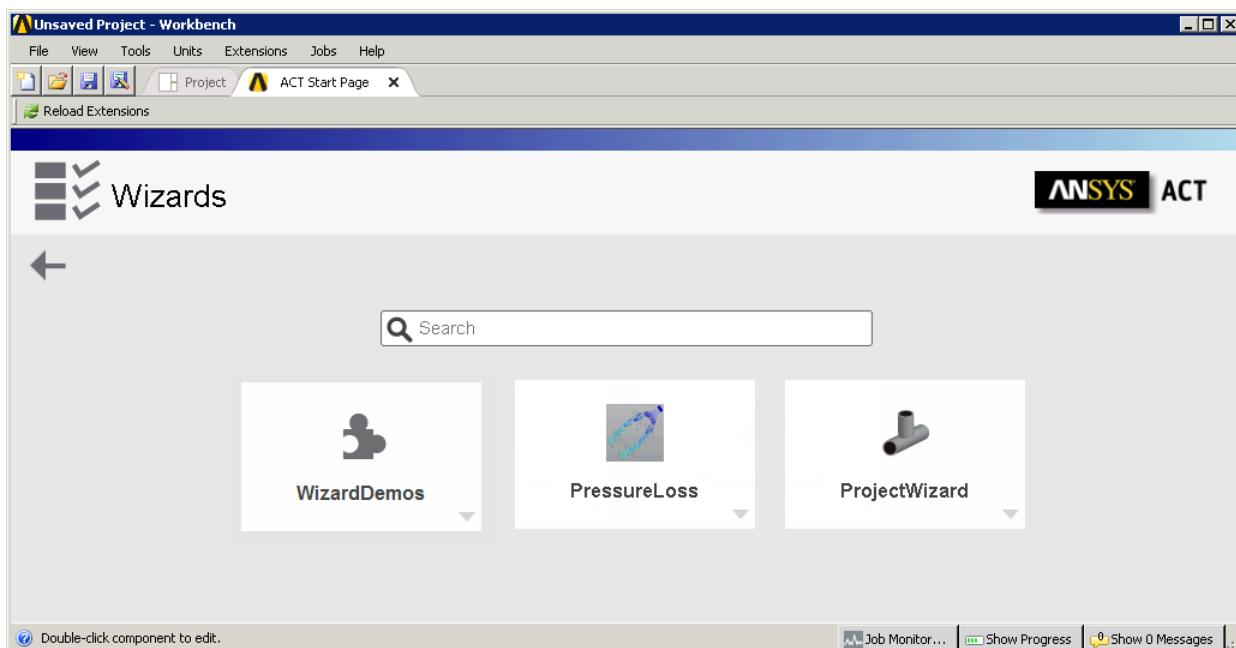
1. Open the ANSYS product in which the wizard is to execute:
 - For a project wizard run in AIM, open AIM.
 - For a target product wizard run in a standalone instance of SpaceClaim, Electronics Desktop, or Fluent, open the corresponding product.

Note

If opening Fluent, in the **Fluent Launcher**, you can select the **Load ACT** check box to display the **ACT Start Page** in Fluent.

- For all other wizards, open Workbench.
2. Open the **ACT Start Page** as follows:
 - In AIM or Workbench, click the **Extensions > ACT Start Page** menu option.
 - In SpaceClaim, click the **Prepare > ACT Start Page** toolbar icon.
 - In Electronics Desktop, click the **View > Extensions** menu option.
 - In Fluent, if you have not already enabled ACT functionality in the **Fluent Launcher**, do so now by selecting **ACT** in the **Arrange the workspace** menu.
 3. On the **ACT Start Page**, click the **Wizards** icon.

The **Wizards** page opens, showing all wizards currently installed for this context.



4. Click the wizard that you want to run. Or, right-click the wizard and select **Execute Wizard**.

The wizard interface opens, showing the first step of the wizard.

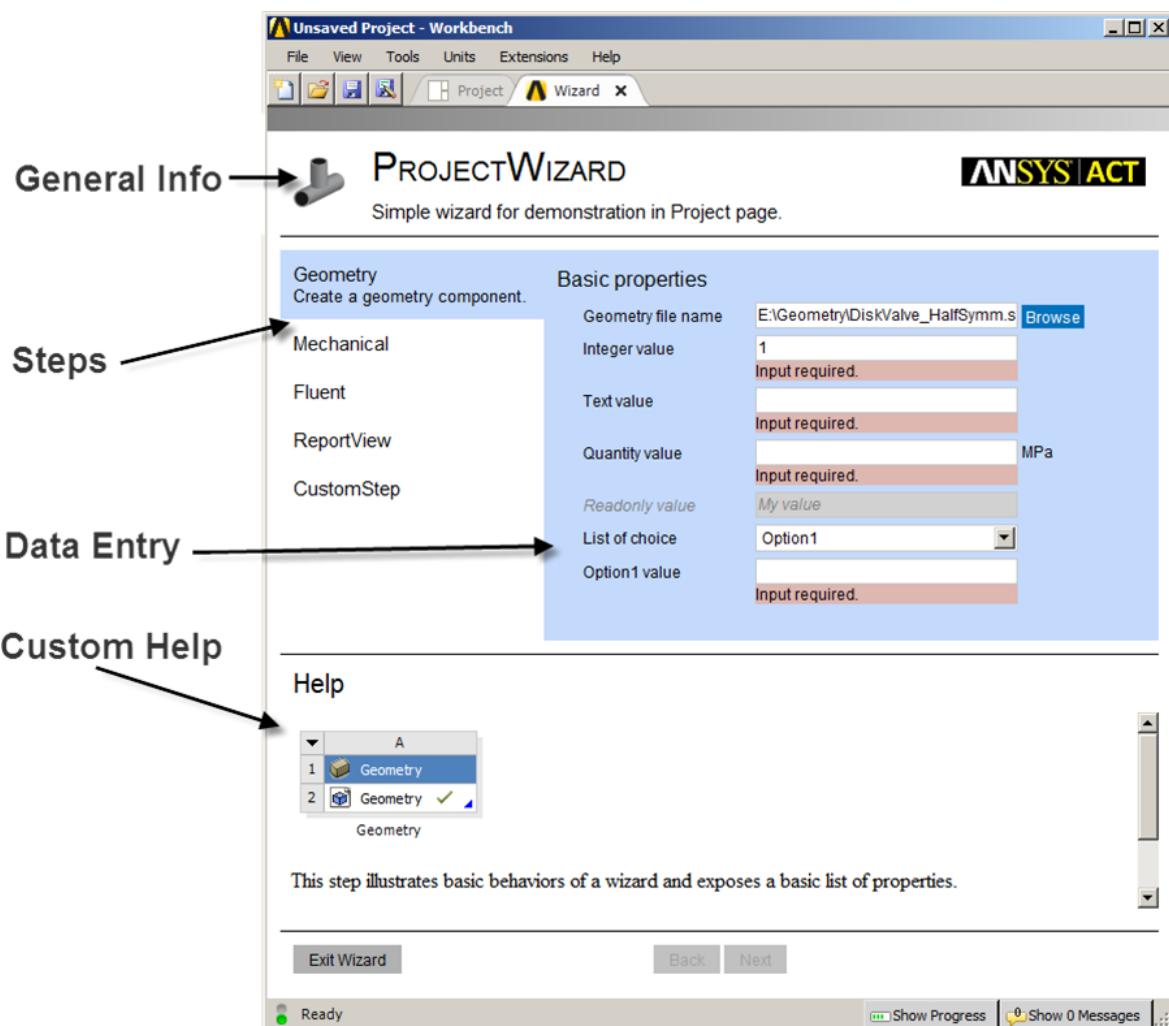
Note

You can use the **Search** field to search for a wizard by the name, description, or author of the extension. The case-insensitive search tool looks for all strings entered that are separated by a blank (serving as the AND operation) and performs a combined search for strings separated by the OR operation.

The Wizard Interface

The default interface for a Workbench-based wizard is divided into four panels:

- The top panel shows general information about the wizard and a logo of your choice.
- The left panel shows a list of the steps in the wizard, with the current step highlighted.
- The right panel shows the data-entry and data-display fields for the selected step.
- The bottom panel shows the custom help for the selected step.



Note

The overall content and function of the wizard interface is consistent across ANSYS products. However, the display can vary slightly, depending on the product and any customizations to the interface.

Entering Data in a Wizard

For each step of the wizard, enter your data in the fields provided. Entered values are validated as specified in the extension.

When all required values have been entered and pass validation, the **Next** button becomes enabled. When you click this button, your data is submitted and the action defined in the step is executed. There might be a progress bar for the step's execution, and a message confirming the completion of the step execution can display. You can also verify completion of the step by reviewing the **Project** tab.

Once you are past the first step, the **Back** button is enabled. When you click this button, the wizard returns to the previous step, but the data generated for that step is lost.

To enter tabular data:

1. Click the **Edit** button.

2. Click the **Add** icon to add a table row.
3. Enter values into the cells.
4. When finished entering data, click the green check box icon to save or the red stop icon to cancel.

Exiting a Wizard or Project

To exit a wizard that has completed, click the **Finish** button. You are returned to the **ACT Start Page**.

To exit a wizard at any time, click the **Exit Wizard** button in the lower left corner of the interface. You are returned to the **Wizards** page.

Note

If you exit the wizard midway through, the changes to the project are retained and can be saved, but the wizard itself is not. When using the **Exit Wizard** button, you cannot resume the wizard where you left off.

To exit the project with a wizard still in progress, save the project and exit as usual. On reopening the project, the wizard is still in its last saved state, so you can resume where you left off.

Using an AIM Target Product Wizard

A target product wizard for AIM is implemented as a custom template, guided simulation, or guided setup. When creating a target product wizard for AIM, you set the attribute `context` to `Study`. You then set the attributes `persistent` and `subcontext` to indicate the extension type. These attributes control where AIM displays the extension. The AIM product documentation refers to the following extension types as *apps*.

Extension Type	Persistent Attribute	Subcontext Attribute
Custom template	Unspecified	Unspecified
Guided simulation	Specified	Unspecified
Guided setup	Unspecified	Specified

The custom templates and guided simulations that you have installed and loaded in AIM are available on the start page and when adding a new simulation on the **Study** panel. They are listed after ANSYS-supplied templates. On the start page, a guided simulation has the following icon on the bottom left of the tile to differentiate it from a custom template:



Guided setups that are installed and loaded appear in the **Guide Me** context (right-click) menu when a compatible object and geometry entity are selected. The **Guide Me** menu is present only if the object and geometry selection to which you've navigated matches the object or task and geometry selection defined in the extension.

Related Topics

- [Creating a Custom Template](#)
- [Creating a Guided Simulation](#)

Creating a Guided Setup

Creating a Custom Template

A custom template automates the definition of a simulation process, prompting you for needed inputs. In the XML definition file of an ACT extension for AIM, both the attributes **persistent** and **subcontext** on the **<wizard>** block are unspecified by default. In this case, the extension is a custom template. Custom templates continue to function in AIM in the same way as they have always have functioned. All custom templates that are installed and loaded in AIM are available on the start page and when adding a new simulation on the **Study** panel.

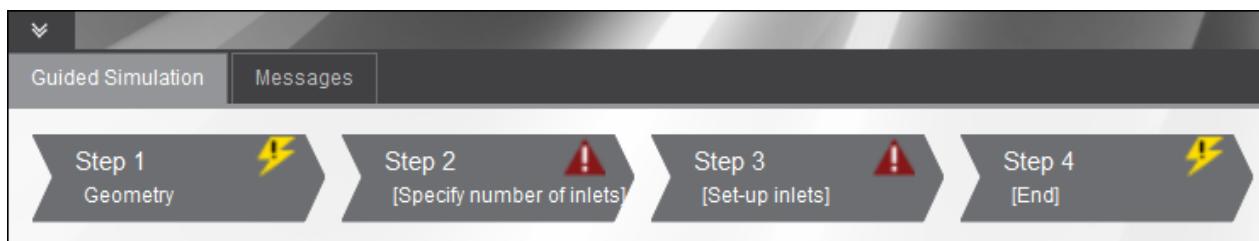
ANSYS supplies two examples of AIM custom templates that you can install and load if desired. For more information, see:

- [AIM Custom Template \(Single-Step\) \(p. 379\)](#)
- [AIM Custom Template \(Multiple-Step\) \(p. 384\)](#)

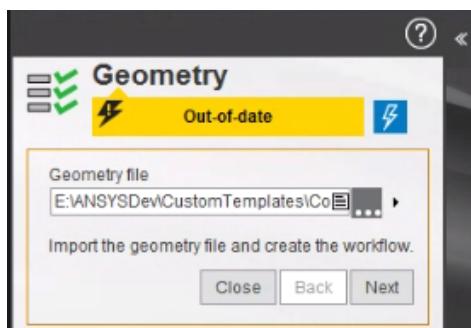
Creating a Guided Simulation

A guided simulation displays your own end-to-end custom workflow in a persisted view, remaining open until you close it. In the XML definition file of an ACT extension for AIM, you set the attribute **persistent** on the **<wizard>** block to **true** to create a guided simulation. All guided simulations that have been installed and loaded in AIM are available on the start page and when adding a new simulation on the **Study** panel. ANSYS does not supply any examples of guided simulations.

When you launch a guided simulation, the view panel immediately displays a **Guided Simulation** tab with all steps in the workflow. You do not have to complete the steps in a guided simulation sequentially.



Clicking any step on the **Guided Simulation** tab makes the step active in the data panel. Using this tab, you can navigate to the various steps, supplying appropriate data and updating as necessary.



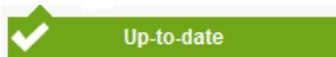
You can view the state of the active step in the data panel. You can view the states of all steps on the **Guided Simulation** tab. If a step is disabled during execution, it is removed from this tab.

Updating a Step in a Guided Simulation

In the data panel, the visual indications that can be shown for the state of a step are a subset of those shown for AIM tasks. A step can be in any of three states:



The **Attention required** state is tied to the current callback `<isValid>` to determine if the data supplied for the step is valid. A step in this state has one or more fields that require your attention before the step can be updated. The step cannot be updated if any previous step requires attention.



The **Up-to-date** state indicates that the callback `<onUpdate>` for the step has been executed and that no change was made to this step or any previous step.



The **Out-of-date** state indicates that the step has been modified after an update and an update is once again required. An up-to-date step becomes out-of-date if the data for a previous step is changed.

When an out-of-date step is active in the data panel, you click either the update icon or the **Update** button to execute the callback `<onUpdate>` for the step. The **Update** button is blue when the step requires an update. In a guided simulation, the update of a step invokes the callback `<onUpdate>` for all previous steps that need to be updated.

Unlike a simulation template, a guided simulation is not dismissed when the last step is executed. Unless you close a guided simulation, you can go back to any step and make changes and update multiple times.

Closing a Guided Simulation

In principle, users of a guided simulation leave it running because they do not need to interact directly with AIM. However, a user can close a simulation by clicking the **Close** button available on any step in the data panel.

Currently, only one guided simulation or simulation template can be open at a time. If you attempt to launch another, a dialog box opens, indicating the name of the one that is already running. You can choose to either close the running instance or let it remain running.

If you choose to close a guided simulation and then launch it again later, the last created data is restored.

Key Distinctions Between Guided Simulations and Simulation Templates

A summary follows of the key distinctions between guided simulations and simulation templates.

- Data supplied in a guided simulation is persisted, which is not the case for a simulation template. Each time you launch a simulation template, a new instance is opened, which means that no previously created data is restored. Each time you launch a guided simulation, the same instance is opened, even

after closing and then resuming a project. This means that the data that you last created is always restored.

- A guided simulation uses the **Next** and **Back** buttons in the data panel only for navigation through the steps. These buttons are not tied to the execution of callbacks as they are in a simulation template. Because the steps in a guided simulation do not have to be completed sequentially, you can navigate to any step, regardless of its state.
- Each step in a guided simulation has **Close** and **Update** buttons.
 - If you click **Close**, a dialog box opens, asking you to confirm that you want to close the guided simulation. Because steps in a guided simulation do not have to be completed sequentially, the last step does not have a **Finish** button.
 - When you click the update icon  or the blue **Update** button for an out-of-date step in the data panel, the callback `<onUpdate>` for the step is invoked. When the update of a step is invoked, the callback `<onUpdate>` for any previous step that is out-of-date is executed before the update of the current step. If any previous step requires attention, the **Update** button is not shown.
- On the **Guided Simulation** tab in the view panel, each step has a context menu with a command for executing the callback `<onUpdate>`. If the callback `<IsReset>` is implemented for the step, the context menu also includes a command for resetting the step. However, the reset of a step does not invoke the callback `<IsReset>` for previous steps.
- When you start working on a guided simulation, the **Workflow** tab in the view panel disappears.

Creating a Guided Setup

In the XML definition file of an ACT extension for AIM, the attribute **subcontext** on the `<wizard>` block specifies if it is a guided setup. Installed and loaded guided setups appear in the **Guide Me** context (right-click) menu when a compatible object and geometry entity are selected. Various examples follow of what you can enter in the `<wizard>` block of a guided setup named **Test1**. For the attribute **subcontext**, you specify a task, task group, or object.

Example 1

```
<wizard name="Test1" caption = "Test1" version="1" context="Study" subcontext = "Physics Solution/Face" icon="Terminal" persistent="False">
```

When AIM is displaying a task group of type **Physics Solution**, the template is shown in the **Guide Me** context menu for the selected face.

Example 2

```
<wizard name="Test1" caption = "Test1" version="1" context="Study" subcontext = "Support/Face" icon="Terminal" persistent="False">
```

When AIM is displaying an object of type **Support**, the template is shown in the **Guide Me** context menu for the selected face.

Example 3

```
<wizard name="Test1" caption = "Test1" version="1" context="Study" subcontext = "Face" icon="Terminal" persistent="False">
```

Regardless of the type of task, task group, or object that AIM is displaying, the template is shown in the **Guide Me** context menu for the selected face.

Example 4

```
<wizard name="Test1" caption = "Test1" version="1" context="Study" subcontext = "Meshing" icon="Terminal" persistent="False">
```

The template is shown in the **Guide Me** content menu for the selected meshing task.

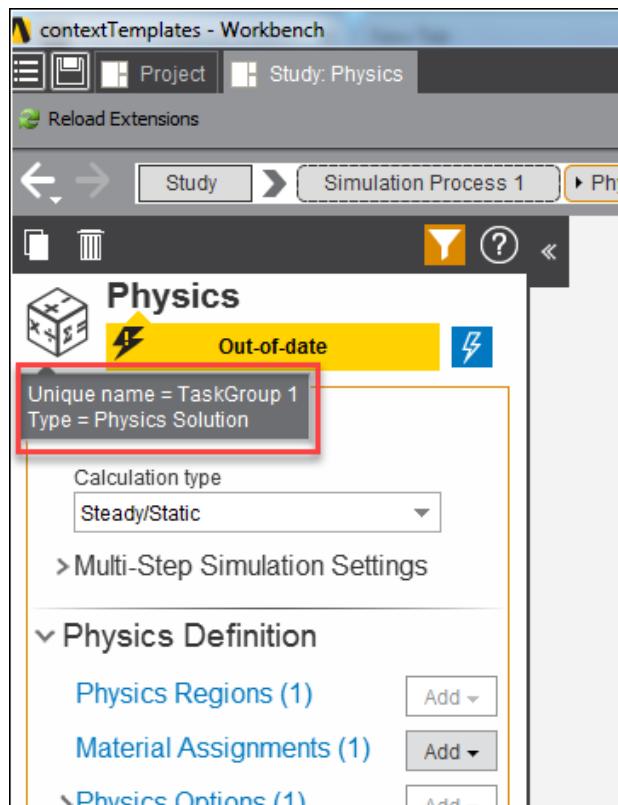
Summary

As you can see by the first two code examples, a forward slash (/) in the attribute **subcontext** identifies a task, task group, or object before the slash and a selection type after the slash. Object types include **Vertex**, **Edge**, **Face**, and **Body**.

The second and third code examples demonstrate how you can enter a string for matching only a particular type of task, task group, or task.

Tip

Placing the mouse cursor in the upper left corner of the AIM data panel displays a tooltip that indicates the internal name of the task, task group, or object. The following figure shows the task group with the internal name of **Physics Solution**.



APIs Description

ACT supports customization of ANSYS products with a set of interfaces. This chapter describes the APIs that ACT delivers. The high-level member interfaces of the APIs expose the properties and methods that allow access to the underlying product data.

The extensions that are loaded by the Workbench Extension Manager are each configured and initialized to support Python scripting. For each extension, a global variable `ExtAPI` gives access to the property `DataModel`. This property returns the object `IDataModel`, which has properties that return all of the high-level interfaces available for customizations. The complete list of member interfaces is given in the [ANSYS ACT API Reference Guide](#).

ACT Automation API

Each ANSYS product can leverage its own scripting language and define unique object API. ACT minimizes inconsistency and confusion introduced by product-specific syntax by providing its own *Automation API*. Automation objects and methods wrap underlying product API by re-exposing user-facing features in a clear and consistent syntax. Instead of piecemeal script statements for cross-product customization, you can program with one ACT language.

The ACT Automation API provides coverage for the following ANSYS products:

- ANSYS Workbench
- ANSYS Mechanical
- ANSYS DesignModeler

You typically engage *Automation*-level objects when issuing statements inside the **ACT Console** or invoking `ExtAPI.DataModel` members. You can confirm Automation API usage by executing a variable name in the **ACT Console**. If the returned string starts with `Ansys.ACT.*.Automation.`, you are working with the Automation API. ACT substitutes the appropriate product name in place of the asterisk (such as `Ansys.ACT.WorkBench.Automation.`).

Conversely, you typically encounter *SimEntity*-level objects as arguments to extension callbacks.

APIs for Custom Workflows in the Workbench Project Page

This section describes the APIs provided by ACT for the customization of the Workbench **Project Schematic**. These APIs provide access to items in the Workbench schematic, including the framework data model, interface components, and user objects and their properties.

Using the APIs, you can create custom workflows in the Workbench **Project Schematic**. At the workflow level, you can use global callbacks to invoke schematic actions (both installed and custom). You can

also create and expose custom tasks (components) and task groups (systems) that can interact with each other, as well as with installed components and systems, as part of the schematic workflow.

Note

In the APIs for custom workflows, class names and member names do not include hyphens. In a PDF file of this guide, column width limitations cause hyphenation of long class and member names in the tables that list the available APIs. You can ignore any hyphenation in these table entries.

Accessing Project-Level APIs

You can access project-level ACT custom workflow APIs off the master **ExtAPI** entry point. The following APIs are available:

Project

The top level of the hierarchy is the framework-level project. It provides additional entry points to the Workbench data model, file manager, and so on. It is accessed as follows:

```
project = ExtAPI.DataModel.Project
```

Context

This is the framework-level command context. It provides additional entry points to the session and project utilities, and is accessed as follows:

```
context = ExtAPI.DataModel.Context
```

Tasks

This is all of the tasks within the schematic, which includes both ACT-defined and pre-installed tasks. It does not include the “master task” level, such as ACT and schematic “template-like” tasks. Only concrete task instances are included. It is accessed as follows:

```
tasks = ExtAPI.DataModel.Tasks
```

Task Groups

This is all of the task groups within the schematic, which includes both ACT-defined task groups and pre-installed task groups. It does not include the “master task group” level, such as ACT and schematic “template-like” task groups. Only concrete task group instances are included. It is accessed as follows:

```
taskGroups = ExtAPI.DataModel.TaskGroups
```

Note

This section provides only the top-level API access points to schematic and workflow-related data. For a comprehensive list of APIs for custom workflows, see the **<workflow>** section of the [ANSYS ACT API Reference Guide](#).

Accessing Task APIs

As of ANSYS 18.0, Workbench tasks are exposed at the user level through [Automation API \(p. 173\)](#) wrappers in the namespace **Ansys.ACT.WorkBench.Automation.Workflows**. The Project Data Model returns this wrapper type:

```
task = ExtAPI.DataModel.Tasks[0]
```

The task wrapper does not store any local data. All data access and modification occur at member call time.

The following table lists the APIs that are available. For the class **Task**, property access is provided through a dictionary. You access a property wrapper by name to get or set values. For example, the Geometry component exposes the property **ImportSolidBodies**. You can access and change this property as follows:

```
task = ExtAPI.DataModel.Tasks[0]
task.Properties["GeometryImportSolidBodies"].Value = True
```

Note

ACT-defined callbacks still receive **UserTasks** as arguments. A **UserTaskTemplate** is now accessible from the property **Template**. All other task functionality remains intact.

Class	Member	Description
ITask	Name	The internal task name.
	Caption	The user-visible task display text.
	TaskGroup	The parent task group.
	InternalObject	The task's internal object.
Task	TaskId	The internal Workbench component identifier.
	Caption	The Workbench UI-visible display text.
	TaskGroup	The task group wrapper for the component's system instance.
	InternalObject	The Workbench component.
	SourceTasks	The list of tasks providing data to this task.
	TargetTasks	The list of tasks consuming data from this task.
	State	The task state.
	Template	Gets the template from which the task was created.
	UserId	Gets the ID to display in the UI.
	DirectoryName	Gets the data directory name to display in the UI.
	MasterTask	Gets the master task from which the task directly consumes shared data. This is always the immediate upstream sharing source, even if the sharing is full-share. For the true root data task, use RealTask .
	RealTask	Gets the shared data root task from which the task consumes shared data. This is always the first (original,

Class	Member	Description
		non-grayed) sharing source if fully shared. Otherwise, it is the task itself.
	IsSharing	Gets a value indicating whether the task is a slave of a master-sharing source.
	Container	Gets the container for the task.
	Notes	Gets the task notes.
	UsedLicenses	Gets the used license for the component. This dictionary maps the license IDs to the number of used licenses for the task.
	Solver	Gets the task solver type.
	ImageName	Gets the name for the task icon image.
	IsFailedState	Gets a value indicating whether the task is in a failed state from a previous data operation, such as update or refresh.
	AlwaysBePartOfD-pUpdate	Gets a value indicating whether the task should always be part of a design point update, even if it does not affect any parameter value.
	CoupledClients	Gets the list of coupled client data tasks.
	CoupledToTask	Gets the coupling server task (if available).
	CoordinateId	Gets the task coordinate ID as a union of the task group letter and task index.
	Parameters	The list of input and output parameters associated with the task.
	Properties	Gets a dictionary of properties, accessible by property name.
	InputData	Gets the input data, by type string key, for the task.
	OutputData	Gets the output data, by type string key, for the task.
	GetProperty-Names()	Gets the names of all properties declared with the task's data model.
	ExecuteCom-mand(command Name, arguments)	Executes a specific task-bound command.
	Clean()	Deletes the heavyweight data of this task to reduce the size of the project. Heavyweight data can include the solution, results, or both.
	Refresh()	Refreshes the input data for a task by reading all changed data from upstream

Class	Member	Description
		(source) tasks. No calculations or updates based on this new data are performed.
	Reset()	Resets the task by removing all user input and result data.
	Update()	Updates the task by refreshing the input from all upstream components and then performs a local calculation based on current data.
	Remove()	Deletes the task and all dependents from its task group.

Accessing Task Template APIs

As of ANSYS 18.0, Workbench task templates are exposed at the user level through [Automation API \(p. 173\)](#) wrappers in the namespace `Ansys.ACT.WorkBench.Automation.Workflows`. The Project Data Model returns this wrapper type:

```
task = ExtAPI.DataModel.Tasks[0]
template = task.Template
```

The task template wrapper does not store any local data. All data access and modification occur at member call time.

The following table lists the APIs that are available.

Class	Member	Description
<code>TaskTemplate</code>	<code>Name</code>	Gets the task template name.
	<code>TaskTypeAbbreviation</code>	Gets the default name for the task.
	<code>DisplayName</code>	Gets the user-visible name of the task created from the template. If the task group template gives a name for the task using the property <code>TaskNames</code> , it takes precedence.
	<code>Solver</code>	Gets the solver type in the task. If specified, this contributes to property <code>Solver</code> for the containing task group.
	<code>IsResults</code>	Gets a value indicating whether the task represents result data, which is then not duplicated when the user invokes <code>DuplicateSystemWithoutResults</code> .
	<code>IsShareable</code>	Gets a value indicating whether one instance of this type of task can be shared by another task.
	<code>IsReplaceable</code>	Gets a value indicating whether sharing of this type of task can be changed dynamically from sharing to unsharing or vice versa.

Class	Member	Description
	IsReplaceableWithSharing	Gets a value indicating whether sharing of this type of task can be changed dynamically from an independent task to a full-sharing task. This covers one of the two aspects of what IsReplaceable defines.
	IsDeletable	Gets a value indicating whether this type of task and its downstream tasks can be deleted from its task group.
	IsPartialShareOnly	Gets a value indicating whether this type of task engages in only partial sharing.
	DontShareByDefault	Gets a value indicating whether this type of task has all connections established as "provides" connections to new systems (instead of sharing).
	RequiredInputTypes	Gets a list of types that are required as inputs for the task to be valid.
	OutputTypes	Gets a list of types that are contained in the task after creation.
	EquivalentInputs	Gets a list of equivalent entity types as inputs for the task.
	CreatingCommandName	Gets the name of a command that creates the container to be inserted in the task group.
	DeletingCommandName	Gets the name of a command that is invoked before deleting the container.
	UpdateCommandName	Gets the name of a command that updates the container's contents based on the upstream containers on which it depends.
	DataGroupUpdateCommandName	Gets the name of a command that updates the data groups of the task's container.
	WillBePendingUponUpdateQueryName	Gets the name of a query that returns whether the task goes into the pending state upon update.
	WillBeUpdatedViaRsmQueryName	Gets the name of a query that returns whether the task is updated via ANSYS Remote Solve Manager (RSM) or the EKM (Engineering Knowledge Manager) Portal.
	RemoteSolvePermissionsQueryName	Gets the name of a query that returns whether the task can be part of a design point update via RSM or the Portal.

Class	Member	Description
	EditorInitiatedRemoteUpdateInfoQueryName	Gets the name of a query that returns all remote update information for this task that was initiated from within the add-in editor.
	RefreshCommandName	Gets the name of a command that refreshes the task's inputs from upstream containers.
	DataSourceListChangedCommandName	Gets the name of a command that handles input source changes for the task.
	TaskStatusQueryName	Gets the name of a query that returns the status of the task.
	DataGroupStatusQueryName	Gets the name of a query that returns the status of the data groups of the task's container.
	DuplicateTaskCommandName	Gets the name of a command that duplicates the entire task content.
	DuplicateUserDataCommandName	Gets the name of a command that duplicates only the user input data.
	AvailableTransferTasks	Gets the list of available transfer tasks that can be exposed from the task.
	ReportContentCreationCommandName	Gets the name of a command that provides general report content for the task.
	DesignPointReportContentCreationCommandName	Gets the name of a command that provides design point report content for the task.
	CleanCommandName	Gets the name of a command that cleans the container's heavyweight output data and deletes associated data files.
	ResetCommandName	Gets the name of a command that resets the container's internal data.
	AboutToResetCommandName	Gets the name of a command that prepares to reset the container's internal data. The task can stop resetting by throwing an exception.
	ComponentPropertyDataQueryName	Gets the name of a query that returns custom data to show in the task group properties view.
	CanUseTransferDataQueryName	Gets the name of a query that returns whether a user-connection can be established between a transfer task and a data task.

Class	Member	Description
	CanDuplicateQueryName	Gets the name of a query that returns whether a task created from the template can be duplicated.
	DuplicateGroup	Gets the group within which tasks can potentially be replaced or duplicated.
	ComponentImageName	Gets the name of the default image to use as the task icon.
	ValidationCodes	Gets the task's validation codes.
	DefaultTransferName	Gets the name given to the default outputs of the task if they are offered as part of a transfer selection popup.
	IncludeInPartialUpdate	Gets a value indicating whether to consider the task for a partial update.
	RefreshAfterPartialUpdate	Gets a value indicating whether to refresh the downstream task after a partial update.
	ShowOptionToAlwaysBePartOfDpUpdate	Gets a value indicating whether the user should be given the option to always update the task as part of a design point update.
	GetEnvironmentForRsmDpUpdateQueryName	Gets the name of a query that returns environment variables set as part of a design point update via RSM.
	GetRemoteUpdateMonitorFilesQueryName	Gets the name of a query that returns monitor files during a RSM design point update.
	SubTasks	Gets the list of subtasks.
	SubTaskStatusQueryName	Gets the name of a query that returns subtask states.
	IsDisabled	Gets a value indicating whether the template is a fake template for an orphaned task, where the normal template is not accessible. The task should be shown as disabled and no operations should be available.
	IsAim	Gets a value indicating whether the template is an AIM task template.
	InternalObject	Gets the internal component template represented by the task template.

Accessing Task Group APIs

As of ANSYS 18.0, Workbench task groups are exposed at the user level through [Automation API \(p. 173\)](#) wrappers in the namespace **Ansys.ACT.WorkBench.Automation.Workflows**. The Project Data Model returns this wrapper type:

```
taskGroup = ExtAPI.DataModel.TaskGroups[0]
```

The task group wrapper does not store any local data. All data access and modification occur at member call time.

The following table lists the APIs that are available.

Note

When a task group is accessed from an ACT script, an instance of **UserTaskGroup** is still used. A **UserTaskGroupTemplate** is now accessible from the property **Template**. All other task group functionality remains intact.

Class	Member	Description
ITaskGroup	Name	The internal task group name.
	Caption	The user-visible task group display text.
	Tasks	The tasks contained in the task group.
	InternalObject	The task group's internal object.
TaskGroup	TaskId	The internal Workbench system identifier.
	Caption	The Workbench UI-visible display text (system caption).
	Abbreviation	The Workbench system abbreviation.
	InternalObject	The Workbench system.
	AnalysisType	The Workbench system analysis type.
	DirectoryName	The Workbench system directory name.
	Physics	The list of Workbench system physics.
	Solver	The list of Workbench system solvers.
	Notes	The Workbench system notes.
	HeaderText	The Workbench system block title.
	IsParametric	Indicates whether the Workbench system operates solely on parameters (appears below the Parameter Set bar in the Workbench Project Schematic).
	Tasks	The list of tasks contained in this task group.
	Template	Gets the template from which the task group was created.
	GetTaskByName(name)	Gets the contained task by the supplied name.
	Clean()	Clears generated data on all tasks in the task group.
	Copy()	Creates a new task group containing a copy of all the data in the task group.

Class	Member	Description
	Delete()	Deletes the task group and all contained data from the project.
	Refresh()	Refreshes the input data for all tasks in the task group by reading changed data from upstream sources. No calculations or updates based on the new data are performed.
	Update()	Updates all tasks in the task group by refreshing the input from all upstream sources and then performing local calculation based on the current data.
	RecreateDeletedTasks()	Recreates new versions of any tasks that have been deleted from the task group.

Accessing Task Group Template APIs

As of ANSYS 18.0, Workbench system task group templates are exposed at the user level through [Automation API \(p. 173\)](#) wrappers in the namespace `Ansys.ACT.WorkBench.Automation.Workflows`. The Project Data Model returns this wrapper type:

```
taskGroup = ExtAPI.DataModel.TaskGroups[0]
taskGroupTemplate = taskGroup.Template
```

The task group template wrapper does not store any local data. All data access and modification occur at member call time.

The following table lists the APIs that are available.

Class	Member	Description
<code>TaskGroupTemplate</code>	<code>Name</code>	Gets the task group template name.
	<code>Category</code>	Gets the toolbox category in which the template appears.
	<code>TaskGroupType</code>	Gets the task group type to display in the UI in the header of the task group block.
	<code>TaskGroup-TypeAbbreviation</code>	Gets the task group type abbreviation, which is used to create the task group unique directory name and also serves as the task group object's base name.
	<code>DisplayName</code>	Gets the user-visible name for the type of task groups created from the template.
	<code>ToolboxGroup</code>	Gets the toolbox group in which the task group template should be displayed.
	<code>ImageName</code>	Gets the image associated with the task group template.
	<code>Physics</code>	Gets the physics associated with the task group template.
	<code>AnalysisType</code>	Gets the analysis type associated with the task group template.

Class	Member	Description
	TaskGroupHelpId	Gets the help ID associated with the task group template.
	ToolTipHelpId	Gets the help ID associated with the tool tip for the task group.
	IsParametric	Gets a value indicating whether the task group consumes parameters.
	Solver	Gets the solver associated with the task group template.
	TaskTemplates	Gets the task templates that make up the task group. Tasks are grouped into task groups. For the normal case with no groups, a single-element list is returned.
	TaskNames	Gets the names of the tasks to create from the templates in TaskTemplates .
	ExcludedTaskTemplates	Gets a list of task templates that are not permitted to be instantiated in the task group using such commands as InsertComponentBefore or InsertComponentAfter .
	RequiredTaskTemplates	Gets a list of task templates that cannot be deleted from the task group.
	Attributes	Gets the attributes to apply to the created task group.
	PostCreationSteps	Gets the list of operations to perform after the task group is created.
	Visible	Gets a value indicating whether the task group should appear in the toolbar.
	IsBeta	Gets a value indicating whether the task group visibility is tied to the beta option.
	IsDeletable	Gets a value indicating whether the task group can be deleted.
	InternalObject	Gets the internal system template represented by the task group template.

Accessing Parameter APIs

As of ANSYS 18.0, Workbench parameter objects are exposed at the user level through [Automation API \(p. 173\)](#) wrappers in the namespace **Ansys.ACT.WorkBench.Automation.Workflows**. Simple parameter interactions are now fully supported through these wrappers:

```
task = ExtAPI.DataModel.Tasks[0]
p0 = task.Parameters[0]
value = p0.Value
newValue = value * 2
p0.Value = newValue
```

The following table lists the APIs that are available.

Class	Member	Description
IWBParameter	Name	The parameter name.

Class	Member	Description
	Caption	The parameter display text.
	Value	The parameter value for the current design point.
	Usage	The parameter usage.
Parameter	Name	The parameter name.
	Caption	The parameter display text.
	Value	The parameter value for the current design point.
	Usage	The parameter usage.
	InternalObject	The Workbench parameter data reference.
ParameterUsage	Input	A parameter whose value is to be used by the data model.
	Output	A parameter whose value is either based on an expression or provided directly by the data model.
	Unknown	The parameter usage could not be determined.

Accessing Property APIs

As of ANSYS 18.0, Workbench property values are exposed at the user level through [Automation API \(p. 173\)](#) wrappers in the namespace `Ansys.ACT.WorkBench.Automation.Workflows`. Simple property interactions are now fully supported through these wrappers:

```
task = ExtAPI.DataModel.Tasks[0]
p1 = task.Properties["PropertyOne"]
value = p1.Value
newValue = value * 2
p0.Value = newValue
```

The following table lists the APIs that are available.

Class	Member	Description
IWBProperty	Name	The property name.
	Value	The property value.
Property	Name	The property name.
	Value	The property value.

Accessing State-Handling APIs

To control state handling, you can access state values from the `Ansys.ACT.Interfaces.Common.State` enumeration.

The following table lists the APIs that are available.

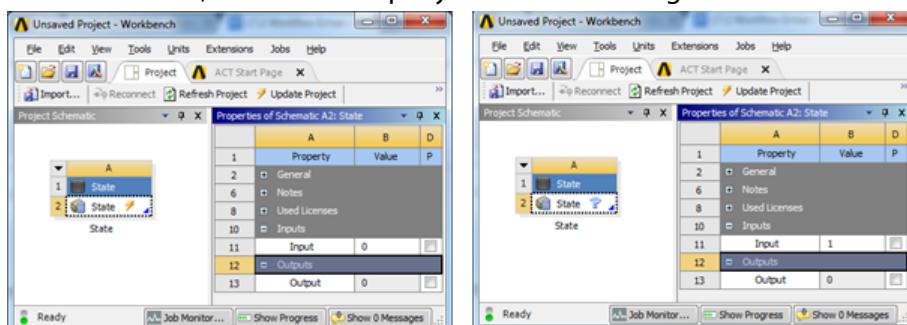
Class	Member	Description
<code>Ansys.ACT.Interfaces.Common.State</code>	<code>UpToDate</code>	The task is in an up-to-date state.
	<code>RefreshRequired</code>	The task requires a refresh to continue upstream changes.
	<code>OutOfDate</code>	The task is in an out-of-date state. This state is valid only in Mechanical.
	<code>UpdateRequired</code>	The task must be updated.
	<code>Modified</code>	The task has been modified. This state is valid only in Mechanical.
	<code>Unfulfilled</code>	Task requirements are unfulfilled.
	<code>Disabled</code>	The task is disabled.
	<code>Error</code>	The task is in error.
	<code>EditRequired</code>	Data entered for the task is invalid or incomplete and must be modified.
	<code>Interrupted</code>	Processing of the task is halted.
<code>UpstreamChangesPending</code>		Updating of the task cannot be completed until previous tasks are updated.
	<code>Unknown</code>	The state of the task could not be determined.

When defining the state callback on tasks, you can return the Framework-defined `ComponentState` instance. However, returning this instance is not recommended unless you must maintain backwards compatibility. Instead, return a two-entry list, where the first entry is an `Ansys.ACT.Interfaces.Common.State` enumeration value and the second entry is a quick-help string.

```
import clr
clr.AddReference('Ansys.ACT.Interfaces')
import Ansys.ACT.Interfaces

def status(task):
    if task.Properties['Inputs'].Properties['Input'].Value == 1:
        return [ Ansys.ACT.Interfaces.Common.State.Unfulfilled ,
                 'cannot enter the value of 1' ]
    else:
        return None #rely on the default framework-calculated state
```

When executed, this code sample yields the following result:



In addition, ACT provides a default state handler whenever the task does not provide a callback **<on-status>**. When any task property is invalid, ACT instructs the framework to use an unfulfilled state, as shown in the following examples of an XML definition file and IronPython script.

XML Definition File:

```

<extension version="1" name="StateValidity">
    <guid shortid="StateValidity">
        69d0155b-e138-4841-a13a-de12238c83f2
    </guid>
    <script src="main.py" />
    <interface context="Project">
        <images>images</images>
    </interface>
    <workflow name="MyWorkflow" context="Project" version="1">
        <tasks>
            <task name="StateValidity" caption="State Validity"
                icon="Generic_cell" version="1">
                <callbacks>
                    <onupdate>update</onupdate>
                    <onreset>reset</onreset>
                </callbacks>
                <property name="Valid" caption="Property Check"
                    control="integer" default="0" readonly="false"
                    needupdate="true" visible="true" persistent="true"
                    isparameter="true">
                    <callbacks>
                        <isvalid>valid</isvalid>
                    </callbacks>
                </property>
                <property name="Input" caption="Input" control="float"
                    default="0.0" readonly="false" needupdate="true"
                    visible="true" persistent="true"
                    isparameter="true" />
                <property name="Output" caption="Output" control="float"
                    default="0.0" readonly="true" visible="true"
                    persistent="true" isparameter="true" />
                <inputs>
                    <input/>
                </inputs>
                <outputs/>
            </task>
        </tasks>
        <taskgroups>
            <taskgroup name="StateValidity" caption="State Validity"
                icon="Generic" category="" abbreviation="State"
                version="1">
                <includeTask name="StateValidity" caption="State Validity"/>
            </taskgroup>
        </taskgroups>
    </workflow>
</extension>

```

IronPython Script:

```

import clr
clr.AddReference('Ansys.ACT.Interfaces')
import Ansys.ACT.Interfaces

def update(task):
    activeDir = task.ActiveDirectory
    extensionDir = task.Extension.InstallDir
    exeName = "ExampleAddinExternalSolver.exe"
    solverPath = System.IO.Path.Combine(extensionDir, exeName)

    inputValue = task.Properties["Inputs"].Properties["Input"].Value

    inputFileName = "input.txt"
    outputFileName = "output.txt"
    dpInputFile = System.IO.Path.Combine(activeDir, inputFileName)

```

```

dpOutputFile = System.IO.Path.Combine(activeDir, outputFileName)

#write input file
f = open(dpInputFile, "w")
f.write('input=' + inputValue.ToString(
    System.Globalization.NumberFormatInfo.InvariantInfo))
f.close()

exitCode = ExtAPI.ProcessUtils.Start(solverPath, dpInputFile,
                                      dpOutputFile)

if exitCode != 0:
    raise Exception ('External solver failed!')

#read output file

outputValue = None
f = open(dpOutputFile, "r")
currLine = f.readline()
while currLine != "":
    valuePair = currLine.split('=')
    outputValue = System.Double.Parse(valuePair[1],
                                       System.Globalization.NumberFormatInfo.InvariantInfo)
    currLine = f.readline()
f.close()

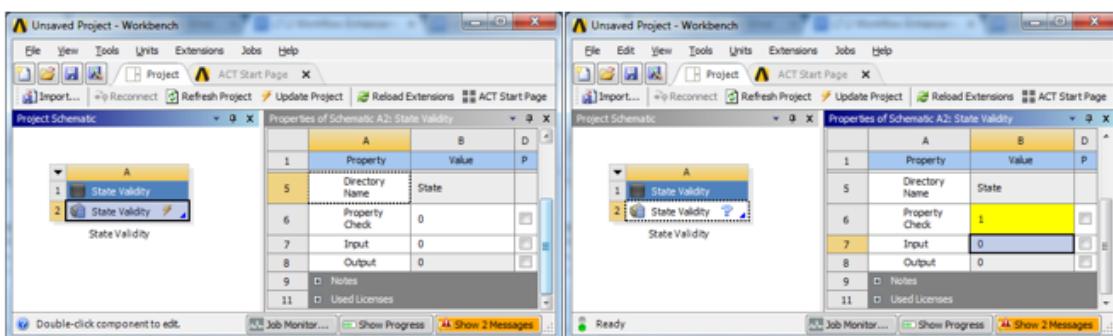
if outputValue == None:
    raise Exception("Error in update - no output value detected!")
else:
    task.Properties["Output"].Value = outputValue

def reset(task):
    task.Properties["Input"].Value = 0
    task.Properties["Output"].Value = 0

def valid(entity, property):
    if property.Value == 1:
        return False
    else:
        return True

```

Result:



Accessing Project Reporting APIs

ACT fully wraps the framework's reporting API. To generalize APIs across all ANSYS products (potential to adopt reporting for other products outside of Workbench), new interfaces have been defined. The following APIs are available:

Class	Member	Description
IReport	AddChild(item)	Adds a child report item to the report.
	AddChild(item, index)	Adds a child report item to the report.

Class	Member	Description
	RemoveChild(index)	Removes a child report item from the report.
	GetChild(index)	Obtains a child report item from the report.
	ChildCount	The number of children inside the report.
	FilePath	The report file path.
	ImageDirectoryPath	The image directory path for report images.
ProjectReport	AddReportImage(name)	Adds an image item to the report.
	InsertReportImage(name, index)	Inserts an image item into the report.
	AddReportLink(name)	Adds a link item to the report.
	InsertReportLink(name, index)	Inserts a link item into the report.
	AddReportSection(name)	Adds a section item to the report.
	InsertReportSection(name, index)	Inserts a section item into the report.
	AddReportText(name)	Adds a text item to the report.
	InsertReportText(name, index)	Inserts a text item into the report.
	AddReportTable(name)	Adds a table item to the report.
	AddReportTable(name, numberOfRows, numberOfColumns)	Adds a table item to the report, with the specified number of rows and columns.
	InsertReportTable(name, index)	Inserts a table item into the report.
IReportItem	Name	The name of the report item.
ProjectReportImage	ProjectReportImage(name)	Constructor.
	ProjectReportImage(name, sourceLocation)	Constructor.
	SourceLocation	The image location (file path).
ProjectReportLink	ProjectReportLink(name)	Constructor.

Class	Member	Description
	ProjectReportLink(name, resourceLocation)	Constructor.
	ResourceLocation	The linked path.
ProjectReportSection	ProjectReportSection()	Constructor.
	ProjectReportSection(name)	Constructor.
	AddChild(item)	Adds a child report item to the report section.
	AddChild(item, index)	Adds a child report item to the report section at the specified index.
	RemoveChild(index)	Removes a child report item from the section at the given index.
	GetChild(index)	Retrieves a child report item from the section at the given index.
	ChildCount	The number of children inside the report section.
	AddReportImage(name)	Adds an image item to the report.
	InsertReportImage(name, index)	Inserts an image item into the report section.
	AddReportLink(name)	Adds a link item to the report section.
	InsertReportLink(name, index)	Inserts a link item into the report section.
	AddReportSection(name)	Adds a section item to the report section.
	InsertReportSection(name, index)	Inserts a section item into the report section.
	AddReportText(name)	Adds a text item to the report section.
	InsertReportText(name, index)	Inserts a text item into the report section.
	AddReportTable(name)	Adds a table item to the report section.
	AddReportTable(name, numberOfRows, numberOfColumns)	Adds a table item to the report section, with the specified number of rows and columns.

Class	Member	Description
	InsertReport-Table(name, index)	Inserts a table item into the report section.
ProjectReportTable	ProjectReport-Table(name)	Constructor.
	ProjectReport-Table(name, numRows, numColumns)	Constructor.
	RowCount	The number of rows in the table.
	ColumnCount	The number of columns in the table.
	GetColumnName(index)	Retrieves the name for a column at the specified index.
	SetColumnName(index)	Sets the name for a column at the specified index.
	AddColumn(columnName)	Adds an empty column to the table using the supplied column name.
	AddColumn(columnName, index)	Adds an empty column to the table at the specified index using the supplied column name.
	AddColumn(columnName, index, columnValues)	Adds a column to the table at the specified index using the supplied column name and populated with the supplied values.
	RemoveColumn(index)	Removes a column at the specified index.
	AddRow(rowValues)	Adds a new row of values to the table.
	AddRow(rowValues, index)	Adds a new row of values to the table at the specified index.
	RemoveRow(index)	Removes a row from the table at the specified index.
	GetValue(rowIndex, columnIndex)	Retrieves a value from the table at the specified row-and-column coordinates.
	SetValue(rowIndex, columnIndex, value)	Sets a report table value in the table at the supplied row-and-column coordinates.

Class	Member	Description
	SetValue(rowIndex, columnIndex, content)	Sets a string value in the table at the supplied row-and-column coordinates.
	SetRowBackgroundColor(index, color)	Sets the background color for the row at the specified index.
	GetRowBackgroundColor(index)	Retrieves the current row background color for the row at the supplied index.
	SetRowForegroundColor(index, color)	Sets the foreground color for the row at the specified index.
	GetRowForegroundColor(index)	Retrieves the current row foreground color for the row at the supplied index.
	ShowHeaders	Indicates whether or not the column headers should be displayed.
	IsCollapsible	Indicates whether or not the table is collapsible.
ProjectReportTable-Value	ProjectReportTable-Value(content)	Constructor.
	ProjectReportTable-Value(content, iconSourceLocation, iconAlternateText)	Constructor.
	Content	The table value string.
	IconSourceLocation	The cell icon source path.
	IconAlternateText	The cell icon alternate text to use when the icon cannot be located.
ProjectReportText	ProjectReportText(text)	Constructor.
	ProjectReportText(name, text)	Constructor.
	Text	Gets or sets the report item's text.

Consider the following code sample:

```
import clr
clr.AddReference("Ansys.ACT.WorkBench")
import Ansys.ACT.WorkBench

def report(task, report):
    section = report.AddReportSection("My Custom ACT Task Report Content")
    text = section.AddReportText("TestText")
    text.Text = "Sample text from the data squares component"
    link = section.AddReportLink("TestLink")
```

```
link.ResourceLocation = r"http://www.ansys.com"
extDir = ExtAPI.ExtensionManager.CurrentExtension.InstallDir
imgDir = System.IO.Path.Combine(extDir, "images")
imgName = "logo-ansys.jpg"
imgPath = System.IO.Path.Combine(imgDir, imgName)
reportImgDir = report.ImageDirectoryPath
destImgPath = System.IO.File.Copy(imgPath,
                                    System.IO.Path.Combine(reportImgDir, imgName))
img = section.AddReportImage("TestImage")
img.SourceLocation = imgName

rowCount = 5
colCount = 3
table = section.AddReportTable("TestTable", rowCount, colCount)
for i in range(0, rowCount):
    evenRow = (i % 2 == 0)
    rowVals = []
    for j in range(0, colCount):
        table.SetValue(i, j, str(j + i*colCount))
    if evenRow:
        table.SetRowBackgroundColor(i, "Red")
        table.SetRowForegroundColor(i, "Blue")
    else:
        table.SetRowBackgroundColor(i, "Blue")
        table.SetRowForegroundColor(i, "Red")
table.SetColumnName(0, "Col 1")
table.SetColumnName(1, "Col 2")
table.SetColumnName(2, "Col 3")
```

When executed, the code sample generates the following report:

Reporting

My Custom ACT Task Report Content

TestText

Sample text from the data squares component

TestLink

TestImage

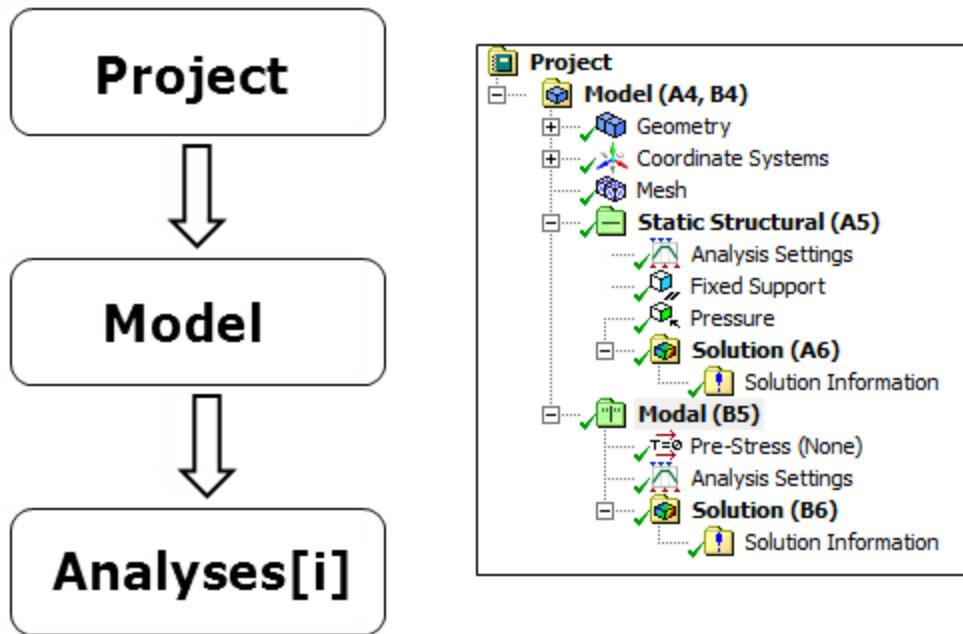


TestTable

Col 1	Col 2	Col 3
0	1	2
3	4	5
6	7	8
9	10	11
12	13	14

APIs for ANSYS Mechanical

This section describes some of the APIs provided by ACT for the customization of ANSYS Mechanical. These APIs provide access to all objects in the Mechanical tree (**Project**, **Model**, and **Analysis**), allowing you to manipulate objects and their properties.



Using the APIs, you can both create new objects and modify existing ones.

Note

When you create a new object, Mechanical initializes the object's property values in the **Details** window to the same default values used when you add an object via standard mouse-click operations. Some properties, such as scoping or the mass of a point mass, can be invalid until you enter a value.

For more information on ANSYS Mechanical APIs and their properties, refer to the [ANSYS ACT API Reference Guide](#).

Directly Accessing an Object

You can get to an object by programmatically navigating the Mechanical tree.

The root node of the tree is `ExtAPI.DataModel.Project`. Each tree node has children that you can access by calling the property `Children`. This returns all of the child nodes in the project.

You can also call nested properties, such as `Model`, `Geometry`, `Mesh`, or `Analyses`, to access all of the instances of a given object or tree level.

```
Connection = ExtAPI.DataModel.Project.Model.Children[3]
Mesh = ExtAPI.DataModel.Project.Model.Mesh
```

Handling Property Types

This section provides examples of how to handle the various property types used within ANSYS Mechanical.

Quantity: A unit-based value typically related to a physical quantity.

Example:

```
my_object.ElementSize = Quantity("0.1 [m]")
```

Numerical Value (float and integer): A property, such as **Relevance** or **Rate**, expecting a numerical value (a number for which a unit is not specified).

Example:

```
my_object.TetraGrowthRate = 2
```

Boolean: A property expecting a **True** or **False** value.

Example:

```
my_object.WriteICEMCFDFiles = True
```

Geometry Scoping: A property whose value is one or more geometric entities. In these cases, you must:

1. Create selection information and specify the IDs of the entities that you want to manipulate.
2. Assign this list of IDs to **Location** / **SourceGeometry** properties.

Example:

```
my_selection = ExtAPI.SelectionManager.CreateSelectionInfo(SelectionTypeEnum.GeometryEntities)
my_selection.Ids= [28,25]
```

```
My_object.Location = my_selection
```

Customizing the User Interface and Existing Toolbars

The API **UserInterface** provides methods to control the ACT-based GUI of Mechanical. The API is available using the following entry point:

```
ExtAPI.UserInterface
```

This API allows you to hide or gray-out ACT-based features and customize existing toolbars. It cannot be used to create new items such as toolbars. The API **UserInterface** provides access only to existing UI elements.

ExtAPI.UserInterface.Toolbars is a collection of toolbar objects. Each object has fields to access **Name/Caption/Visibility/Child** Entries. Each child has the following properties: **Caption**, **Enabled**, **Entries**, **EntryType**, **Name**, and **Visible**.

The Boolean fields **Visible** and **Enabled** can be set to "show" or "hide" so that you can control the availability of the buttons depending on the current context.

Working with Command Snippets

The API **CommandSnippet** provides for defining a MAPDL script that you want to invoke during the preprocessing, solution solving, or postprocessing phase of your analysis. Use the function **AddCommandSnippet()** to insert a new child command snippet into the project tree:

```
sol = ExtAPI.DataModel.Project.Model.Analyses[0].Solution
cs = sol.AddCommandSnippet()
cs.Input = "/COM, New input"
cs.AppendText("\n/POST1")
```

You can also use `ImportTextFile(string)` to import content from a text file or use `ExportTextFile(string)` to export a command snippet to a text file.

Manipulating Fields for Boundary Conditions

ACT provides an API that enables you to manipulate **boundary condition fields**, managing the various time-, space-, or frequency-dependent values of boundary conditions during a simulation. A field has one or more inputs and a single output.

Mathematically speaking:

```
output = F(input1, input2, ...)
```

Where **F** is the boundary condition field.

As with other types of ACT functionality available in Mechanical, the API-based actions for editing boundary condition fields mirror the actions that can be performed manually in the Mechanical GUI. The ACT API provides you with the capabilities of:

- Setting variable definition types (called “modes” in Mechanical)
- Setting input loading data definitions, which includes setting discrete input and output values

Input and Output Variables

The inputs and output are represented by objects of the type **Variable**. Input variables represent the various input values over the course of the simulation. Output variables represent the various values of the boundary conditions with regard to the input values.

For example, assume **time** is an input variable representing the various time values (in seconds) and **magnitude** is an output variable representing the magnitude of a force over time.

Variable Definition Types

The ACT API enables you to set input and output variables to one of three variable definition types (or modes):

Discrete

The variable contains a discontinuous set of values. By default, most variables are initially defined as discrete.

Formula

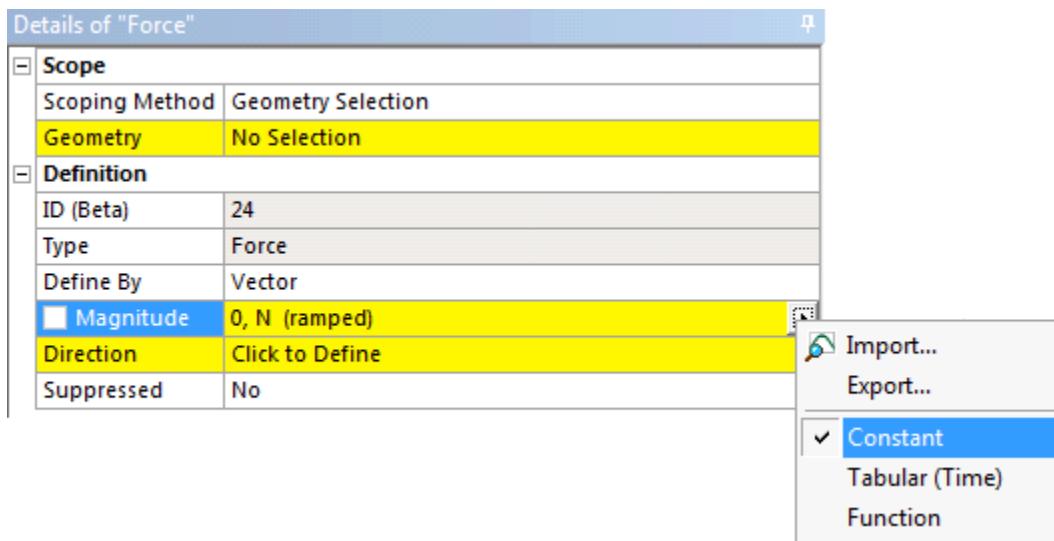
The variable is a continuous function depending on inputs, such as **time*10**.

Free

Determines the selection’s freedom to move along the specified plane. This variable definition type (mode) is available only for certain boundary conditions such as displacements. It is not available for force or pressure.

Input Loading Data Definitions

For input variables, each variable definition type (or mode) corresponds to one of the input loading data definition options available in the Mechanical **Details** view. You can determine the loading data definition options available for a given input by checking its drop-down in the Mechanical GUI.



Input loading data definitions are ranked in terms of complexity:

Complexity	Variable Definition Type	Input Loading Data Definition	Description
1	Discrete	Constant (ramped)	Two values, with the first value always 0.
2	Discrete	Constant (time-stepping)	One value.
3	Discrete	Tabular	Values listed in tabular format.
4	Formula	Function	Values generated by the expression assigned to the variable.
NA	Free	NA	<p>Indicates that the selection can move freely along the specified plane. (A value of 0 indicates that movement is fully constrained.)</p> <p>Not supported by all types of boundary conditions.</p>

When you use the ACT API to define boundary condition fields, ACT automatically opts for the least complex loading data definition that is applicable to the input. For example, if a given input could be defined as **Constant (ramped)** but you execute commands defining it as **Tabular**, ACT defines the input as **Constant (ramped)**.

Setting the Variable Definition Type

The property `variable.DefinitionType` provides both get and set capabilities, enabling you to set input variables to one of three variable definition types (which also specifies a corresponding input loading data definition) and then verify the variable type and definition in the **ACT Console**.

Note

You can also change the variable definition type to **Discrete** or **Free** by using the property `variable.DiscreteValues` as described in [Setting Discrete Values for Variables \(p. 200\)](#).

This example consists of the following steps:

[Creating a Displacement and Verifying Its Variable Definition Types](#)

[Changing the Y Component from Free to Discrete and Verifying](#)

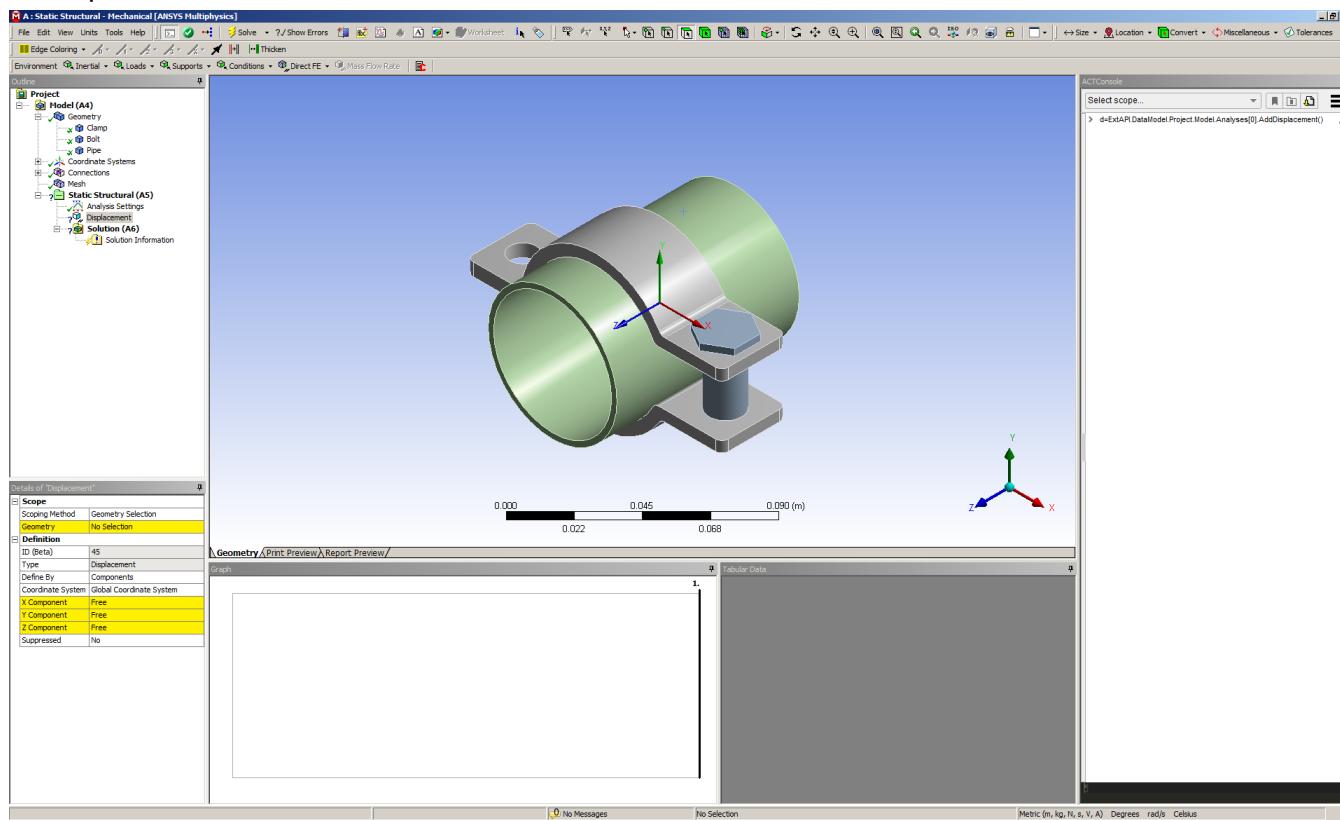
[Changing the Y Component Back to Free and Verifying](#)

Creating a Displacement and Verifying Its Variable Definition Types

You start by creating a displacement:

```
d=ExtAPI.DataModel.Project.Model.Analyses[0].AddDisplacement()
```

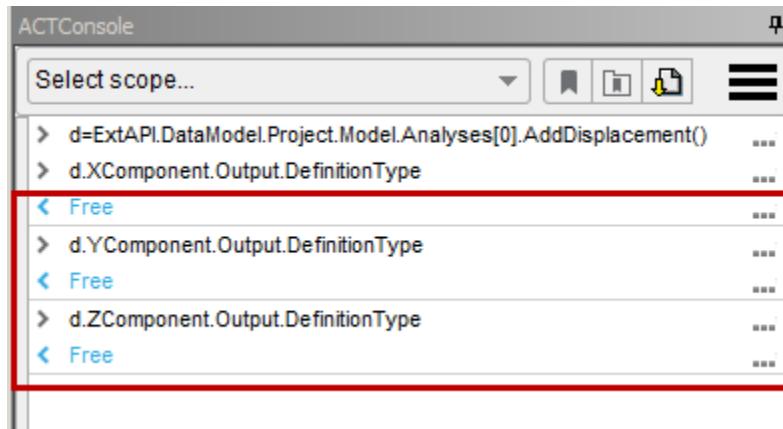
In Mechanical, you can see that by default, the variable definition types for the displacement's X, Y, and Z components are set to **Free**.



You can verify this by executing the following commands in the **ACT Console**, one at a time.

```
d.XComponent.Output.DefinitionType
d.YComponent.Output.DefinitionType
d.ZComponent.Output.DefinitionType
```

In the following image, each of the variable's components has a variable definition type of **Free**.



Note

Even though the query `d.XComponent.Output` would produce the same output in the console when the component is **Free**, be aware that it returns an instance of the **Variable** class. This object is then converted into a string by the **ACT Console**. Appending `.DefinitionType` to the query consistently returns the actual enum value, whatever the mode.

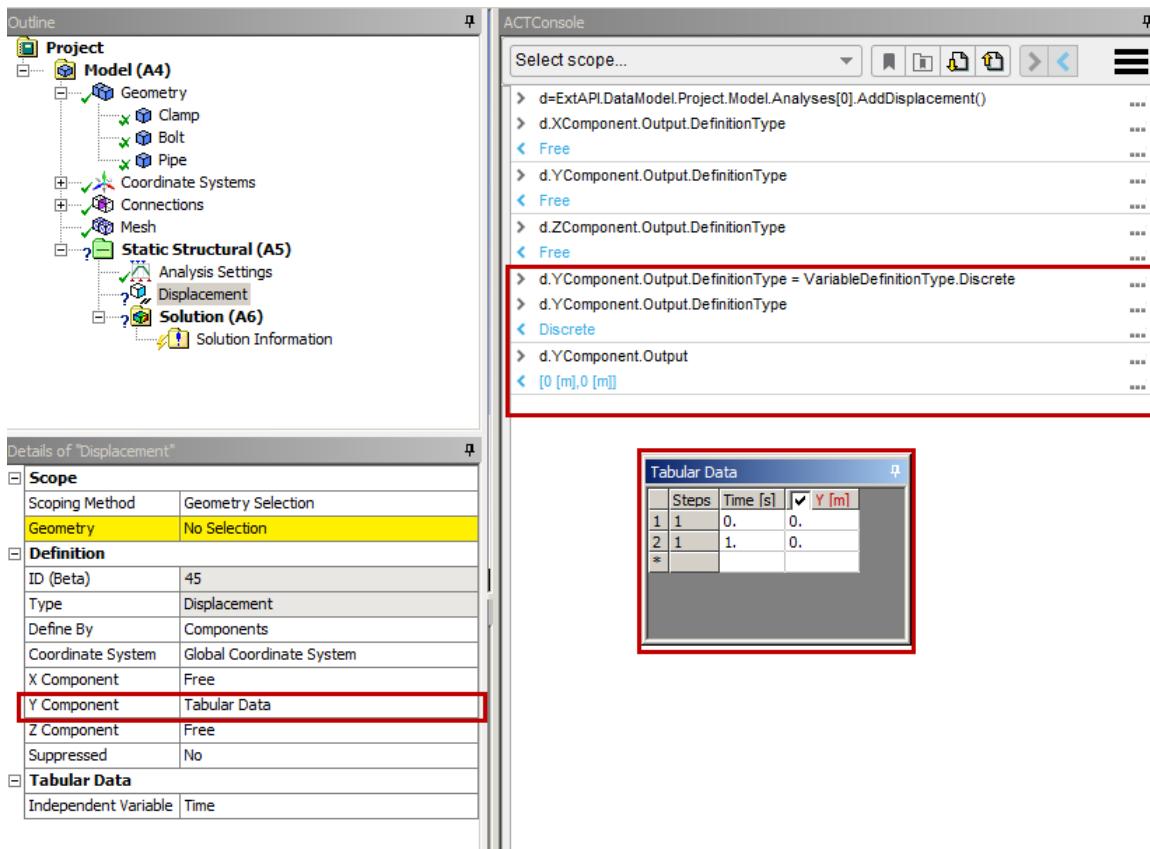
Changing the Y Component from Free to Discrete and Verifying

Next, you change the variable definition type for the Y component from **Free** to **Discrete** and then verify the definition type change and the output values in the **ACT Console**:

```
d.YComponent.Output.DefinitionType =VariableDefinitionType.Discrete
d.YComponent.Output.DefinitionType
d.YComponent.Output
```

In the following image, you can see that:

- The **Y Component** is set to **Tabular Data** because tabular data is the least complex discrete loading data definition that is applicable to this input.
- The **Tabular Data** window is now populated with output values of **0** and **0**. You can verify in the **ACT Console** that the output values are **0** and **0**.
- You can verify in the **ACT Console** that the variable definition type is set to **Discrete**.

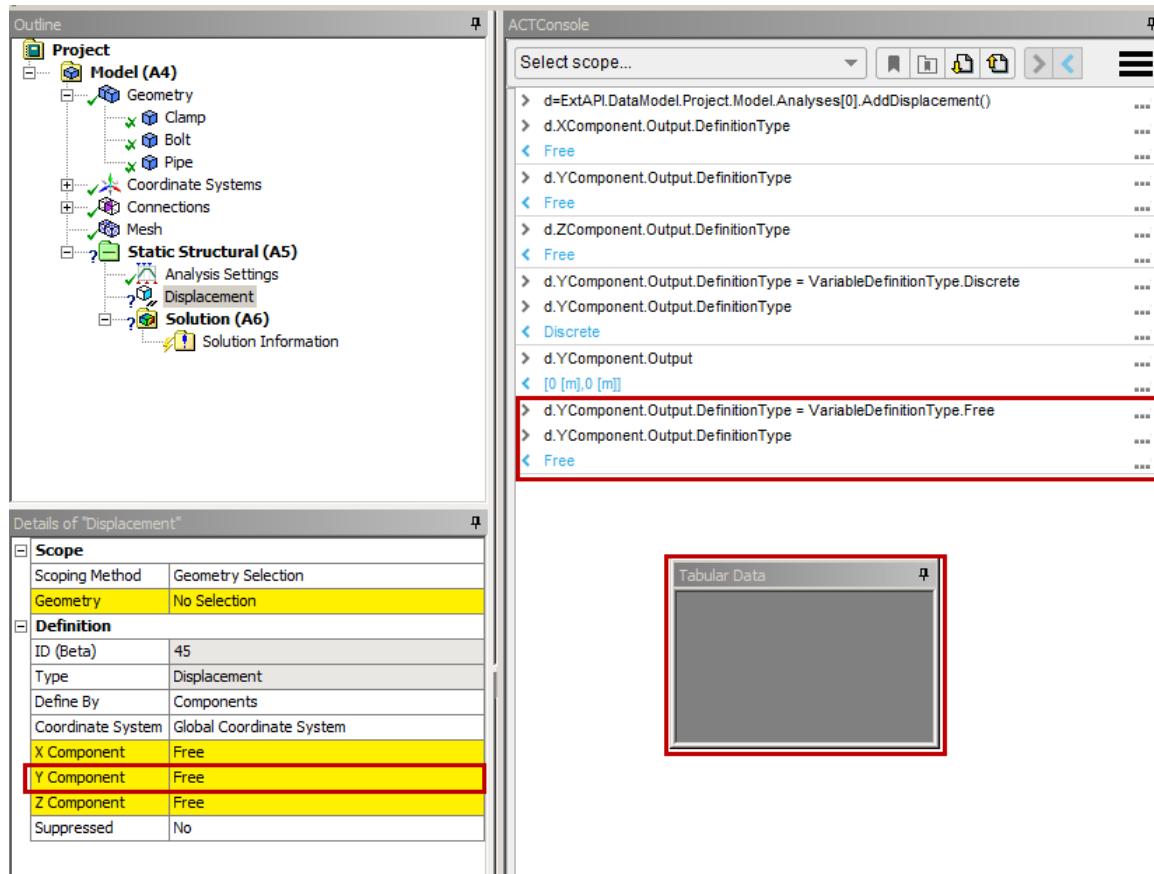


Changing the Y Component Back to Free and Verifying

Finally, you change the variable definition type for the Y output from **Discrete** back to **Free** and verify the change in the **ACT Console**:

```
d.YComponent.Output.DefinitionType = VariableDefinitionType.Free
d.YComponent.Output.DefinitionType
```

In the following image, you can see that the **Y Component** is set back to **Free** and the **Tabular Data** window is now empty.



Setting Discrete Values for Variables

The property **Variable.DiscreteValues** provides both **get** and **set** capabilities for both input and output variables. Behavior of these capabilities varies according to whether it is being performed on an input or an output variable.

Getting and Setting Discrete Values for an Input Variable

Getting Discrete Input Values

You can get the discrete input values for a given input variable. For example, a force in a one-step analysis looks similar to the following image by default. The two rows in the **Tabular Data** window correspond to the discrete values of the single input variable, with **0s** as the start time of the first step and **1s** as the end time of the last step.



Note

With a three-step analysis, you'd see four values by default: the start time and three end times (one end time for each step).

Setting Discrete Input Values

You can change discrete values for a given input variable. For example, the following code sample would add a discrete value, inserting a value of **t=0.5s** without defining an additional step in the analysis. In the following image, you can see that:

- There is a new row for **t=0.5**.
- The output cell for row **t=1** has a yellow background. This is because a value has not been set for the output variable.



Removing Discrete Values

You can also remove a discrete value, deleting a row by defining a shorter list of values for the input variable. In the following image, you can see that:

- You've specified a list with a single value, indicating that the other rows should be removed.
- Although a single row is specified, two rows are still showing in the **TabularData** window. The value **=0** is actually a repeat of the value in the row above and does not correspond to an actual value stored for the variable. This is because the tabular data in Mechanical always displays a row for time values that correspond to step end times.



Getting and Setting Discrete Values for an Output Variable

Output variables are not limited to the **Discrete** variable definition type, so the behavior when getting and setting outputs is slightly different from that of inputs.

Getting Discrete Output Values

The following table shows how the `get` capability behaves for the different variable definition types.

Variable Definition Type	Behavior
Discrete	Same as described earlier for getting discrete input values.
Free	For C#, the property returns <code>null</code> . For Python, the property returns <code>None</code> .
Formula	The property returns the series of variable values.

Setting Discrete Output Values

The following table shows how the `set` capability behaves according to the content of the list provided to the property.

List Content	Behavior
None or null	The variable is switched to the Free definition type if it is supported by the boundary condition. Otherwise, an error is issued.
Single Quantity object	The variable is switched to the Constant definition type if supported in the GUI. Otherwise, Constant (ramped) can be chosen. If neither of those two types are supported, Tabular Data is the default.
Two Quantity objects, with the first value being 0	The variable is switched to Constant (ramped) if supported in the GUI. Otherwise, Tabular Data is the default.
As many Quantity objects as Discrete values for the input variables	The variable is switched to Tabular Data .
Does not fall into any of the above categories	An error is issued, indicating that you should either: <ul style="list-style-type: none"> Provide the appropriate number of values.

List Content	Behavior
	<ul style="list-style-type: none"> Change the number of input values first, as the input value count is used to determine the row count of the tabular data. Output values must then comply with this number of rows. <p>For a sample message, see the following image.</p>

```
> f.Magnitude.Output.DiscreteValues = [Quantity('1 [N]'), Quantity('10 [N]'), Quantity('5 [N]')]
✖ Expected 2 values but 3 were found. Consider changing the variable's input discrete values.
Parameter name: value
```

Setting Variable Definition Type by Acting on Discrete Values

Because values are interconnected with variable definitions, you can use the property **Variable.DiscreteValues** as an alternate way of switching your variable definition type. To set a variable to a particular variable type, you set the values that are consistent with that variable type. Examples follow:

- To set an input variable to **Tabular Data**, assign values in tabular format:

```
d.XComponent.Output.DiscreteValues = [Quantity("1 [m]"), Quantity("10 [m]")]
```

- To set a variable to **Free**, assign **None** to its **DiscreteValues**:

```
d.XComponent.Output.DiscreteValues = None
```

Mechanical Worksheets

This section describes how to use the ACT API to automate the creation of worksheets in Mechanical. ACT enables you to create and populate worksheets by entering commands into the **ACT Console** and executing them in Mechanical. For more information, see [ACT Console \(p. 251\)](#).

The following examples are used to illustrate worksheet capabilities in ACT:

[Named Selection Worksheets](#)
[Mesh Order Worksheets](#)
[Layered Section Worksheets](#)
[Bushing Joint Worksheets](#)

Named Selection Worksheets

This section describes how to use the ACT API to automate the creation of a named selection worksheet in Mechanical. It assumes that the **Scoping Method** for your named selection is already set to **Worksheet**.

ACT supports all available actions for named selection worksheets, as described in [Specifying Named Selections Using Worksheet Criteria](#) in the *ANSYS Mechanical User's Guide*. This example focuses on the **Add Row** action to add new geometric entities to a named selection.

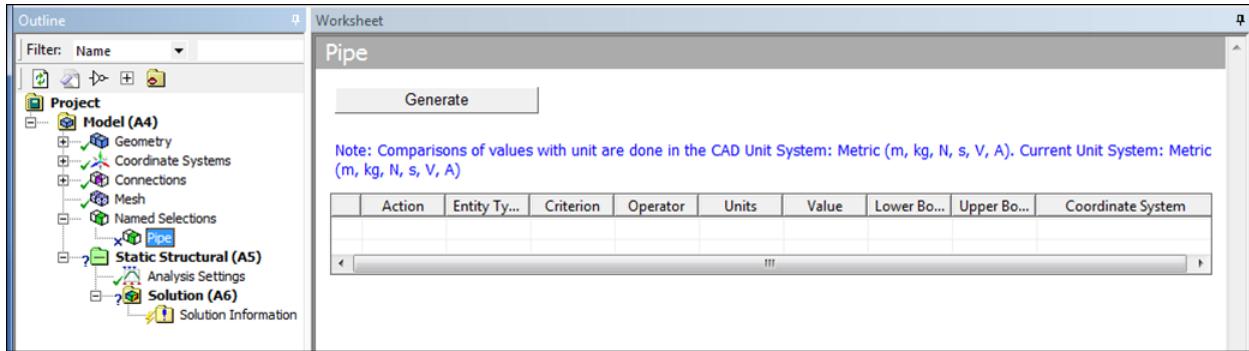
Creating the Named Selection Worksheet

To create the named selection worksheet, execute the following commands in the **ACT Console**:

```
model = ExtAPI.DataModel.Project.Model
sel = model.AddNamedSelection()
sel.Name = "Pipe"
selws = ExtAPI.SelectionManager.CreateSelectionInfo(SelectionTypeEnum.WorksheetSpecific)
```

```
sel.Location = selws
pipews = sel.Location
```

This creates the named selection object, renames it **Pipe**, and creates the worksheet for your named selection.



Adding New Rows

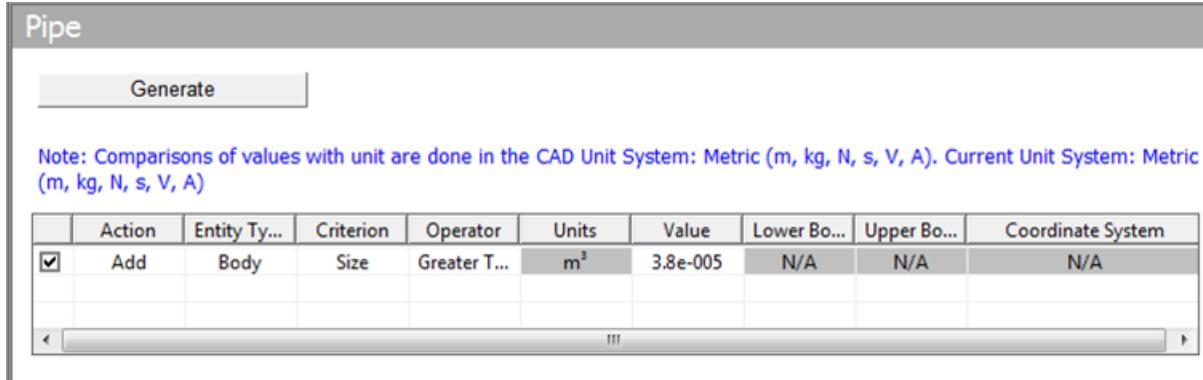
Data in the named selection worksheet defines the criteria based on geometric or meshing entities. Each row of the worksheet performs a calculation for the specified criteria.

The command **AddRow** adds worksheet rows according to the standard behavior of the **Add** option in Mechanical named selection worksheets. It adds the information defined in the current row to information in the previous row, provided that the **Entity Type** is the same for both rows.

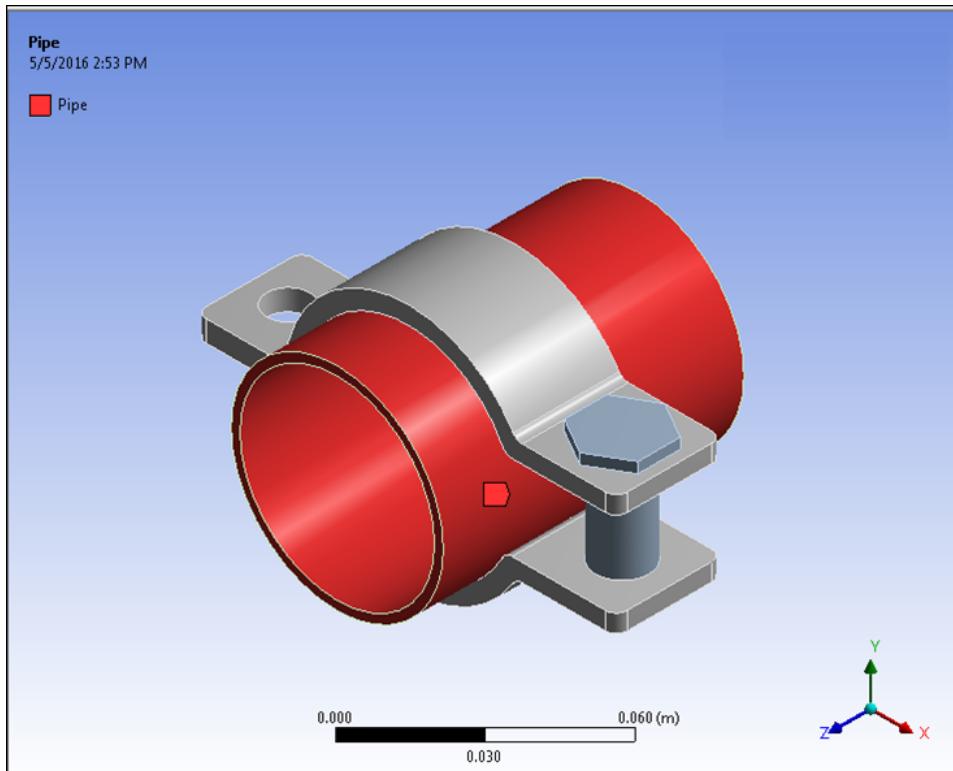
To add the first row in your worksheet, you execute these commands:

```
pipews.AddRow()
pipews.SetEntityType(0,NamedSelectionWorksheetEntityType.Body)
pipews.SetCriterion(0,NamedSelectionWorksheetCriterion.Size)
pipews.SetOperator(0,NamedSelectionWorksheetOperator.GreaterThan)
pipews.SetValue(0,3.8e-5)
pipews.Generate()
```

This example uses set methods to define a **Body** with a **Size** value greater than **3.8e-5**.



The method **Generate** generates the body. After executing this method, the pipe body is selected in the **Graphics** view. In **Details**, the property **Total Selection** is set to **1 Body**.



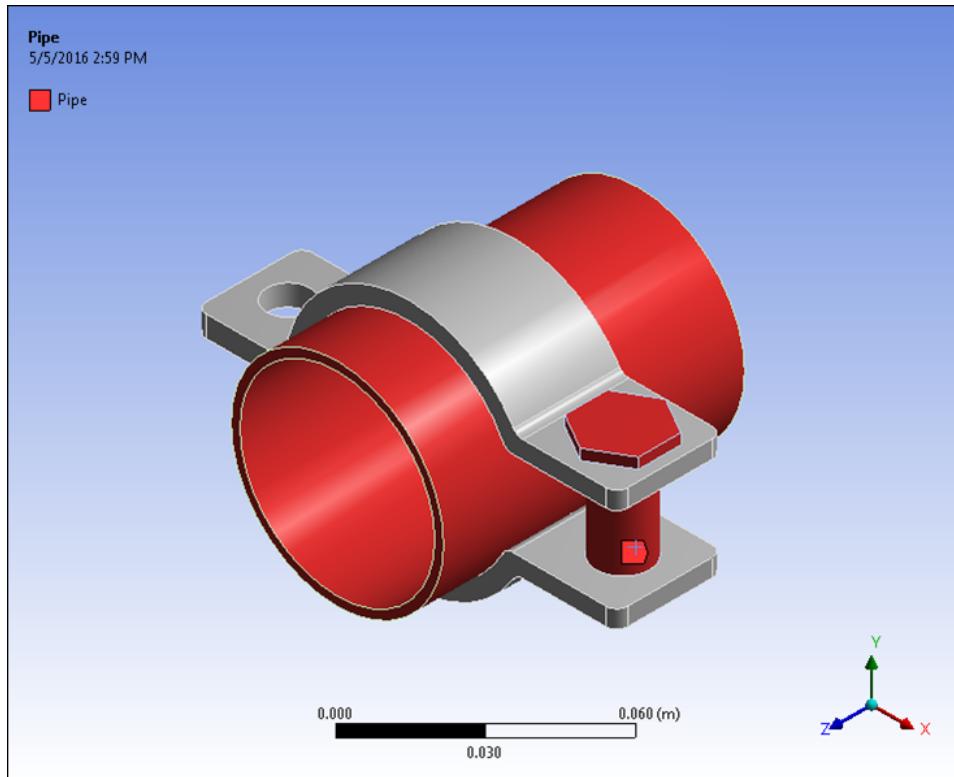
Next, you define and generate another body:

```
pipews.AddRow()
pipews.SetEntityType(1,NamedSelectionWorksheetEntityType.Body)
pipews.SetCriterion(1,NamedSelectionWorksheetCriterion.Size)
pipews.SetOperator(1,NamedSelectionWorksheetOperator.LessThan)
pipews.SetValue(1,8e-6)
pipews.Generate()
```

In this row, set methods define a **Body** with a **Distance** value less than **3.8e-6**. When there are no locally defined coordinate systems, you don't need to set the **Coordinate System** value. It is set to **Global Coordinate System** by default.

Pipe									
<input type="button" value="Generate"/> <i>Note: Comparisons of values with unit are done in the CAD Unit System: Metric (m, kg, N, s, V, A). Current Unit System: Metric (m, kg, N, s, V, A)</i>									
	Action	Entity Ty...	Criterion	Operator	Units	Value	Lower Bo...	Upper Bo...	Coordinate System
<input checked="" type="checkbox"/>	Add	Body	Size	Greater T...	m ³	3.8e-005	N/A	N/A	N/A
<input checked="" type="checkbox"/>	Add	Body	Distance	Less Than	m	3.8e-006	N/A	N/A	Global Coordinate System

After executing the method **Generate**, the bolt body is now selected in the **Graphics** view, along with the pipe body. In **Details**, the property **Total Selection** is set to **2 Bodies**.



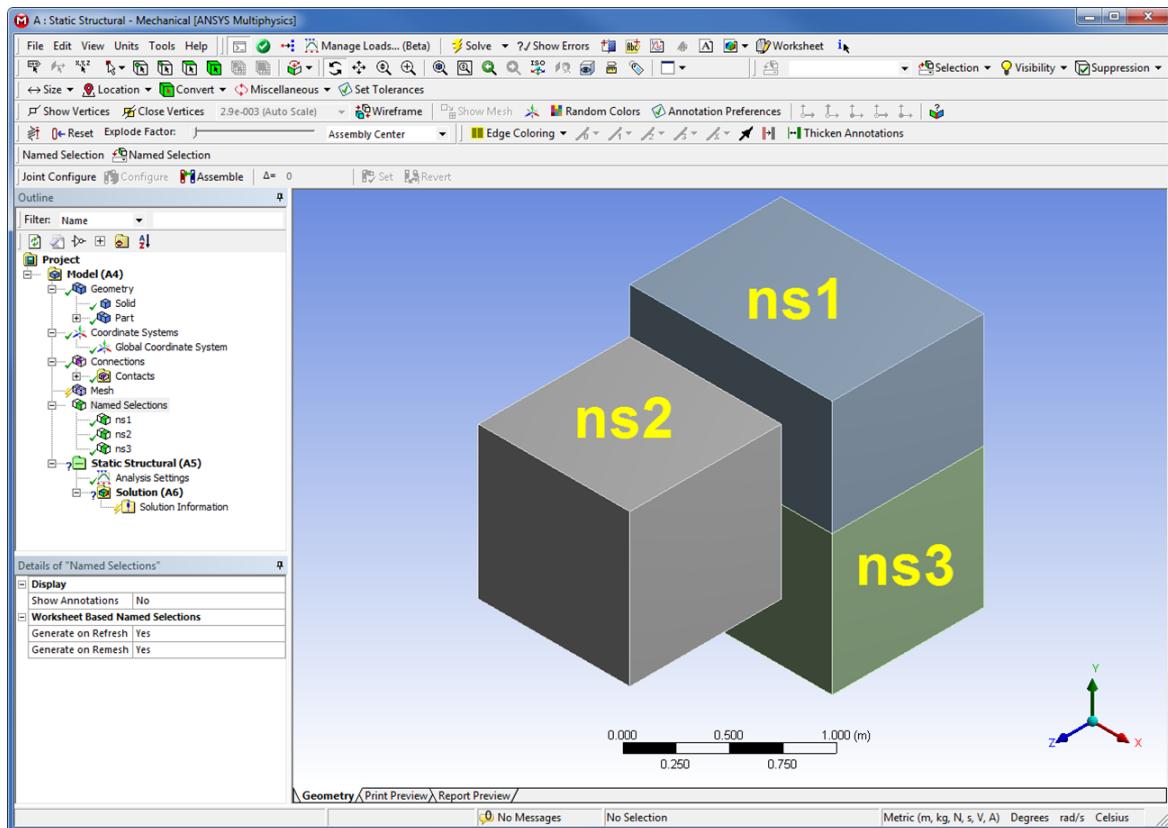
Mesh Order Worksheets

This section describes how to use the ACT API to add rows to an existing mesh worksheet, specifying the order in which meshing steps are performed in Mechanical.

ACT supports most of the functionality available for mesh worksheets, which is described in [Using the Mesh Worksheet to Create a Selective Meshing History](#) in the *ANSYS Meshing User's Guide*. The following exceptions apply:

- The named selections must have **Entity Type** set to **Body**. Other entity types are not supported.
- The **Start Recording** and **Stop Recording** buttons are not supported.

This section assumes that you have already defined two or more named selections in Mechanical. In your sample Static Structural analysis, you've defined three named selections, as shown in the following figure:

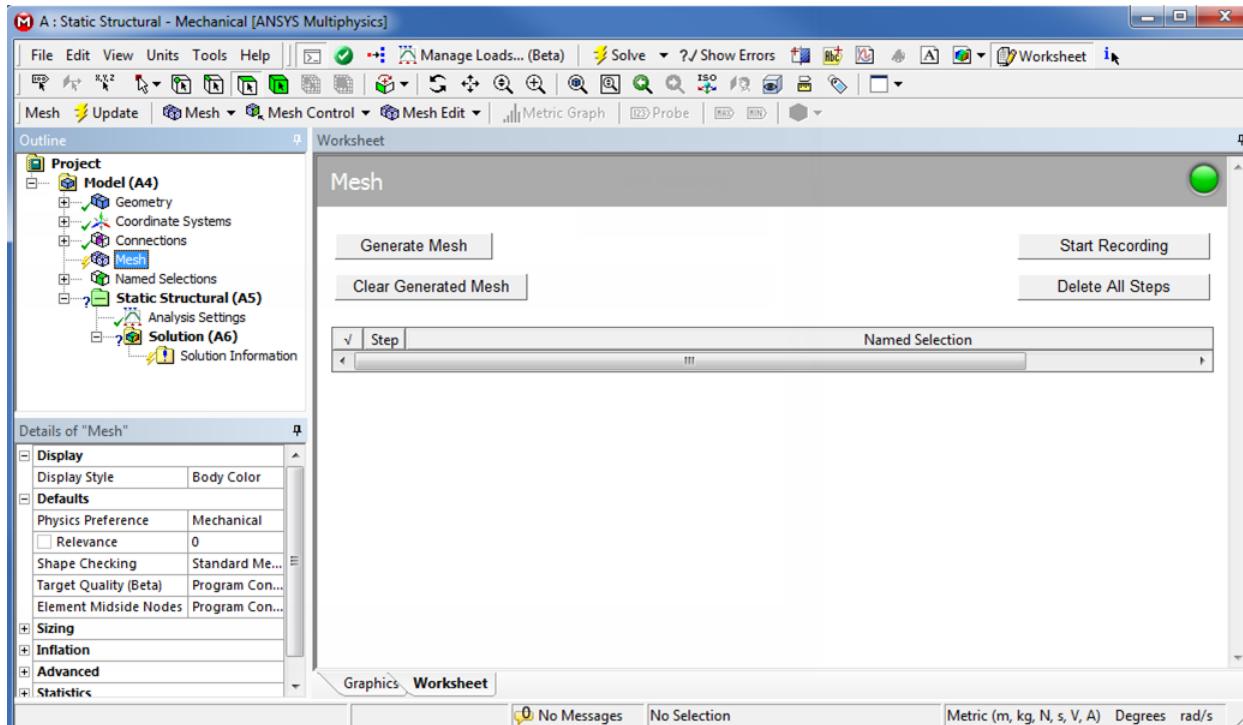


Displaying the Mesh Worksheet

To display the mesh worksheet, perform the following steps:

1. Select the **Mesh** node in the tree.
2. Click the **Worksheet** toolbar button.

The empty worksheet is displayed in the **Worksheet** tab.



Specifying the Mesh Worksheet and Named Selections

Next, you need to define variables for the mesh worksheet and the named selections to use in the worksheet rows:

```
model = ExtAPI.DataModel.Project.Model
msh = model.Mesh
mws = msh.Worksheet
nsels = model.NamedSelections
ns1 = nsels.Children[0]
ns2 = nsels.Children[1]
ns3 = nsels.Children[2]
```

Adding New Rows

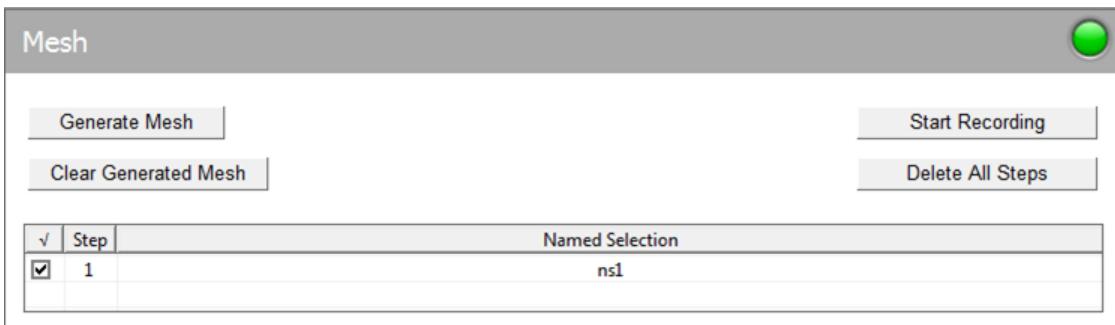
Finally, you add new rows to the mesh worksheet. Each row in the worksheet corresponds to a step in the meshing sequence.

To add the first row to your worksheet, you execute these commands:

```
mws.AddRow()
mws.SetNamedSelection(0,ns1)
mws.SetActiveState(0,True)
```

- The command **AddRow** adds an empty row.
- The command **SetNamedSelection** sets the **Named Selection** value to your named selection object **ns1**.
- The command **SetActiveState** sets the property **Active State** at the row index (enables the row, as indicated by the check mark in the left column).

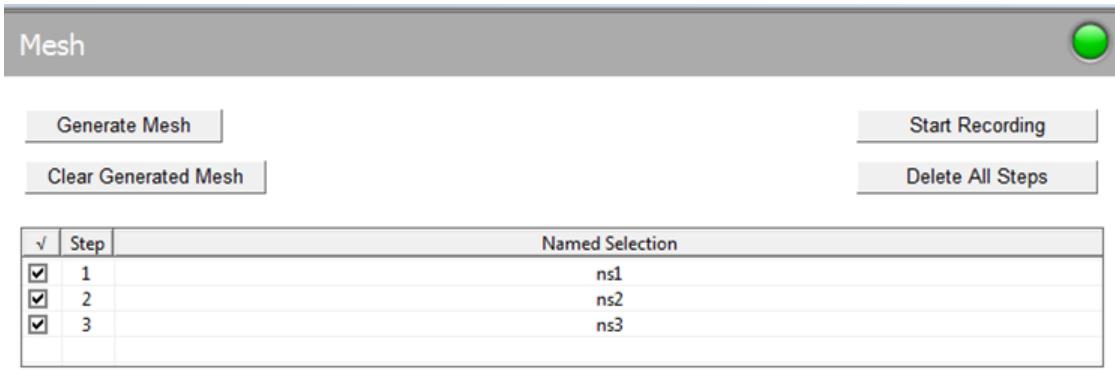
The following image shows the mesh worksheet after the execution of these commands.



Next, you add rows for your other two named selections, **ns2** and **ns3**. Note that for **ns3**, you use the command **SetNamedSelectionID** to set the named selection.

```
mws.AddRow()
mws.SetNamedSelection(1,ns2)
mws.SetActiveState(1,True)
mws.AddRow()
mws.SetNamedSelectionId(2,ns3.Id)
mws.SetActiveState(2,True)
```

The following image shows the mesh worksheet after the two new rows have been added:

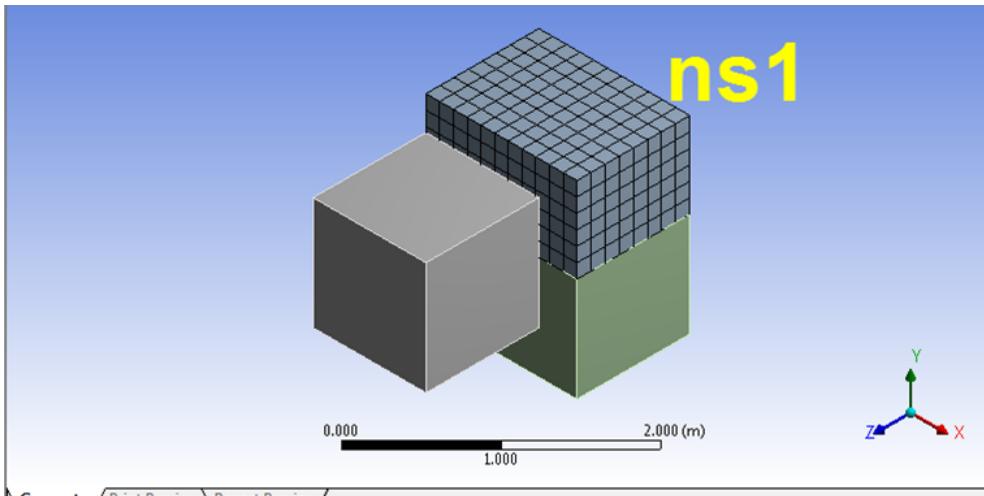


Meshing the Named Selections

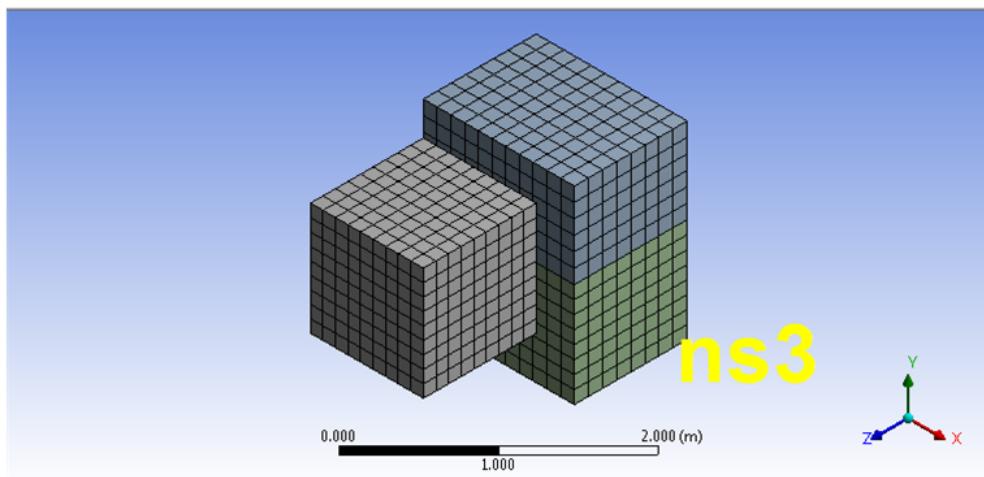
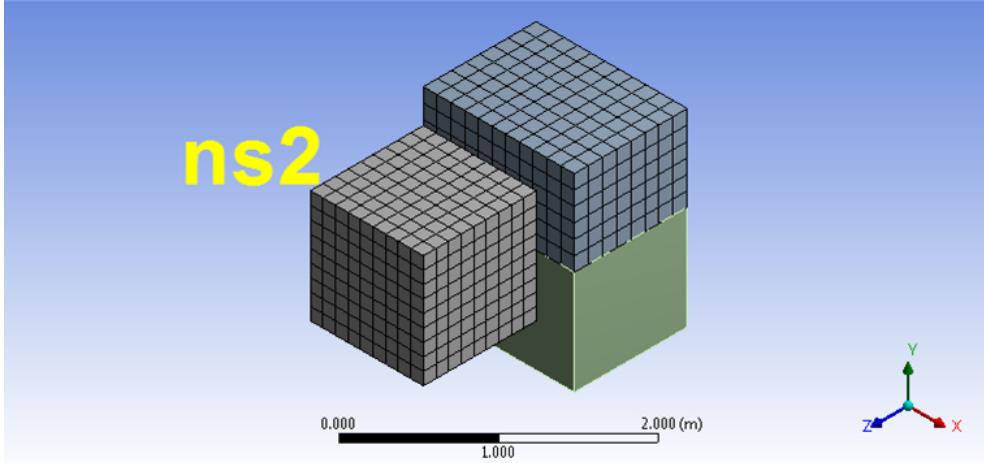
When you check the **Graphics** tab, you can see that no mesh has been generated yet. To generate the mesh for the named selections in the mesh worksheet, execute the following command:

```
mws.GenerateMesh()
```

When you execute this command, the steps are processed one by one in the order specified by the worksheet. For each step, the bodies identified by the named selection are meshed using the meshing controls applied to them. By watching the mesh generation in the **Graphics** tab, you can see the order in which each body is meshed.



Geometry Print Preview Report Preview



Additional Commands

Clear the Generated Mesh

To clear the generated mesh, execute the following command:

```
mws.ClearGenerateMesh()
```

Delete a Row

To delete a single row, execute the following command:

```
void DeleteRow(int index)
```

Delete All Rows

To delete all rows, execute the following command:

```
mws.DeleteAllRows()
```

Get Row Count

To display the row count in the **ACT Console** command history, execute the following command:

```
mws.RowCount
```

Layered Section Worksheets

This section describes how to use the ACT API to access the worksheet for a layered section definition in Mechanical. It assumes that you've already created a layered section and set the property **Layers** to **Worksheet**.

Displaying the Layered Section Worksheet

To display the layered section worksheet in Mechanical, perform the following steps:

1. In the **Project** tree, expand the **Geometry** node and select the **Layered Section** node.
2. In the details of the layered section, select the **Layers** property.
3. To the right of the **Worksheet** selection for this property, click the right arrow and select **Worksheet**.

The layered section worksheet is displayed in the **Worksheet** tab.

Getting the Layered Section Worksheet and Its Properties

To get to the layered section worksheet object in the **ACT Console**, enter:

```
mymodel = ExtAPI.DataModel.Project.Model
geo = mymodel.Geometry
ls = geo.Children[1]
lsws = ls.Layers
```

Getting and Displaying the Material Property

To get and display the material property for layer index 2, enter:

```
lsmat = ls(ws).GetMaterial(2)
lsmat
```

Assume that **Structural Steel** displays as the material property.

Getting and Displaying the Thickness Property

To get and display the thickness property for layer index 2, enter:

```
lsthick = ls(ws).GetThickness(2)
lsthick
```

Assume that **0.00253999746** displays as the thickness property.

To get and display the thickness property for a different layer index, such as layer index 3, enter:

```
ls(ws).GetThickness(3)
```

Assume that **0.004** displays as the thickness property.

Getting and Displaying the Angle Property

To get and display the angle property for layer index 2, enter:

```
lsang = lsws.GetAngle(2)  
lsang
```

Assume that **45** displays as the angle property.

Setting and Displaying the Thickness Property

To set the thickness property for layer index 2 to **0.0999999** and display this value, enter:

```
lsthick = lsws.SetThickness(2,0.0999999)  
lsthick = lsws.GetThickness(2)  
lsthick
```

The thickness property then displays as **0.0999999**.

Setting and Displaying the Angle Property

To set the angle property for layer index 2 to **22.5** and display this value, enter:

```
lsang = lsws.SetAngle(2,22.5)  
lsang = lsws.GetAngle(2)  
lsang
```

The angle property then displays as **22.5**.

Bushing Joint Worksheets

This section describes how to use the ACT API to access the bushing joint worksheet for Mechanical. A bushing joint worksheet defines stiffness and damping coefficients via symmetric matrices. This section assumes that you've already created a bushing joint worksheet.

Displaying the Bushing Joint Worksheet

To display the worksheet for a bushing joint in Mechanical, perform the following steps:

1. In the **Project** tree, expand the **Connections** node and then the **Joints** node.
2. Select the node for the bushing joint.
3. To the right of the **Worksheet** selection for this property, click the right arrow and select **Worksheet**.

The worksheet for the bushing joint displays in the **Worksheet** tab. The size of the matrices for stiffness and damping coefficients are fixed, which means that rows and columns cannot be added or deleted. Another aspect of these square matrices (6x6) is that they are both symmetric about their main diagonals. To ensure that these matrices are always symmetric, you cannot enter values into their upper right corners.

Getting the Bushing Joint Worksheet and Its Properties

To get to the bushing joint worksheet object in the **ACT Console**, enter:

```
mymodel = ExtAPI.DataModel.Project.Model  
joints = mymodel.Connections.Children[1]
```

```
bushing = joints.Children[0]
bws = bushing.BushingWorksheet
```

Getting the Value for a Specified Unit

The ACT API for a bushing joint worksheet enforces the fixed matrices by limiting the range of indices that can be specified to set or get the value from a particular cell. The row indices are zero-based.

For example, to get the coefficient for the stiffness per unit Y at row index 1, enter:

```
bws.GetBushingsStiffnessPerUnitY(1)
```

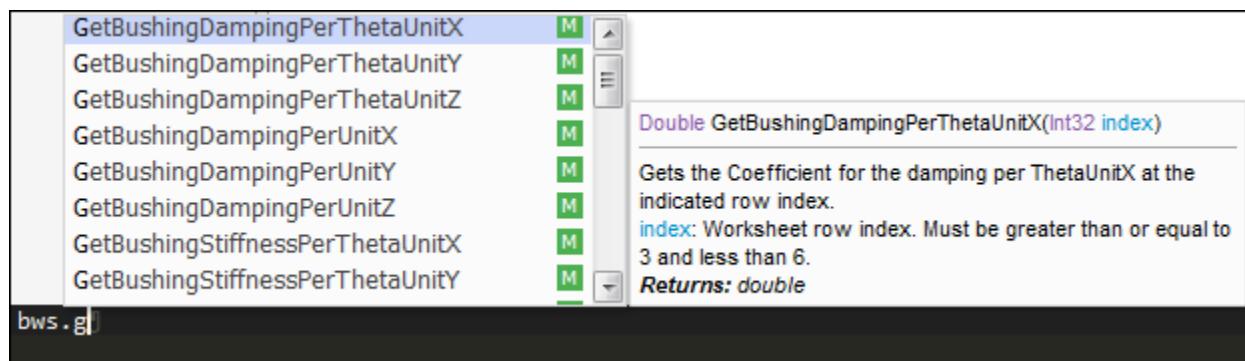
Assume that **136073.550978543** displays as the stiffness per unit Y.

Now, assume you've entered the following:

```
bws.GetBushingsStiffnessPerUnitZ(0)
```

The given index of 0 (zero) produces an error because that cell lies above the main diagonal. As indicated by the error message, the valid range of indices is equal to 2 and less than or equal to 5.

The following image displays some of the many methods available for getting coefficients for damping and stiffness.



Tree Object

ACT provides an API for accessing information about objects contained within the **Tree** object in Mechanical.

The **Tree** object can be accessed via **ExtAPI.DataModel.Tree**. The following table provides a sampling of the APIs available for querying the tree for its contained objects. Usage examples appear after the table.

Member	Description
ActiveObjects	Lists all selected objects. Read-only.
AllObjects	Lists all of the objects available in the tree. Read-only.
GetObjectsByName	Lists all objects that match the specified name.
GetObjectsByType	Lists all objects that match the specified type. To do so, use the method GetObjectsByType and pass the type (such as Force) as

Member	Description
	an argument. The whole namespace of the type must be specified.
GetPathToFirstActiveObject	Shows the full statement that must be typed to get the selected object.
Suspend	<p>Prevents updates to the tree.</p> <p>When many objects are added, you can avoid performance issues by specifying that the tree is not refreshed with each addition.</p> <p>This command should be used with the keyword "with" (in Python) or "using" (in C#).</p>

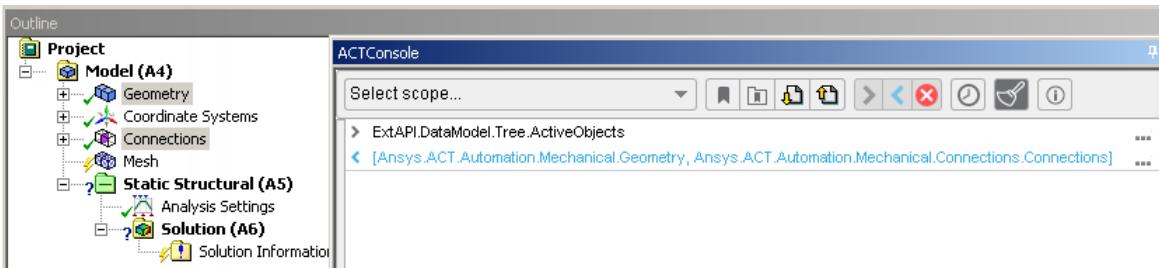
Note

A comprehensive list can be found in the [ANSYS ACT API Reference Guide](#) in the installed help or in the online version available from the [ACT Resources](#) page on the [ANSYS Customer Portal](#). To display the page, select **Downloads > ACT Resources**. To display the online version, expand the **Help & Support** section and click the **ACT Online Reference Guide** link under **Documentation**.

In the **ACT Console**, you can enter commands to query the tree for its contained objects.

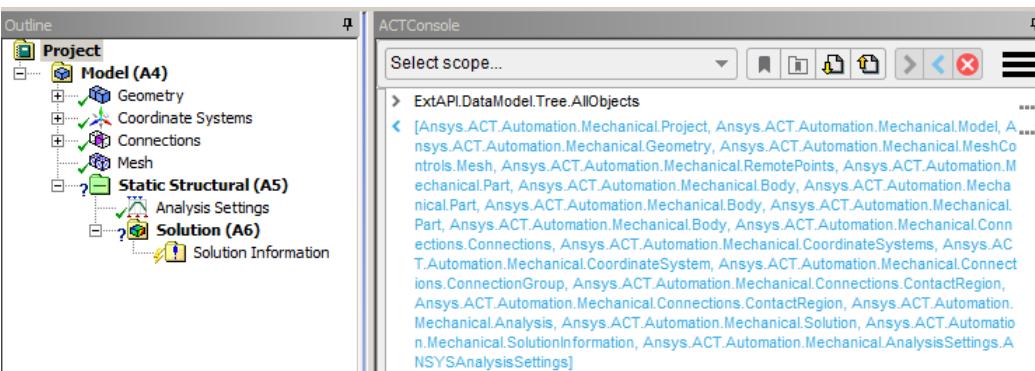
To list all objects selected in the tree, type:

```
ExtAPI.DataModel.Tree.ActiveObjects
```



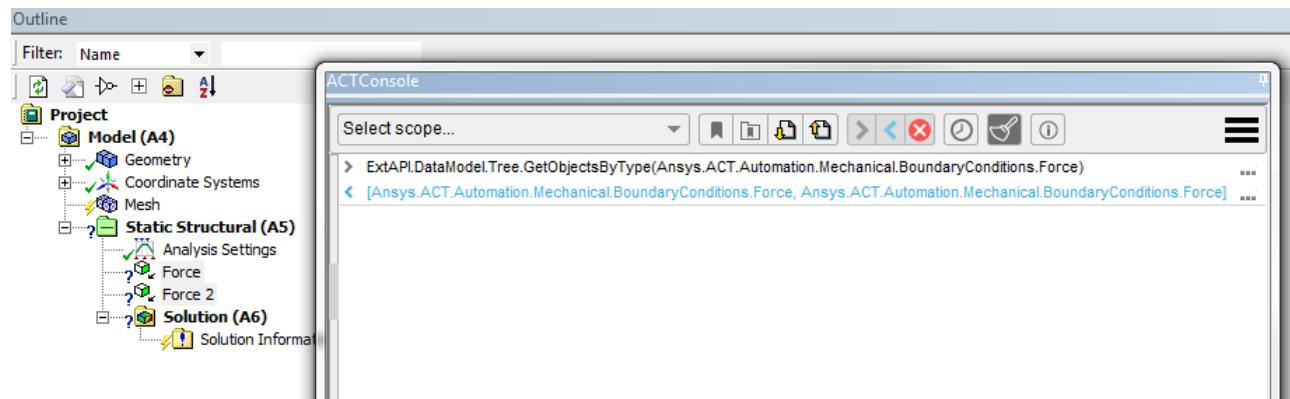
To list all objects available in the tree, type:

```
ExtAPI.DataModel.Tree.AllObjects
```



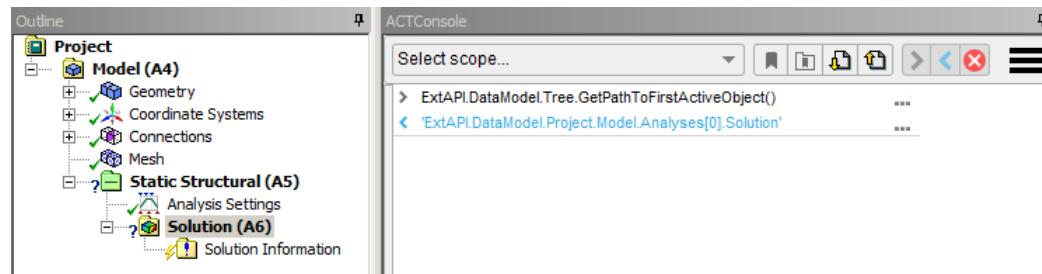
To access objects by type, type:

```
ExtAPI.DataModel.Tree.GetObjectsByType(Ansys.ACT.Mechanical.BoundaryConditions.Force)
```



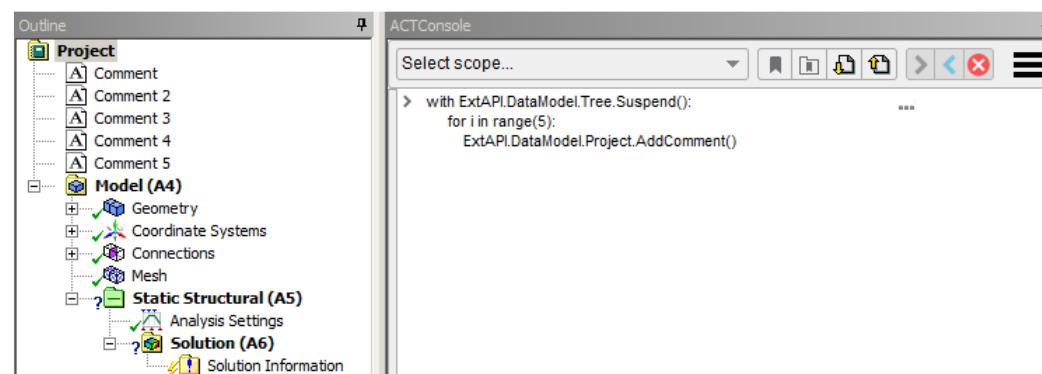
To show the command to get the selected object, type:

```
ExtAPI.DataModel.Tree.GetPathToFirstActiveObject()
```



To suspend an update and add comments, type:

```
with ExtAPI.DataModel.Tree.Suspend():
    for i in range(5):
        ExtAPI.DataModel.Project.AddComment()
```



Model Object

This section addresses the programming calls that query for properties of **Geometry**, **Mesh**, and **Connections** properties within the **Model** object of the Mechanical tree.

The following examples all share the same four general steps:

1. Obtain the tree item under which the object is to be created.

2. Create the object.
3. Create the selection information on which the object is to be based.
4. Specify property values for the object.

Geometry: Point Mass

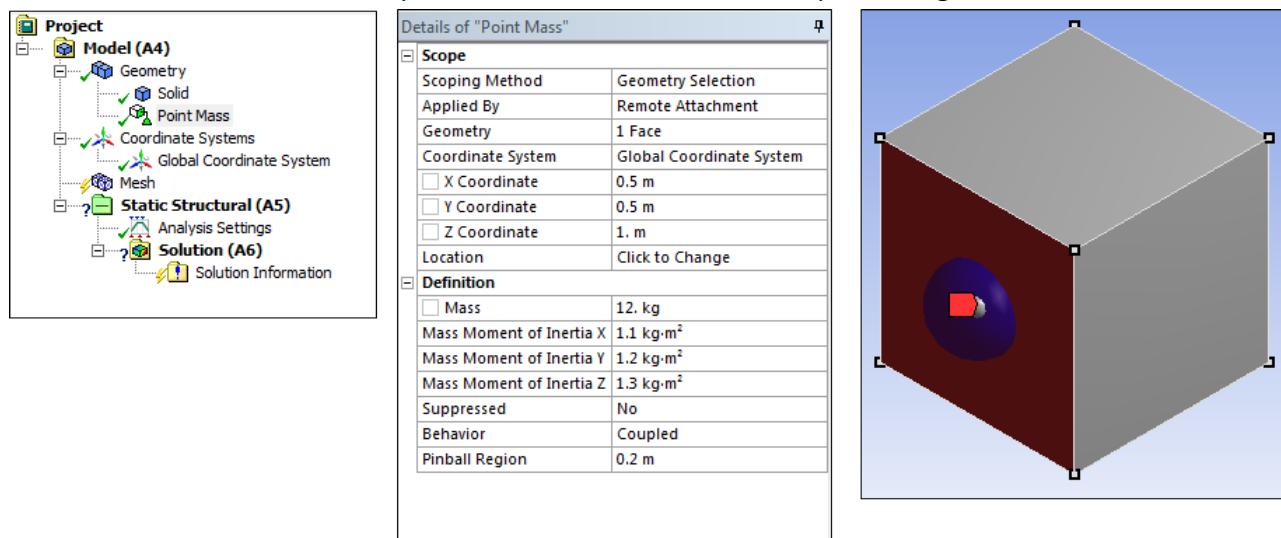
This example shows how to use the ACT API to customize the Geometry object in the Mechanical tree. Specifically, it illustrates how to add a point mass on a face of the geometry and activate the pinball region.

```
geometry = ExtAPI.DataModel.Project.Model.Children[0]

point_mass = geometry.AddPointMass()

my_selection = ExtAPI.SelectionManager.CreateSelectionInfo(SelectionTypeEnum.GeometryEntities)
my_selection.Ids = [22]
point_mass.Location = my_selection
point_mass.Mass = Quantity("12 [kg]")
point_mass.MassMomentOfInertiaX = Quantity("1.1 [kg m m]")
point_mass.MassMomentOfInertiaY = Quantity("1.2 [kg m m]")
point_mass.MassMomentOfInertiaZ = Quantity("1.3 [kg m m]")
point_mass.Behavior = LoadBehavior.Coupled
point_mass.PinballRegion = Quantity("0.2 [m]")
```

It results in the creation of the point mass and activation of the pinball region in Mechanical.



Geometry: Export Geometry Object to an STL File

This example shows how to use the ACT API to export the Geometry object in the Mechanical tree to an STL file. An STL (STereoLithography) file is the most commonly used file format in 3D printing. The following commands get the project model object and geometry object and then export the geometry object to an STL file.

```
mymodel = ExtAPI.DataModel.Project.Model
geo = mymodel.Geometry
geo.ExportToSTL("C:\Temp\geoasst1.stl")
```

The result is the creation of a geometry file (geoasst1.stl) to the fully qualified directory path (C:\Temp).

Mesh: Mesh Control

This example shows how to use the ACT API to customize the Mesh object in the Mechanical tree. Specifically, it illustrates how to create a meshing control that applies the Patch Independent algorithm to the mesh.

```
mesh = ExtAPI.DataModel.Project.Model.Mesh
mesh_method = mesh.AddAutomaticMethod()

my_selection = ExtAPI.SelectionManager.CreateSelectionInfo(SelectionTypeEnum.GeometryEntities)
my_selection.Ids = [16]
mesh_method.Location = my_selection
mesh_method.Method = MethodType.AllTriAllTet
mesh_method.Algorithm = MeshMethodAlgorithm.PatchIndependent
mesh_method.MaximumElementSize = Quantity("0.05 [m]")
mesh_method.FeatureAngle = Quantity("12.000000000000002 [degree]")
mesh_method.MeshBasedDefeaturing = True
mesh_method.DefeaturingTolerance = Quantity("0.0001 [m]")
mesh_method.MinimumSizeLimit = Quantity("0.001 [m]")
mesh_method.NumberOfCellsAcrossGap = 1
mesh_method.CurvatureNormalAngle = Quantity("36 [degree]")
mesh_method.SmoothTransition = True
mesh_method.TetraGrowthRate = 1
```

It results in the creation of a Patch Independent meshing control in Mechanical.

Connections: Frictionless Contact and Beam

This example shows how to use the ACT API to customize the Connections object in the Mechanical tree. Specifically, it illustrates how to add a frictionless contact and a beam connection.

```
connection = ExtAPI.DataModel.Project.Model.Connections
contact_region = connection.Children[0].Children[0]
contact_region.ContactType = ContactType.Frictionless

beam = connection.AddBeam()
beam.Radius = Quantity("0.005 [m]")
reference_scoping = ExtAPI.SelectionManager.CreateSelectionInfo(SelectionTypeEnum.GeometryEntities)
reference_scoping.Ids = [110]
beam.ReferenceLocation = reference_scoping
beam.ReferenceBehavior = LoadBehavior.Deformable
beam.ReferencePinballRegion = Quantity("0.001 [m]")
mobile_scoping = ExtAPI.SelectionManager.CreateSelectionInfo(SelectionTypeEnum.GeometryEntities)
mobile_scoping.Ids = [38]
beam.MobileLocation = mobile_scoping
beam.MobileZCoordinate = Quantity("6.5E-03 [m]")
beam.MobilePinballRegion = Quantity("0.001 [m]")
```

It results in the creation of a frictionless contact and beam connection in ANSYS Mechanical.

Analysis: Load Magnitude

This example shows how to use the ACT API to customize an Analysis object in the Mechanical tree, adding the magnitude of a load. Specifically, it illustrates how to customize a Static Structural analysis, specifying the external and internal pressure exerted on a pipe and then applying a force to a section of the pipe.

```
static_structural = ExtAPI.DataModel.Project.Model.Analyses[0]
analysis_settings = static_structural.AnalysisSettings.NumberOfSteps = 4

bolt = static_structural.AddBoltPretension()
bolt_scoping = ExtAPI.SelectionManager.CreateSelectionInfo(SelectionTypeEnum.GeometryEntities)
bolt_scoping.Ids = [200]
bolt.Location = bolt_scoping
```

```

bolt.SetDefineBy(1, BoltLoadDefineBy.Load) # Change definition for step #1.
bolt.Preload.Output.SetDiscreteValue(0, Quantity("15 [N]")) # Change preload value for step #1.

support = static_structural.AddFixedSupport()
support_scoping = ExtAPI.SelectionManager.CreateSelectionInfo(SelectionTypeEnum.GeometryEntities)
support_scoping.Ids = [104]
support.Location = support_scoping

pressure = static_structural.AddPressure()
pressure_scoping = ExtAPI.SelectionManager.CreateSelectionInfo(SelectionTypeEnum.GeometryEntities)
pressure_scoping.Ids = [220]
pressure.Location = pressure_scoping
pressure.Magnitude.Output.Formula = '10*time'

pressure = static_structural.AddPressure()
pressure_scoping = ExtAPI.SelectionManager.CreateSelectionInfo(SelectionTypeEnum.GeometryEntities)
pressure_scoping.Ids = [221]
pressure.Location = pressure_scoping
pressure.Magnitude.Output.DiscreteValues=[Quantity('6 [Pa]')]

force = static_structural.AddForce()
force_scoping = ExtAPI.SelectionManager.CreateSelectionInfo(SelectionTypeEnum.GeometryEntities)
force_scoping.Ids = [219]
force.Location = force_scoping
force.Magnitude.Output.DiscreteValues=[Quantity('11.3 [N]'), Quantity('12.85 [N]')]

```

In this example, you apply the internal and external pressures to the pipe:

```

pressure.Magnitude.Output.Formula = '10*time'
pressure.Magnitude.Output.DiscreteValues=[Quantity('6 [Pa]')]

```

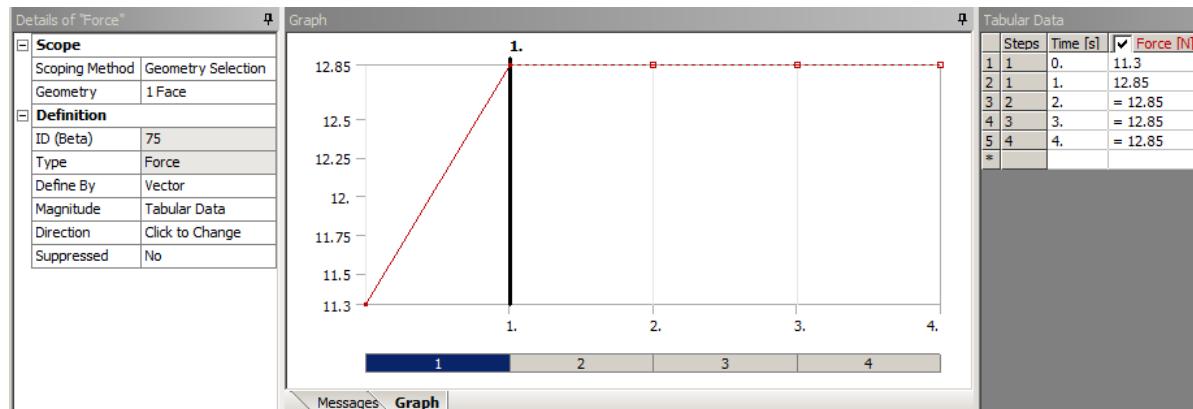
Then you use tabular data to apply a vector force to the pipe:

Note

If you use a constant ramped from **t=0s** to define the force, the first value cannot be "0".

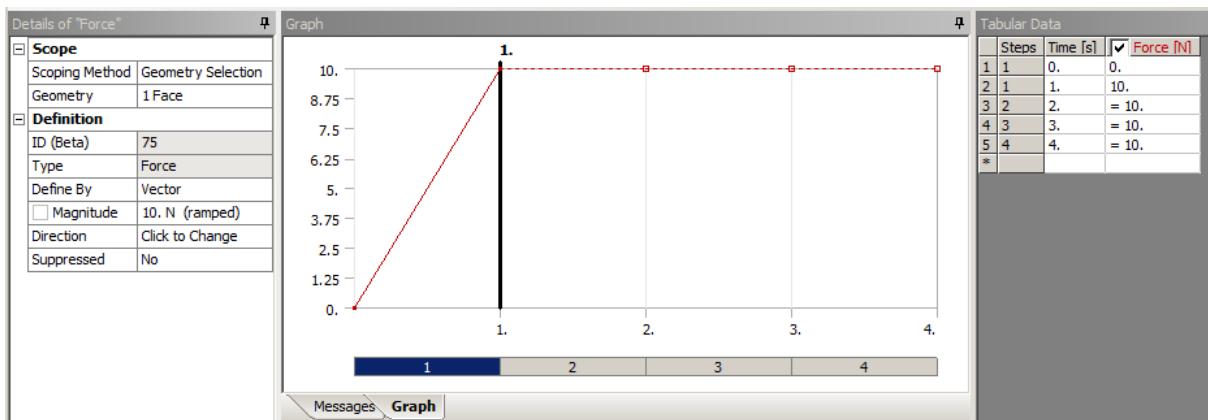
```
force.Magnitude.Output.DiscreteValues=[Quantity('11.3 [N]'), Quantity('12.85 [N]')]
```

Script execution results in the creation of the property **Magnitude** for the applied force, with time as an input variable and a single output variable.



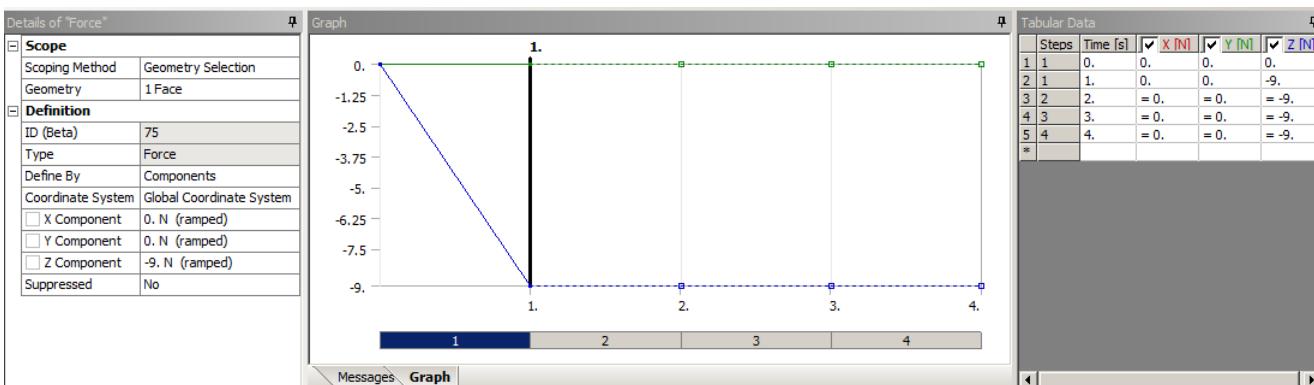
Although initially you used tabular data to define the property **Magnitude**, you can also use a constant value or a time- or space-dependent formula. An example follows of how to use a constant value to define the property.

```
force.Magnitude.Output.DiscreteValues=[Quantity('10 [N]')]
```



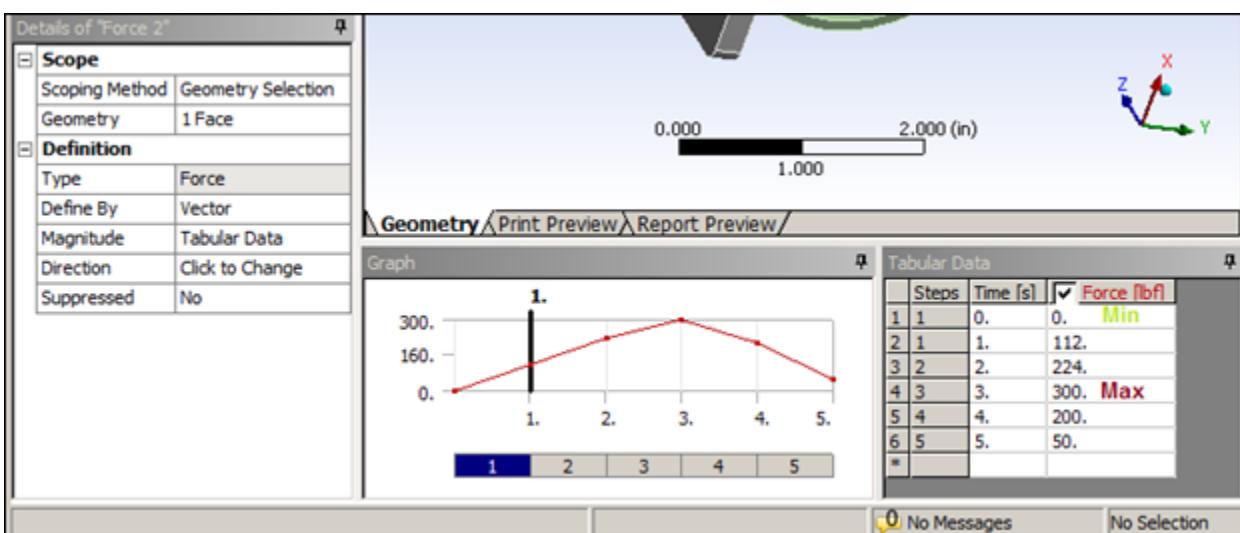
You can also opt to define the property **Magnitude** with global coordinates instead of a vector by using the following commands:

```
force.DefineBy = LoadDefineBy.Components
force.ZComponent.Output.DiscreteValues = [Quantity('0 [N]'),Quantity('-9 [N]')]
```



Analysis: Extract Min-Max Tabular Data for a Boundary Condition

You can use the ACT API to extract minimum and maximum tabular data for boundary conditions (force, pressure, moment, and so on) from an Analysis object in the Mechanical tree. This example shows how to extract minimum and maximum force quantities from tabular data for an Analysis object. For your reference, the following image displays tabular data for a force and highlights its min and max quantities.



First, to get the project model object, analysis object, force object, and tabular data object, enter:

```
mymodel = ExtAPI.DataModel.Project.Model  
anal = mymodel.Analyses[0]  
f2 = anal.Children[2]  
f2td = f2.Magnitude.Output
```

Next, to get the tuple containing the minimum and maximum force quantities and then display these quantities, enter:

```
mnmx2 = f2td.MinMaxDiscreteValues  
mnmx2
```

Given the above tabular data, the following results display:

(0 [lbf], 300 [lbf])

The variable `mnmx2` is a tuple (pair) of quantities. Each element in the tuple can be gotten by using the tuple's `Item1` and `Item2` properties.

If you want to get and display only the minimum force quantity, you enter:

```
mnv = mnmx2.Item1  
mnv
```

Given the above tabular data, the following results display:

(0 [lbf])

If you want to get and display only the maximum force quantity, you enter:

```
mxv = mnmx2.Item2  
mxv
```

Given the above tabular data, the following results display:

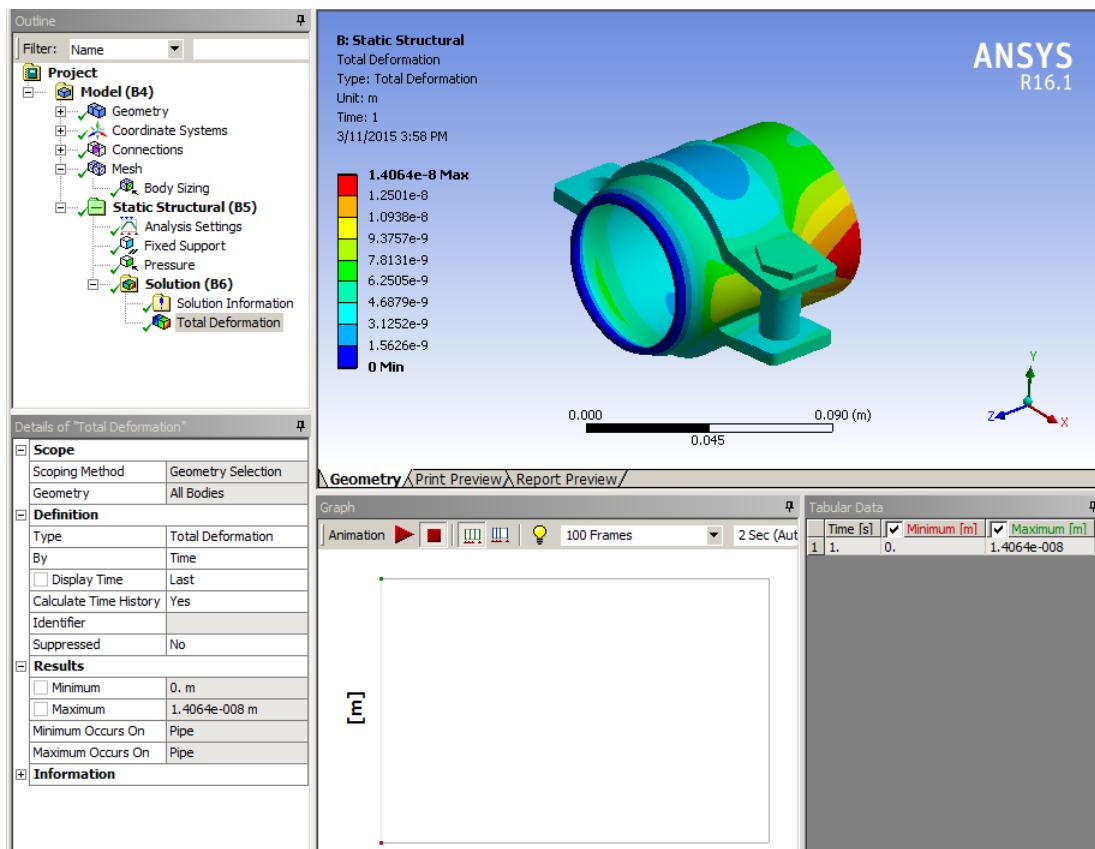
(300 [lbf])

Result: Total Deformation Maximum

This example shows how to use the ACT API to customize the Result object in the Mechanical tree. Specifically, it illustrates how to add a **Total Deformation** result to a Static Structural analysis and then solve for the minimum and maximum total deformation.

```
solution = ExtAPI.DataModel.Project.Model.Analyses[0].Solution  
total_deformation = solution.AddTotalDeformation()  
analysis = ExtAPI.DataModel.Project.Model.Analyses[0]  
analysis.Solve(True)  
minimum_deformation = total_deformation.Minimum  
maximum_deformation = total_deformation.Maximum
```

It results in a solved analysis indicating the values for the **Minimum** and **Maximum** properties for the **Total Deformation** result:



TraverseExtension

The examples in this section address programming calls that query geometry, mesh, simulation, and results properties. Each of the code samples are taken from the **TraverseExtension** extension.

Note

This extension is included in the package **ACT Developer's Guide Examples**, which you can download from the [ACT Resources](#) page on the [ANSYS Customer Portal](#). To display the page, select **Downloads > ACT Resources**. To download the package, expand the **Help & Support** section and click **ACT Developer's Guide Examples** under **Documentation**.

Traversing the Geometry

The API for geometry data is organized to match the underlying hierarchical data model. The API offers a variety of property queries through which the connectivity of the geometry entities can be determined. For instance, it is possible to query for the faces upon which an edge is defined by using the property **Faces** exposed by the edge interface. See the [ANSYS ACT API Reference Guide](#) for a comprehensive list of these interfaces and their properties.

The basic hierarchy of the geometry is:

- Geometry
- Assembly
- Part
- Body
- Shell
- Face

- Edge
- Vertex

The IronPython script function **traversegeometry()** demonstrates how an extension could traverse the geometry data. In this example, an object of type **IGeoData** is obtained from the object **Analysis** using the property **GeoData**. This object **GeoData** is then used to query for a list of **IGeoAssembly** objects by calling the property **Assembly**. For each of the **IGeoAssembly** objects in the returned list, the property **Parts** is called and a list of **IGeoPart** objects is returned. This pattern is repeated through the hierarchy of the geometry down to the vertices of each edge.

The content of the function **traversegeometry()** follows:

```
def traversegeometry(analysis):
    now = datetime.datetime.now()
    outFile = SetUserOutput(analysis, "SolutionDetails.log")
    f = open(outFile,'a')
    f.write("*.**.*.*.*\n")
    f.write(str(now)+"\n")
    # --- IGeometry Interface
    # +++ Properties and Methods
    # ++++ Assemblies
    # ++++ CellFromRefId
    # ++++ SelectedRefIds
    geometry = analysis.GeoData
    assemblies = geometry.Assemblies
    assemblies_count = assemblies.Count
    # --- IGeoAssembly Interface
    # +++ Properties and Methods
    # ++++ Name
    # ++++ Parts
    for assembly in assemblies:
        assembly_name = assembly.Name
        parts = assembly.Parts
        parts_count = parts.Count
        # --- IGeoPart Interface
        # +++ Properties and Methods
        # ++++ Name
        # ++++ Bodies
        for part in parts:
            part_name = part.Name
            bodies = part.Bodies
            bodies_count = bodies.Count
            # --- IGeoBody Interface
            # +++ Properties and Methods
            # ++++ Name
            # ++++ Vertices
            # ++++ Edges
            # ++++ Faces
            # ++++ Shells
            # ++++ Material
            for body in bodies:
                faces = body.Faces
                faces_count = faces.Count
                # --- IGeoFace Interface
                # +++ Properties and Methods
                # ++++ Body
                # ++++ Shell
                # ++++ Vertices
                # ++++ Edges
                # ++++ Loops
                # ++++ Area
                # ++++ SurfaceType
                # ++++ PointAtParam
                # ++++ PointsAtParams
                for face in faces:
                    edges = face.Edges
                    edges_count = edges.Count
                    # --- IGeoEdge Interface
                    # +++ Properties and Methods
```

```

# +++ Faces
# +++ Vertices
# +++ StartVertex
# +++ EndVertex
# +++ Length
# +++ CurveType
# +++ Extents
# +++ IsParamReversed
# +++ ParamAtPoint
# +++ PointAtParam
# +++ PointsAtParams
for edge in edges:
    vertices = edge.Vertices
    vertices_count = vertices.Count
    # --- IGeoVertex Interface
    # +++ Properties and Methods
    # +++ Edges
    # +++ Faces
    # +++ Bodies
    # +++ X
    # +++ Y
    # +++ Z
    for vertex in vertices:
        xcoord = vertex.X
        ycoord = vertex.Y
        zcoord = vertex.Z
        try:
            f.write("      Vertex: "+vertex.ToString()+" , X = "+xcoord.ToString()+" , Y = "
"+ycoord.ToString()+" , Z = "+zcoord.ToString()+"\n")
        except:
            continue
f.close()
return

```

Traversing the Mesh

The API **Mesh** offers a variety of property queries through which the connectivity of the mesh entities can be determined. For comprehensive information on the interfaces and properties for this API, see the [ANSYS ACT API Reference Guide](#).

The following IronPython script function **traversemesh()** demonstrates how an extension could traverse the mesh data. In this example, an object of type **IMeshData** is obtained from the object **IAnalysis** using the property **MeshData**. This mesh object is then used to query for a list of element IDs with the property **Elements**. For each of the element IDs in the returned list, the method **Element** is called and the corresponding object **IElement** is returned. This pattern is repeated for all the nodes for each element. Finally, the coordinates of the nodes are queried. Comments are included in the content of the function **traversemesh()** to clarify script functionality.

```

def traversemesh(analysis):
    now = datetime.datetime.now()
    outFile = SetUserOutput(analysis, "SolutionDetails.log")
    f = open(outFile,'a')
    f.write(".*.*.*.*.*.*\n")
    f.write(str(now)+"\n")
    # --- IMesh Interface
    # +++ Properties and Methods
    # +++ MeshRegion
    # +++ Node
    # +++ Element
    # +++ Nodes
    # +++ Elements
    # +++ NumNodes
    # +++ NumElements
    mesh = analysis.MeshData
    elementids = mesh.ElementIds
    # --- IElement Interface
    # +++ Properties and Methods

```

```

# +++ Id
# +++ Type
# +++ Nodes
for elementid in elementids:
    element = mesh.ElementById(elementid)
    nodeids = element.NodeIds
    # --- INode Interface
    # +++ Properties and Methods
    # +++ Id
    # +++ X
    # +++ Y
    # +++ Z
    # +++ Elements
    for nodeid in nodeids:
        node = mesh.NodeById(nodeid)
        nodeX = node.X
        nodeY = node.Y
        nodeZ = node.Z
        try:
            f.write("      Element: "+elementid.ToString()+"  Node: "+nodeid.ToString()+" , X = "
"+nodeX.ToString()+" , Y = "+nodeY.ToString()+" , Z = "+nodeZ.ToString()+"\n")
        except:
            continue
    f.close()
return

```

The script function **elementcounter()** can also be used to access the mesh data. In this example, only the elements of user-selected geometry entities are considered. First, the script obtains the objects **IGeoData** and **IMeshData**. The property **CurrentSelection.Ids** then queries the IDs of the selected geometry entities using the object **ISelectionMgr**. If no IDs are returned, a message box displays the message "Nothing Selected." Otherwise, the methods **GeoEntityById** and **MeshRegionById** obtain the objects **IGeoEntity** and **IMeshRegion** corresponding to each selected entity. These two objects are used inside the try-except block to query for the type of entity selected and the number of elements in each entity's mesh. The property **Type** of the interface **IGeoEntity** and the property **NumElements** of the interface **IMeshRegion** are used here and the results are displayed in a message box.

```

def elementcounter(analysis):
    geometry = analysis.GeoData
    mesh = analysis.MeshData
    selectedids = ExtAPI.SelectionManager.CurrentSelection.Ids
    if selectedids.Count == 0:
        MessageBox.Show("Nothing Selected!")
    else:
        for selectedid in selectedids:
            entity = geometry.GeoEntityById(selectedid)
            meshregion = mesh.MeshRegionById(selectedid)
            try:
                numelem = meshregion.ElementCount
                MessageBox.Show("Entity of type: "+entity.Type.ToString()+
                               " contains "+numelem.ToString()+
                               " elements.")
            except:
                MessageBox.Show("The mesh is empty!")
    return

```

Traversing Results

The script function **minmaxresults()** computes the minimum and maximum component values of the nodal displacement and the SXX stress component. It begins by instantiating a result reader using the method **analysis.ResultsData**. Results are retrieved relative to the finite element model and queried using either the **elementID** (elemental result) or the **nodeID** (nodal result). The displacement result "U" is a nodal result, whereas the stress result "S" is a result on nodes of the elements. The dis-

placement result stores a set of component values for each node, where the component names are X, Y, and Z.

The script **minmaxresults** first iterates over the **nodeIDs** to compute the minimum and maximum values. Then, the script iterates over the **elementIDs** and the nodes of each element to compute the minimum and maximum values.

Note

The second loop over the nodes is filtered to the primary nodes of the elements because stress results are available only on these primary nodes.

Finally, the results are written to the output file.

The contents of the function **minmaxresults()** follow.

```
def minmaxresults(analysis):
    now = datetime.datetime.now()
    outFile = SetUserOutput(analysis, "SolutionDetails.log")
    f = open(outFile,'a')
    f.write("*.*.*.*.*.*\n")
    f.write(str(now)+"\n")
    #
    # Get the element ids
    #
    meshObj = analysis.MeshData
    elementids = meshObj.ElementIds
    nodeids = meshObj.NodeIds
    #
    # Get the results reader
    #
    reader = analysis.GetResultsData()
    reader.CurrentResultSet = int(1)
    #
    # Get the displacement result object
    displacement = reader.GetResult("U")

    num = 0
    for nodeid in nodeids:
        #
        # Get the component displacements (X Y Z) for this node
        #
        dispvals = displacement.GetNodeValues(nodeid)
        #
        # Determine if the component displacement (X Y Z) is min or max
        #
        if num == 0:
            maxdispX = dispvals[0]
            mindispX = dispvals[0]
            maxdispY = dispvals[1]
            mindispY = dispvals[1]
            maxdispZ = dispvals[2]
            mindispZ = dispvals[2]

        num += 1

        if dispvals[0] > maxdispX:
            maxdispX = dispvals[0]
        if dispvals[1] > maxdispY:
            maxdispY = dispvals[1]
        if dispvals[2] > maxdispZ:
            maxdispZ = dispvals[2]
        if dispvals[0] < mindispX:
            mindispX = dispvals[0]
        if dispvals[1] < mindispY:
            mindispY = dispvals[1]
```

```

if dispvals[2] < mindispz:
    mindispz = dispvals[2]

# Get the stress result object
stress = reader.GetResult("S")

num = 0
for elementid in elementids:
    element = meshObj.ElementById(elementid)
    #
    # Get the SXX stress component
    #
    stressval = stress.GetElementValues(elementid)
    #
    # Get the primary node ids for this element
    #
    nodeids = element.CornerNodeIds
    for i in range(nodeids.Count):
        #
        # Get the SXX stress component at node "nodeid"
        #
        SXX = stressval[i]
        #
        # Determine if the SXX stress component is min or max
        #
        if num == 0:
            maxsxx = SXX
            minsxx = SXX

        if SXX > maxsxx:
            maxsxx = SXX
        if SXX < minsxx:
            minsxx = SXX

        num += 1
#
# Write the results to the output
#
f.write("Max U,X:Y:Z = "+maxdispX.ToString()+" : "+maxdispy.ToString()+" : "+maxdispz.ToString()+"\n")
f.write("Min U,X:Y:Z = "+mindispX.ToString()+" : "+mindispy.ToString()+" : "+mindispz.ToString()+"\n")
f.write("Max SXX = "+maxsxx.ToString()+"\n")
f.write("Min SXX = "+minsxx.ToString()+"\n")
f.close()

```

Graphical Views

The API **ModelViewManager** allows you to set, create, delete, apply, rename, rotate, save, import, and export graphical views, providing an alternative to the **View Manager** window in Mechanical.

Note

The API **ModelViewManager** does not export graphical views exactly as they appear in Mechanical. For example, the exported views exhibit inconsistencies in the zoom and model orientation. If you want to compare views to see changes in results, export them to PNG files using the command **ExtAPI.Graphics.ExportScreenToImage** instead. This command exports the current view as it appears in Mechanical to a PNG file:

```
ExtAPI.Graphics.ExportScreenToImage(filePath)
```

Where **filePath** is the fully qualified path for the PNG file to create.

Currently, only the PNG file format is supported. The file path must be surrounded by escape characters. An example follows:

```
ExtAPI.Graphics.ExportScreenToImage("C:\\\\Users\\\\Desktop\\\\image1.png")
```

The exported view retains the same orientation, zoom factor, and other graphics rendering properties as the view in Mechanical.

In situations where exported graphical views do not have to exactly match what is shown in Mechanical, you can use the API **ModelViewManager**:

[Getting the ModelViewManager Data Object](#)

[Setting the View](#)

[Creating a View](#)

[Deleting a View](#)

[Applying a View](#)

[Renaming a View](#)

[Rotating a View](#)

[Saving a View](#)

[Saving an Object](#)

[Importing a Saved View List](#)

[Exporting a Saved View List](#)

[Exporting an Object](#)

Getting the ModelViewManager Data Object

To get the data object **ModelViewManager**, you enter the following in the command line of the **ACT Console**:

```
model = ExtAPI.DataModel.Project.Model
view_manager = ExtAPI.Graphics.ModelViewManager
```

If you wanted to get the number of graphical views that are currently defined, you would enter:

```
views_count = view_manager.NumberOfViews
```

Setting the View

Before creating a graphical view, you can set the type of view you want to capture. The following table provides commands for setting views.

Table 1: Commands for Setting Graphical Views

Action	Command
Set to ISO view	<code>view_manager.SetISOView()</code>
Set to Fit view	<code>view_manager.SetFitView()</code>
Set to Front view	<code>view_manager.SetFrontView()</code>
Set to Back view	<code>view_manager.SetBackView()</code>
Set to Right view	<code>view_manager.SetRightView()</code>
Set to Left view	<code>view_manager.SetLeftView()</code>
Set to Top view	<code>view_manager.SetTopView()</code>
Set to Bottom view	<code>view_manager.SetBottomView()</code>

Creating a View

You can create a graphical view from current graphics using either a default name or a specified name.

To create a view using a default name, you enter:

```
view_manager.CreateView
```

The name assigned to the new view is View followed by the next sequential number.

To create a view using a specified name, you enter the following, where `viewName` is the name to assign to the new view:

```
view_manager.CreateView(string viewName)
```

For example, assume that you want to set the top view and then create a view named Top View. You enter:

```
view_manager.SetTopView()
view_manager.CreateView("Top View")
```

Deleting a View

You can delete a view by specifying its name or index.

To delete a view by specifying its name, you enter the following, where `viewLabel` is the name of the view to delete:

```
DeleteView(string viewLabel)
```

For example, to delete a view named Left View, you enter:

```
view_manager.DeleteView("Left View")
```

To delete a view by specifying its index, you enter the following, where `viewIndex` is the index of the listed view to delete:

```
DeleteView(int viewIndex)
```

For example, to delete a view with index 0, you enter:

```
view_manager.DeleteView(0)
```

Applying a View

You can apply a graphical view by specifying its name or index.

To apply a view by specifying its name, you enter the following, where `viewLabel` is the name of the view to apply:

```
view_manager.ApplyModelView(string viewLabel)
```

For example, to apply a view named Fit View, you enter:

```
view_manager.ApplyModelView("Fit View")
```

To apply a view by specifying its index, you enter the following, where `viewIndex` is the index of the listed view to apply:

```
view_manager.ApplyModelView(int viewIndex)
```

For example, to apply a view with index 1, you enter:

```
view_manager.ApplyModelView(1)
```

Renaming a View

To rename a saved graphical view, you enter the following, where `viewLabel` is the current name of the view to rename and `newLabel` is the new name to assign to the view:

```
view_manager.RenameView(string viewLabel, string newLabel)
```

For example, assume that a view is currently named `View 7` and you want to rename it to `ISO View`. You enter:

```
view_manager.RenameView("View 7", "ISO View")
```

Rotating a View

To rotate the active view, you enter the following, where `angle` is the amount of rotation:

```
view_manager.RotateView(double angle)
```

For example, to rotate the active object 45 degrees clockwise, you enter:

```
view_manager.RotateView(45)
```

Note

For the `angle` parameter, you enter a positive value to rotate the view clockwise or a negative value to rotate the view counter-clockwise.

Saving a View

You can save a graphic view of the entire model as an image file by specifying its name or index. For all methods, `mode` is a string representing the graphics format of the file to which to save the image. The values that can be entered for `mode` are "PNG", "JPG", "TIF", "BMP", and "EPS".

Note

For commands that use the parameter `folder`, an empty string ("") for this parameter defaults to the project user files. This is a null operation if the specified folder does not exist.

To save a view specified by name as an image to the project user files, you enter the following, where `viewLabel` is the name of the view to save and `mode` is the graphics format for the image file:

```
view_manager.CaptureModelView(string viewLabel, string mode)
```

For example, to save a view named `Bottom View` to a TIF file to the project user files, you enter:

```
view_manager.CaptureModelView("Bottom View", "TIF")
```

To save a view specified by name as an image to a specified folder, you enter the following, where `viewLabel` is the name of the view to save, `mode` is the graphics format for the image file, and `folder` is the name of the folder in which to save the file:

```
view_manager.CaptureModelView(string viewLabel, string mode, string folder)
```

For example, to save a view named `Right View` to a JPG file to `D:\My_Projects\Views`, you enter:

```
view_manager.CaptureModelView("Right View", "JPG", "D:\My_Projects\Views")
```

To save a view specified by index as an image to a specified folder, you enter the following, where `index` is the index of the view to save, `mode` is the graphics format for the image file, and `folder` is the name of the folder in which to save the file:

```
view_manager.CaptureModelView(index, string mode, string folder)
```

For example, to save a view with index 0 to a PNG file to the project user files, you enter:

```
view_manager.CaptureModelView(0, "PNG")
```

Saving an Object

To save an object as an image to a file having the same name as the object, you enter the following, where `obj` is the object to save, `mode` is the graphics format for the image file, and `folder` is the name of the folder in which to save the file:

```
view_manager.CaptureObjectImage(Ansys.ACT.Automation.Mechanical.DataModelObject obj, string mode, string folder)
```

For example, to save an object named `plate` to a BMP file to the project user files, you enter:

```
view_manager.CaptureObjectImage(Ansys.ACT.Automation.Mechanical.DataModelObject "Plate", "BMP")
```

Note

An empty string ("") for the parameter `folder` defaults to the project user files. This is a null operation if the specified folder does not exist.

Importing a Saved View List

To import a saved view list by specifying a filename, you enter the following, where `viewfilepath` is the name of the view (XML file) to import:

```
view_manager.ImportModelViews(string viewfilepath)
```

For example, to import a saved view list named `allviews.xml` that is located in `D:\My_Projects\Views`, you enter:

```
view_manager.ImportModelViews("D:\My_Projects\Views\allviews.xml")
```

Exporting a Saved View List

To export all saved graphical views to an XML file, you enter the following, where `viewfilepath` is the name of the XML file to create:

```
view_manager.ExportModelViews(string viewfilepath)
```

For example, to export a saved view list named `myviews.xml` to `D:\My_Projects\Views`, you enter:

```
view_manager.ExportModelViews("D:\My_Projects\Views\myviews.xml")
```

Exporting an Object

To export the active object to a 3D AVZ file, you enter the following, where `avzfffilepath` is a fully qualified AVZ filepath:

```
view_manager.Capture3DImage(string avzfilepath)
```

For example, to export the active object to a file named `my_plane.avz` to `D:\My_Projects\Views`, you enter:

```
view_manager.Capture3DImage( "D:\My_Projects\Views\my_plane.avz" )
```

APIs for ANSYS DesignModeler

The following sections describe some of the APIs provided by ACT for customizing the interface and functionality of ANSYS DesignModeler. Once customizations are in place, they can be used to create and manipulate geometries. You can use them to first define and generate primitives and bodies, and then to apply various operations, tools, or automated processes to the resulting geometries.

The examples provided in this chapter address working with a selected graphic, creating different types of primitives, and applying different operations. The code for creating primitive bodies or applying operations must be integrated into the callback `<ongenerate>`.

For more comprehensive information on the DesignModeler API interfaces and their properties, refer to the [ANSYS ACT API Reference Guide](#).

Using the Selection Manager in DesignModeler

The DesignModeler API `SelectionManager` allows you to work with graphical selections. It provides service to get information from and to modify, clear, or retrieve the current selection. Additionally, it enables you to create new selections or add entities to the new selection.

Working with the Current Selection

The following Selection Manager commands allow you to perform different actions on the currently selected graphic.

To access the Selection Manager:

```
selection_manager = ExtAPI.SelectionManager
```

To clear the current selection:

```
selection_manager.ClearSelection()
```

To retrieve the current selection:

```
selection = selection_manager.CurrentSelection
```

This command returns the object `ISelectionInfo`, which describes the selected entities.

To retrieve entities:

```
selection.Entities
```

Creating a New Selection and Adding Entities

The Selection Manager enables you to create new selections and add entities to the current selection. Both of these tasks can be accomplished either by creating a new object **ISelectionInfo** or by directly using a list of entities.

To create a new selection using a new object **ISelectionInfo**:

```
face = ExtAPI.DataModel.GeoData.Bodies[0].Faces[0]
selection = selection_manager.CreateSelectionInfo(SelectionTypeEnum.GeometryEntities)
selection.Entities = [face]
selection_manager.NewSelection(selection)
```

To create a new selection using a list of entities:

```
face = ExtAPI.DataModel.GeoData.Bodies[0].Faces[0]
selection_manager.NewSelection([face])
```

To add entities to the current selection using a new object **ISelectionInfo**:

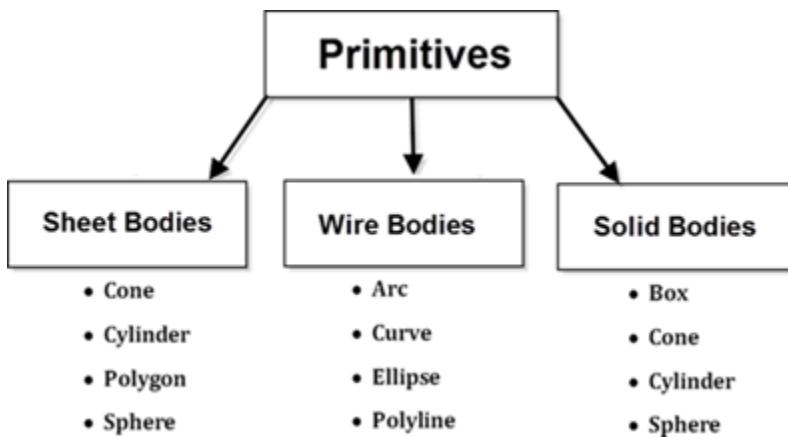
```
face = ExtAPI.DataModel.GeoData.Bodies[0].Faces[0]
selection = selection_manager.CreateSelectionInfo(SelectionTypeEnum.GeometryEntities)
selection.Entities = [face]
selection_manager.AddSelection(selection)
```

To add entities to the current selection using a list of entities:

```
face = ExtAPI.DataModel.GeoData.Bodies[0].Faces[0]
selection_manager.AddSelection([face])
```

Creating Primitives

The DesignModeler API **Primitives** provides you with the ability to create different types of primitive bodies: sheet bodies, wire bodies, and solid bodies. For each primitive body type, multiple shape options are available.



The primitives variable **ExtAPI.DataModel.GeometryBuilder.Primitives** is the gateway for designing a new geometric body. The code must be integrated into the callback **<ongenerate>**.

The script function **ongenerate()** can be used to create a primitive body. The API provides a variety of queries that allow you to specify properties such as body type, dimensions, shape, point coordinates, and material.

Creating a Sheet Body

When creating a sheet body, you have the option of defining it in the shape of a cone, cylinder, polygon, or sphere. An example follows of using the function `ongenerate()` to define a sheet body cylinder.

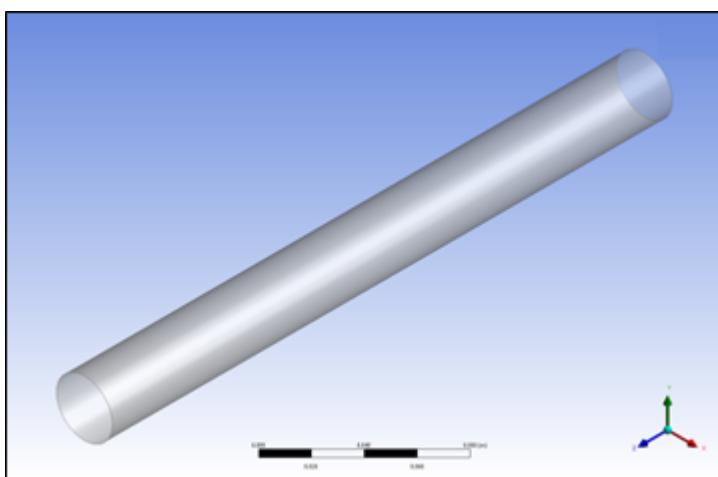
```
def Ongenerate(feature,function):
    width = 0.015
    height = 0.3
    sheetBodies = []
    primitive = ExtAPI.DataModel.GeometryBuilder.Primitives
    cylinder = primitive.Sheet.CreateCylinder([0.,0.,0.],[0.,0.,height],width)
    cylinder_generated = cylinder.Generate()
    sheetBodies.Add(cylinder_generated)

    feature.Bodies = sheetBodies
    feature.MaterialType = MaterialTypeEnum.Freeze

    return True
```

In this example:

- The variables `width` and `height` are used to define the width and the height of the cylinder.
- The variable `sheetBodies` specifies the type of primitive to be created.
- The primitive variable uses the global variable `ExtAPI` to access the data model and define the geometry builder, which serves as entry point for ACT into DesignModeler.
- The method `CreateCylinder ()` is used to generate the new body, specifying that it is defined by the following arguments:
 - Coordinates of the center point of the cylinder's base
 - Coordinates of the center point of the upper face (which defines the direction of the cylinder)
 - Value of the radius
 (This is a float. The integer value `<< 3 >>`, for instance, is refused, while a value of `<< 3. >>` is accepted.)
- With the object `cylinder_generated`, you use the method `Generate ()` to generate the cylinder.



- With the lines `sheetBodies.Add(cylinder_generated)` and `feature.Bodies = sheetBodies`, you add the sheet body cylinder to the list `feature.bodies` so that it is added to DesignModeler after generation. Bodies not added to this list are not retained.
- With the line `MaterialType`, you specify a material property of `Freeze`.

Creating a Wire Body

When creating a wire body, you have the option of defining it in the shape of an arc, curve, ellipse, or polyline. An example follows of using the function `ongenerate()` to define a wire body polyline.

```
def Ongenerate(feature,function):
    points_list = [0.,0.,0., 1.,0.,0., 1.,1.,0., 1.,1.,1.]
    wireBodies = []

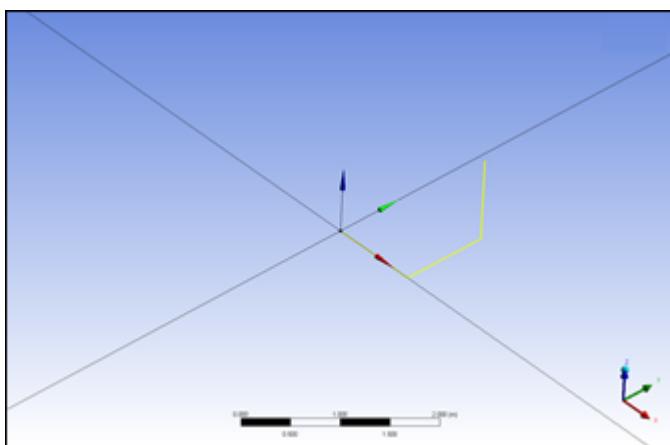
    primitive = ExtAPI.DataModel.GeometryBuilder.Primitives
    polyline = primitive.Wire.CreatePolyline(points_list)
    polyline_generated = polyline.Generate()
    wireBodies.Add(polyline_generated)

    feature.Bodies = wireBodies
    feature.MaterialType = MaterialTypeEnum.Add

    return True
```

In this example:

- The method `points_list ()` is defined for later use in the creation of the polyline body. For arguments, it expects as a list of coordinate points (defined by three float values per point).
- The primitive variable uses the global variable `ExtAPI` to access the data model and define the geometry builder, which serves as entry point for ACT into DesignModeler.
- The method `CreatePolyline ()` is applied to the object `Wire` to generate the new body. As arguments, this method is expecting the coordinate points defined by the method `points_list`.
- With the object `polyline_generated`, you use the method `Generate ()` to generate the polyline.



- The new body is added to the list `feature.bodies` as described in [Creating a Sheet Body \(p. 233\)](#).
- With the line `MaterialType`, you specify a material property of `Add`.

Creating a Solid Body

When creating a solid body, you have the option of defining it in the shape of a box, cone, cylinder, or sphere. The following example uses the function `ongenerate()` to define a solid body box.

```
def Ongenerate(feature,function):
    point1 = [0.,0.,0.]
    point2 = [1.,2.,2.]
    solidBodies = []

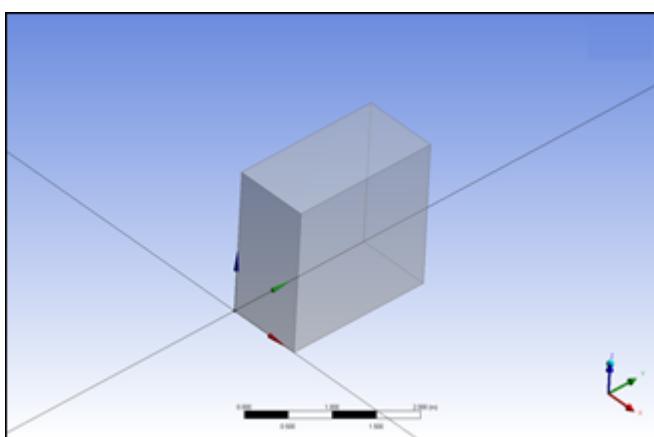
    primitive = ExtAPI.DataModel.GeometryBuilder.Primitives
    box1 = primitive.Solid.CreateBox(point1, point2)
    box1_generated = box1.Generate()
    solidBodies.Add(box1_generated)

    feature.Bodies = solidBodies
    feature.MaterialType = MaterialTypeEnum.Freeze

    return True
```

In this example:

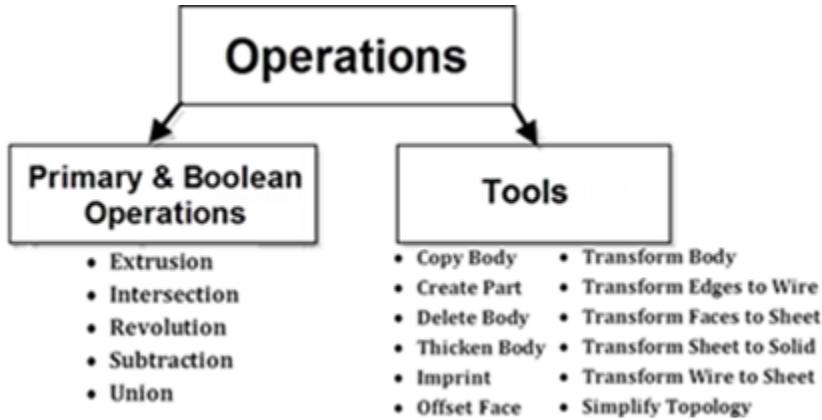
- The methods `point1()` and `point2()` are defined for later use in the creation of the solid body. For arguments, they each expect as a list of coordinate points (defined by three float values per point).
- The primitive variable uses the global variable `ExtAPI` to access the data model and define the geometry builder, which serves as entry point for ACT into DesignModeler.
- The method `CreateBox()` is applied to the object `Solid` to generate the new body. For arguments, this method is expecting the coordinate points defined by the methods `point1` and `point2`.
- With the object `box1_generated`, you use the method `Generate()` to generate the box.



- The new body is added to the list `feature.bodies` as described in [Creating a Sheet Body \(p. 233\)](#).
- With the line `MaterialType`, you specify a material property of `Freeze`.

Applying Operations

The DesignModeler API **Operations** enables you to perform different operations on a geometric body. Available operations can be divided into two different categories: **Primary and Boolean Operations** and **Tools Operations**.



The operations variable `ExtAPI.DataModel.GeometryBuilder.Operations` is the gateway for performing operations. The code must be integrated into the callback `<ongenerate>`.

The script function `ongenerate()` can be used to perform various types of operations on a geometry feature. The API provides primary and Boolean operations that allow you to specify properties such as body type, dimensions, shape, point coordinates, and material. It also offers a number of tools for manipulating your geometry, including actions such as copying or deleting a body or transform operations for creating a body based on another body or topological components.

Applying the Extrude Operation

When using the operation `Extrude`, you must first create the geometry to which the extrusion is to be applied. You can create any of the sheet primitive shapes listed in [Creating Primitives \(p. 232\)](#). Once the geometry has been created, you can define and apply the operation `Extrude` to it.

In the following example, the function `ongenerate()` is used to create a polygon sheet body and perform the operation `Extrude`.

```

def Ongenerate(feature,function):
    length = 0.3
    bodies = []

    builder = ExtAPI.DataModel.GeometryBuilder
    polygon=builder.Primitives.Sheet.CreatePolygon([0.,0.,3*length,0.,0.,2.*length,length,0.,2.*length])
    polygon_generated = polygon.Generate()
    extrude = builder.Operations.CreateExtrudeOperation([0.,1.,0.],length/2.)
    bodies.Add(extrude.ApplyTo(polygon_generated)[0])

    feature.Bodies = bodies
    feature.MaterialTypeEnum.Add

    return True
  
```

In this example:

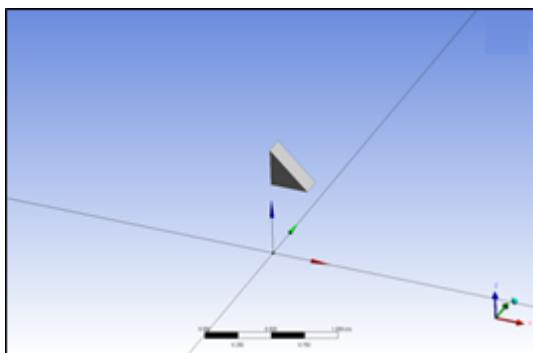
- The first part of the function creates a polygon sheet body, using a process similar to the one used to create a cylinder in [Creating a Sheet Body \(p. 233\)](#).
- With the operation `Extrude`, you use the operations generator `builder.Operations.CreateExtrudeOperation`. The method `CreateExtrudeOperation` specifies that the operation is defined by the following arguments:
 - Vector coordinates (which defines the direction of the extrusion)

- Length of the extrusion

Note

In this example, the variable **Length** is used for both the sheet body and the extrusion definition. However, you could have used a different variable for each.

- The method **ApplyTo** specifies the geometry to which to apply the operation **Extrude**. The method **ApplyTo()** returns a list of bodies to which to apply the operation.
- With the lines **bodies.Add(extrude.ApplyTo(polygon_generated)[0])** and **feature.Bodies = bodies**, you add the extruded sheet body polygon to the list **feature.bodies** so it is added to DesignModeler after generation. Bodies not added to this list are not retained.



Applying the Transform Edges to Wire Tool

When using the **Transform Edges to Wire** operation tool, you must first create a geometry that has some edges. You can create any of the sheet body or solid body shapes listed in [Creating Primitives \(p. 232\)](#). Once the primitive has been created, you can obtain a wire body from its edges by defining and applying the **Transform Edges to Wire** tool.

In the following example, the function **ongenerate()** creates a polygon sheet body and applies the **Transform Edges to Wire** tool.

```
def OnGenerate(feature, function):
    length = 0.3
    bodies = []

    builder = ExtAPI.DataModel.GeometryBuilder
    polygon = builder.Primitives.Sheet.CreatePolygon([0., 0., 2.*length, 0., 0., 1.*length, length, 0., 0.7])
    polygon_generated = polygon.Generate()
    body = builder.Operations.Tools.EdgesToWireBody(polygon_generated.Edges);
    bodies.Add(body)

    feature.Bodies = bodies
    feature.MaterialType = MaterialTypeEnum.Add

    return True
```

In this example:

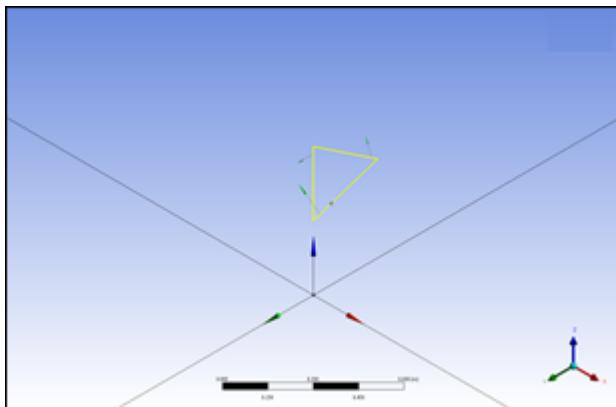
- The first part of the function creates a polygon sheet body, using a process similar to the one used to create a cylinder in [Creating a Sheet Body \(p. 233\)](#).

- With the body function, you use the operations generator `builder.Operations.Tools.EdgesToWireBody`. The method `EdgesToWireBody` specifies that the operation tool is to transform the edges of the sheet body polygon into a wire body.
-

Note

In this example, the variable `Length` is used for both the sheet body and the extrusion definition. However, you could have used a different variable for each.

- The new wire body is added to the list `feature.bodies` as described in [Applying the Extrude Operation \(p. 236\)](#).



APIs for ANSYS DesignXplorer

The following sections address the APIs provided by ACT to customize ANSYS DesignXplorer.

Related Topics:

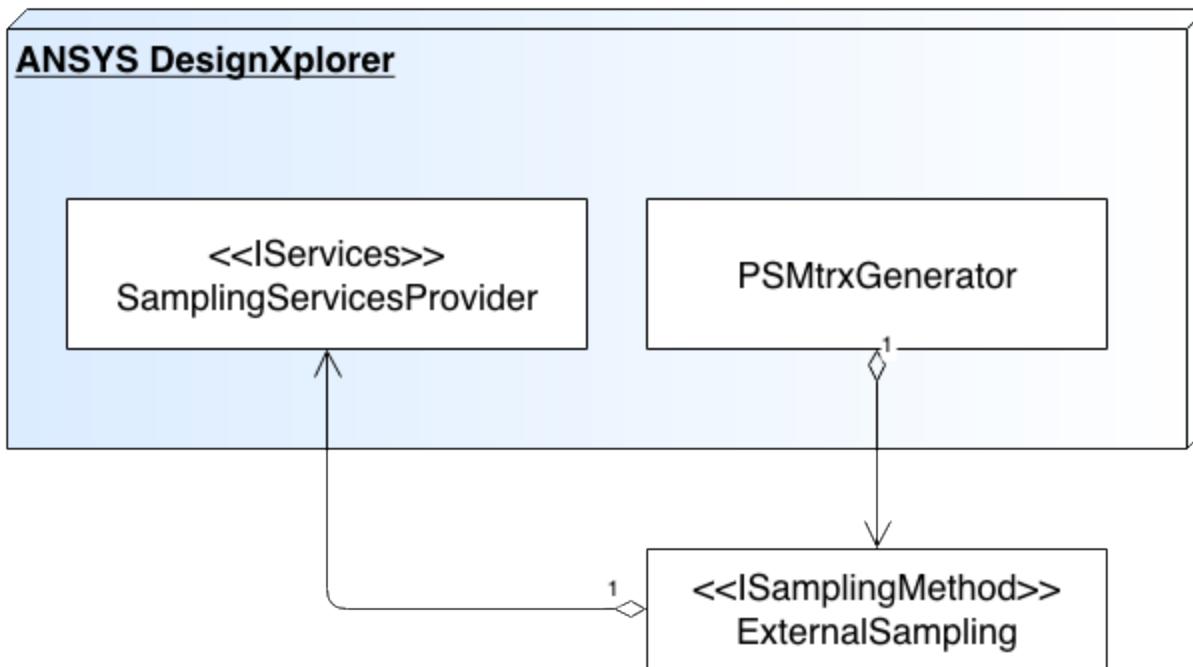
DOE APIs

This section discusses the APIs provided by ACT to customize DesignXplorer DOE functionality.

DOE Architecture

The object `PSMtrxGenerator` is responsible for the execution of the DOE. Regardless of whether the sampling is built-in or external, it executes the same process via the sampling API.

The following figure shows the DesignXplorer sampling architecture.



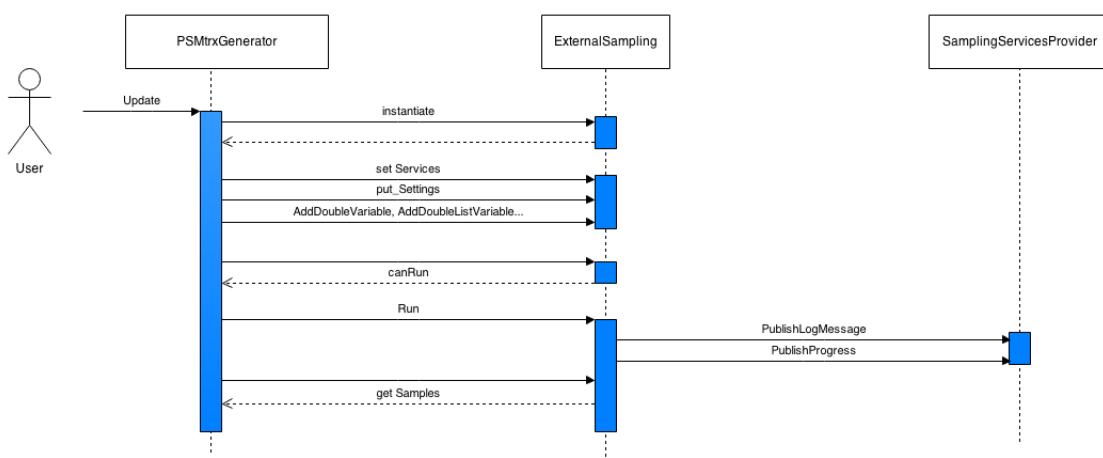
The main elements of the API are:

- **ISamplingMethod**: The main interface between DX and the external sampling, which is required to implement it
- **IServices**: The interface for the external sampling to access DX services, such as the calculation of one or more points
- **DXPoint**: The class that describes and stores the point data, such as variable values, calculation state, and so on.

For more information, see the "API Description" section for DesignXplorer in the [ANSYS ACT API Reference Guide](#).

Sampling Process

The **PSMtrxGenerator** object drives the sampling process as shown in the following sequence diagram.



First, the sampling process retrieves an instance of the sampling through the callback declared in the extension. It provides the **Services** object and transfers the complete sampling definition: all variables, parameter and settings.

The object **PSMtrxGenerator** retrieves an instance of the sampling each time the sampling process is started by invoking the callback **OnCreate**. The **PSMtrxGenerator** object releases the instance on completion by invoking the callback **OnRelease**.

After a final check **CanRun**, the sampling's method **Run** is invoked. From this point in the sequence, the external sampling triggers the calculation of points as needed and pushes progress messages and log messages, depending on its capabilities.

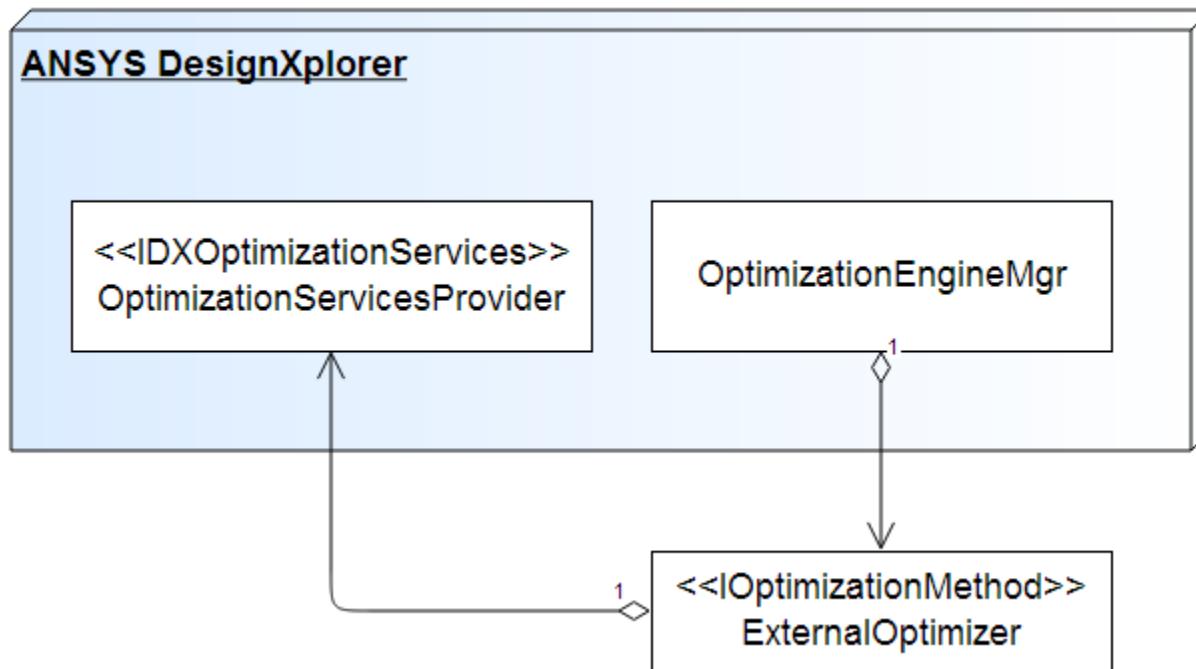
Optimization APIs

This section discusses the APIs provided by ACT to customize DesignXplorer optimization functionality.

Optimization Architecture

The object **OptimizationEngineMgr** is responsible for the execution of the optimization. Regardless of whether the optimizer is built-in or external, it executes the same process via the optimization API.

The following figure shows the DesignXplorer optimization architecture.



The main elements of the API are:

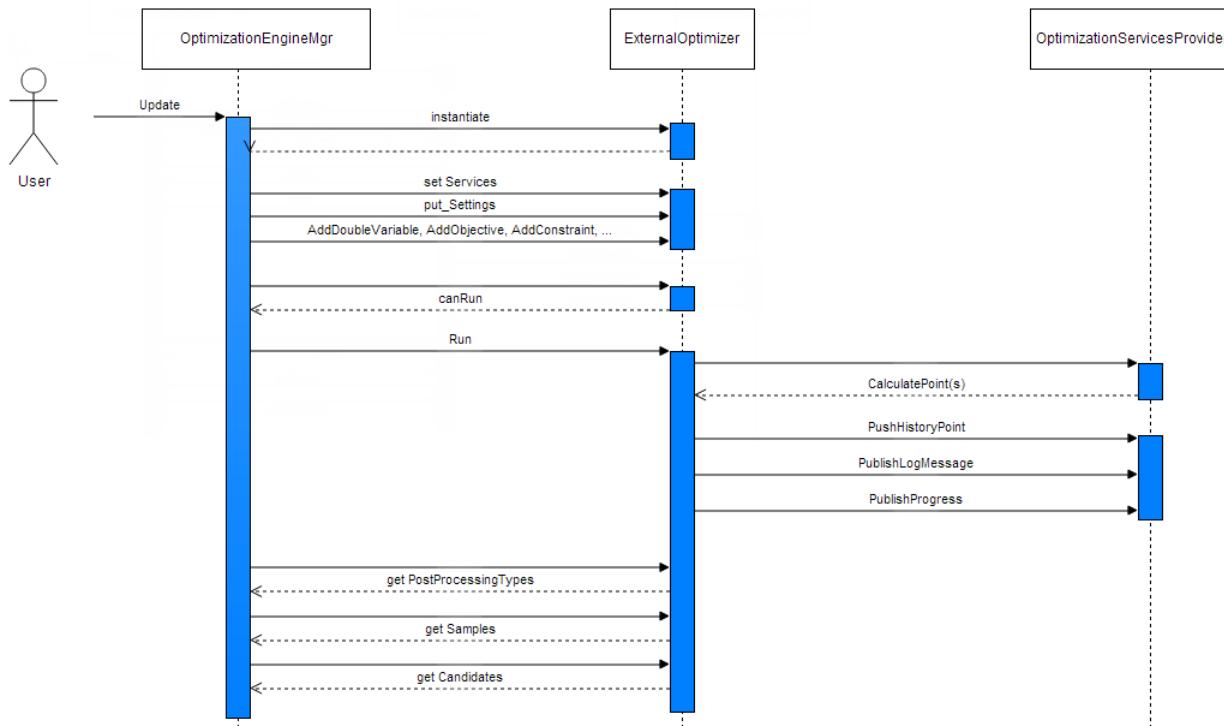
- **IOptimizationMethod**: The main interface between DesignXplorer and the external optimizer, which is required to implement it
- **IOptimizationServices**: The interface for the external optimizer to access DesignXplorer services, such as the calculation of one or more points

- **DXOptimizationPoint**: The class that describes and stores the points data, such as variable values, calculation state, and so on.

For more information, see the "API Description" section in the [ANSYS ACT API Reference Guide](#).

Optimization Process

The object **OptimizationEngineMgr** drives the optimization process as shown in the following sequence diagram.



First, the optimization process retrieves an instance of the optimizer through the callback declared in the extension. It provides the object **Services** and transfers the complete optimization study definition, which includes all variables, parameter relationships, objectives, constraints, and settings.

The object **OptimizationEngineMgr** retrieves an instance of the optimizer each time the optimization process is started by invoking the callback **OnCreate**. The object **OptimizationEngineMgr** releases the instance on completion by invoking the callback **OnRelease**.

After a final check **CanRun**, the optimizer's method **Run** is invoked. From this point in the sequence, the external optimizer triggers the calculation of points as needed and pushes progress messages, history points, convergence values, and log messages, depending on its capabilities.

Associated Libraries

This chapter describes the libraries installed with ACT. These libraries include a set of Python functions that can be used to develop extensions more efficiently.

The commands related to the examples provided can be interactively tested using the ACT Console. The ACT Console is installed with ACT. For more information, see [ACT Console \(p. 251\)](#).

The following sections describe a subset of the available libraries. You can access other libraries from the ANSYS installation directory and consult the source code for a reference on their use. Libraries can be found in the following folders in the %ANSYSversion_DIR%\\..\\Addins\\ACT\\libraries\\ directory:

- DesignModeler
- ElectronicsDesktop
- Fluent
- Mechanical
- Project
- SpaceClaim
- Study

Query to Material Properties

Description

This library allows you to access to all material properties defined in Engineering Data. The material is defined for each body and can be retrieved using the geometry API.

Location

- libraries/Mechanical/materials.py
- libraries/Project/materials.py
- libraries/Study/materials.py

Usage

```
import materials
```

Functions

```
GetListMaterialProperties(mat)
```

This function returns a list of property names for the material **mat** given in argument.

```
GetMaterialPropertyByName(mat, name)
```

This function returns the material property **name** for the material **mat**.

```
InterpolateData(vx, vy, vin)
```

This function computes a linear interpolation **vout = f(vin)** with **f** defined by **vy = f(vx)**.

- **vx** represents a list of values for the input variable.
- **vy** represents a list of values for the property that depends on the input variable defined by **vx**.
- **vin** is the value on which the function has to evaluate the property.

Examples

Assume you enter this command:

```
import materials  
  
mat = ExtAPI.DataModel.GeoData.Assemblies[0].Parts[0].Bodies[0].Material  
  
materials.GetListMaterialProperties(mat)
```

Results like these are returned:

```
['Compressive Ultimate Strength', 'Compressive Yield Strength', 'Density', 'Tensile Yield Strength',  
'Tensile Ultimate Strength', 'Coefficient of Thermal Expansion', 'Specific Heat', 'Thermal Conductivity',  
'Alternating Stress', 'Strain-Life Parameters', 'Resistivity', 'Elasticity', 'Relative Permeability']
```

Assume that you enter this command:

```
prop = materials.GetMaterialPropertyByName(mat, "Elasticity")  
  
prop
```

Results like these are returned:

```
{"Poisson's Ratio": ['', 0.2999999999999999, 0.2999999999999999, 0.2999999999999999],  
'Bulk Modulus': ['Pa', 1666666666.667, 175000000000.0, 18333333333.333],  
'Temperature': ['C', 10.0, 100.0, 1000.0], "Young's Modulus": ['Pa', 20000000000.0,  
21000000000.0, 22000000000.0], 'Shear Modulus': ['Pa', 76923076923.0769, 80769230769.2308, 84615384615.3846]}
```

Assume that you enter this command:

```
val = materials.InterpolateData(prop["Temperature"][1:], prop["Young's Modulus"][1:], 10.)  
  
val  
  
val = materials.InterpolateData(prop["Temperature"][1:], prop["Young's Modulus"][1:], 20.)  
  
val
```

Results like these are returned:

```
200000000000.0  
201111111111.0
```

Units Conversion

Description

This library implements a set of functions to manipulate the unit-dependent quantities within an extension. This library is of particular interest each time quantities have to remain consistent with the current unit system activated in the ANSYS product.

Location

- libraries/DesignModeler/materials.py
- libraries/ElectronicsDesktop/materials.py
- libraries/Fluent/materials.py
- libraries/Mechanical/materials.py
- libraries/Project/materials.py
- libraries/SpaceClaim/materials.py
- libraries/Study/materials.py

Usage

```
import units
```

Function

```
ConvertUnit(value,fromUnit,toUnit[,quantityName])
```

This function converts the **value** from the unit **fromUnit** to the unit **toUnit**. The user can specify the quantity name to avoid conflict during conversion. This quantity name is not mandatory.

Example

Assume that you enter this command:

```
import units
units.ConvertUnit(1,"m","mm")
```

A result like this is returned:

```
1000.0
```

Function

```
ConvertUserAngleUnitToDegrees(api, value)
```

This function converts the angle **value** to the unit degrees. If the current activated unit is already degrees, then the value remains unchanged.

Example

Assume that you enter this command:

```
import units
units.ConvertUserAngleUnitToDegrees(api,3.14)
```

A result like this is returned if the angle unit is set to radius when the command is called:

180

A result like this is returned if the angle unit is set to degrees when the command is called:

3.14

Function

```
ConvertToSolverConsistentUnit(api, value, quantityName, analysis)
```

This function converts the **value** of the quantity **quantityName** from the currently activated unit in the ANSYS product to the corresponding consistent unit used by the solver for the resolution of the analysis **analysis**.

Example

Assume that you enter this command:

```
import units
units.ConvertToSolverConsistentUnit(api,1.,"pressure","Static Structural")
```

A result like this is returned if the unit system is set to Metric(mm,dat,N,s,mV,mA) when the command is called:

10

Function

```
ConvertToUserUnit(api, value, fromUnit, quantityName)
```

This function converts the **value** of the quantity **quantityName** from the unit **fromUnit** to the currently activated unit system in the ANSYS product.

Example

Assume that you enter this command:

```
import units
units.ConvertToUserUnit(api,1.,"m","length")
```

A result like this is returned if the current activated unit is millimeter when the command is called:

1000

Function

```
ConvertUnitToSolverConsistentUnit(api, value, fromUnit, quantityName, analysis)
```

This function converts the **value** of the quantity **quantityName** from the unit **fromUnit** to the consistent unit that is used by the solver for the resolution of the analysis **analysis**.

Example

Assume that you enter this command:

```
import units
units.ConvertUnitToSolverConsistentUnit(api,1.,"m","length","Static Structural")
```

A result like this is returned if the consistent unit is millimeter when the command is called:

```
1000
```

Function

```
GetMeshToUserConversionFactor(api)
```

This function returns the scale factor to be applied to convert a length from the unit associated with the mesh and the currently activated unit in the ANSYS product.

Example

Assume that you enter this command:

```
import units
units.GetMeshToUserConversionFactor(api)
```

A result like this is returned if the unit associated with the mesh is meter and if the current activated unit in the application is millimeter:

```
1000
```

MAPDL Helpers

Description

This library implements some helpers to write APDL command blocks or to execute ANSYS Mechanical APDL programs.

Location

- libraries/ElectronicsDesktop/ansys.py
- libraries/Fluent/ansys.py
- libraries/Mechanical/ansys.py
- libraries/Project/ansys.py
- libraries/Study/ansys.py

Usage

```
import ansys
```

Functions

```
createNodeComponent(refIds,groupName,mesh,stream,fromGeoIds=True)
```

This function writes the APDL command block (CMBLOCK) for the creation of a node component related to the geometry entities identified by **refIds** into the stream **stream**. "refIds" refer to geometric IDs if "fromGeoids" is true and the node IDs are retrieved from geometric entities using the associativity

between the geometry and mesh. If “fromGeoids” is false, then “refIds” refer directly to the node IDs to be written in the component.

```
createElementComponent(refIds,groupName,mesh,stream,fromGeoids=True)
```

This function writes the APDL command block (CMBLOCK) for the creation of an element component related to the geometry entities identified by “refIds” into the stream “stream”. “refIds” refer to geometric IDs if “fromGeoids” is true and the element IDs are retrieved from geometric entities using the associativity between the geometry and the mesh. If “fromGeoids” is false, the “refIds” refer directly to the element IDs to be written in the component.

```
RunANSYS(api,args,[runDir[,exelocation]])
```

This function calls the ANSYS Mechanical APDL program with the command line initialized by **args**. The user can specify the folder from which he wants to execute the program with **runDir**. The location of the executable is also managed with **exelocation**.

The parameter **api** must be **ExtAPI**.

Example

```
import ansys
ansys.createNodeComponent([refId],"myGroup",mesh,stream)
ansys.RunANSYS(ExtAPI,"")
```

Journaling Helper

Description

This library implements a helper that can be used to transfer data between Mechanical and the Project page.

Location

- libraries/Mechanical/ansys.py
- libraries/Project/ansys.py
- libraries/Study/ansys.py

Usage

```
import wbjn
```

Functions

```
ExecuteCommand(api,cmd,**args)
```

This function executes a journal command specified by **cmd**. You can get a result object by using the function **returnValue(obj)** in your journal command. All arguments must implement the serialization interface provided by .Net. The object sent to the function **returnValue(obj)** must also implement the serialization interface. This interface is already implemented for many standard types of data (integer, double, list...). Note that the standard Python dictionary does not implement this interface by default. If a dictionary is required, use the class **SerializableDictionary** provided by ACT.

The parameter **api** must be **ExtAPI**.

```
ExecuteFile(api,file,**args)
```

The same as above but executes a journal file specified by **file**.

Examples

The following commands return **5** as the result:

```
import wbjn  
wbjn.ExecuteCommand(ExtAPI,"returnValue(a+b)",a=2,b=3)
```

The following commands return **My first system is: Static Structural !**:

```
import wbjn  
wbjn.ExecuteCommand(ExtAPI,"returnValue(a+ GetAllSystems()[0].DisplayText+b)" ,  
a="My first system is: ",b=" !")
```

The following commands return **3** as the result:

```
import wbjn  
dict = SerializableDictionary[str,int]()  
dict.Add("e1",1)  
dict.Add("e2",2)  
wbjn.ExecuteCommand(ExtAPI,'returnValue(d["e1"]+d["e2"])', d=dict)
```


Development and Debugging

This section provides information that is helpful during the development and testing of extensions.

When creating an extension, you use the IronPython programming language to develop the functions used to execute the extension. Comprehensive IronPython documentation is available at [IronPython.net](#). Additionally, David Beazley's book, *Python Essential Reference*, is highly recommended. This book is considered the definitive reference guide to the Python language.

The following development and debugging functionality is available:

- [ACT Console](#)
- [Binary Extension Builder](#)
- [Debug Mode](#)
- [Debugging with Microsoft® Visual Studio](#)

Note

Most development and debugging tools also have at least one alternate method of access. Where applicable, these alternate access points are noted in the discussion of the tool.

ACT Console

The **ACT Console** is a tool for interactively testing commands during the development or debugging of an extension. It is available by default in ANSYS Workbench, AIM, and Mechanical.

In Workbench or AIM:

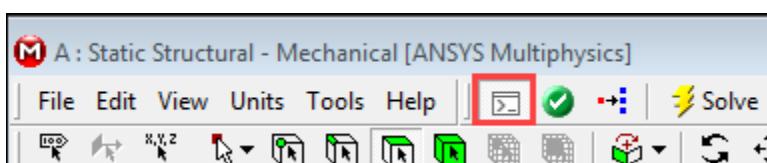
You can open the **ACT Console** in one of two ways:

- Click the corresponding icon on the **ACT Start Page**.
- Select **Extensions > View ACT Console** from the menu.

The **ACT Console** window can be resized, repositioned, and kept open as you switch between products.

In Mechanical:

You can open and close the **ACT Console** by clicking the **View ACT Console** toolbar button.



The **ACT Console** is initially exposed in a locked position to the right of the graphics view. However, you can drag it to anywhere in Mechanical and drop it into a new locked position. By double-clicking

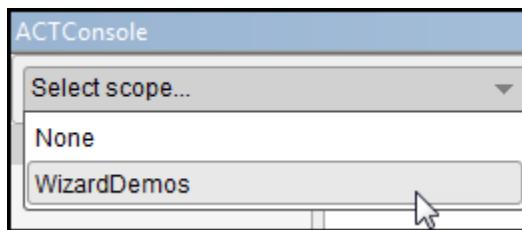
its header bar, you can toggle between the locked position and a floating window. The floating window can be resized and moved as needed.

Related Topics:

- [Using the Scope Menu](#)
- [Using the Command Line](#)
- [Using the Command History](#)
- [Using Autocompletion](#)
- [Using Snippets](#)
- [ACT Console Keyboard Shortcuts](#)

Using the Scope Menu

Each extension runs with a dedicated script engine. To access global variables or functions associated with a given extension, you must first select the extension from the drop-down list for the **Select scope** option. This list is populated with the names of all extensions that are currently loaded for use with the active ANSYS product. The following image shows a possible list for **Select scope** when the **ACT Console** is opened from Mechanical.



Using the Command Line

The command line allows you to enter multiple lines of code and offers features like automatic indentation, text coloration, and matching-bracket highlighting.

You can enter commands either by typing them, pasting them in, or clicking a snippet. As you type commands, you can make use of command-entry assistance capabilities such as autocompletion.

You use the following key combinations when entering commands in the command line:

Key Combination	Action
Shift + Enter	Insert a new line
Enter	Execute the commands

Executed commands are displayed as entries in the command history.

For available keyboard shortcuts, see [ACT Console Keyboard Shortcuts \(p. 263\)](#).

Using the Command History

When you execute commands, the entries that display in the command history are color-coded by item type. To work with these entries, you use key combinations, icons, and context menu options.

History Color-Coding

In a command history entry, you identify the type of each item by its text color:

Text Color	Item Type
Black	Inputs
Blue	Outputs
Red	Errors

History Key Combinations

To move between command history entries, you use the following key combinations:

Key Combination	Action
Ctrl + up-arrow	Go to the previous entry
Ctrl + down-arrow	Go to the next entry

For available keyboard shortcuts, see [ACT Console Keyboard Shortcuts \(p. 263\)](#).

History Icons

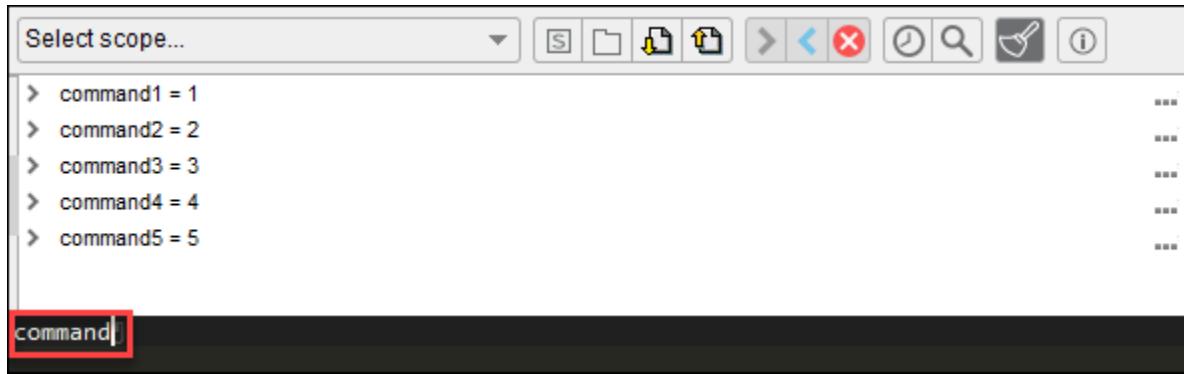
To work with command history entries, you use toolbar icons:



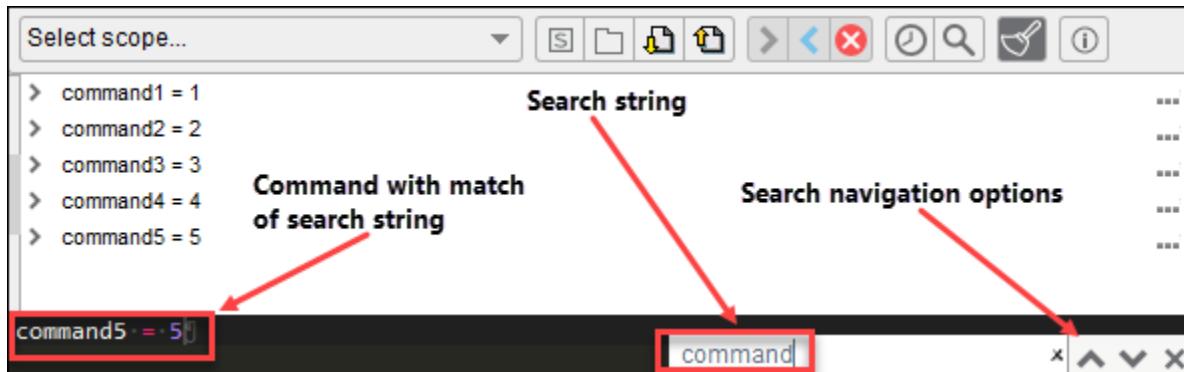
1. Show or hide the history of inputs.
2. Show or hide the history of outputs.
3. Show or hide the history of errors.
4. Display the last executed commands (maximum of 500).
5. Search command history for an entered search string.
6. Clear the command history log.
7. Open the **Help** window.
8. Show hidden icons (visible only when the **ACT Console** window is not wide enough to display all icons in the toolbar).

History Search

When you click the search toolbar icon to search the command history, a reverse search is performed using the text currently entered in the command line. The following figure shows five executed commands and **command** entered in the command line.

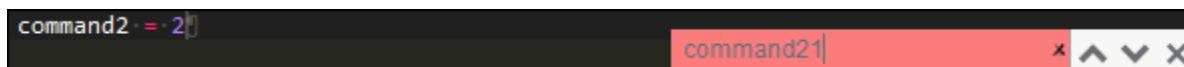


To start the search, either click the search toolbar icon or press **Ctrl + R**. A search box opens to the right of the command line. Besides displaying the current search string, the search box provides buttons for stepping backward and forward through the command history entries and a button for closing the search box. When the search starts, it immediately steps backward through the command history, stopping when it finds an entry with a match. In this example, the search stops when it finds **command5 = 5**, displaying this entry in the command line. If you wanted to execute this command again, you would press **Enter**.



To resume the reverse search, you either press **Ctrl + R** or click the up arrow button for the search box. To change the search direction to step forward rather than backward in the history, you either press **Ctrl + Shift + R** or click the down arrow button for the search box. To clear the search string, you click the button with the small X to the right of the search string.

As long as the result in the command line matches the current search string, no change occurs. When the result no longer matches the search string, the search automatically steps backward to find the next match. Assume that you enter **command2** as the search string. Because **command5 = 5** no longer matches **command2**, the search steps back through the command entries to find a match, displaying **command2 = 2** in the command line. Now assume that you enter **command21** as the search string. The search box turns red because no match exists in the command history.



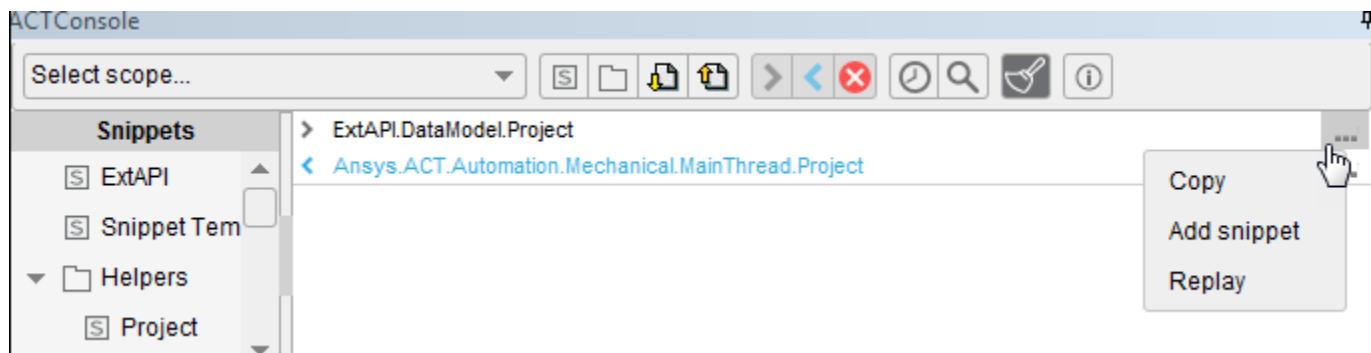
If you enter **command** as the search string once again, the search box is no longer red. Because **command2 = 2** still matches the current search string, it remains displayed in the command line.

If the search reaches the end of the history without finding a match, it wraps around to the beginning of the command history. When you finish searching, you click the close button for the search box.

History Context Menu

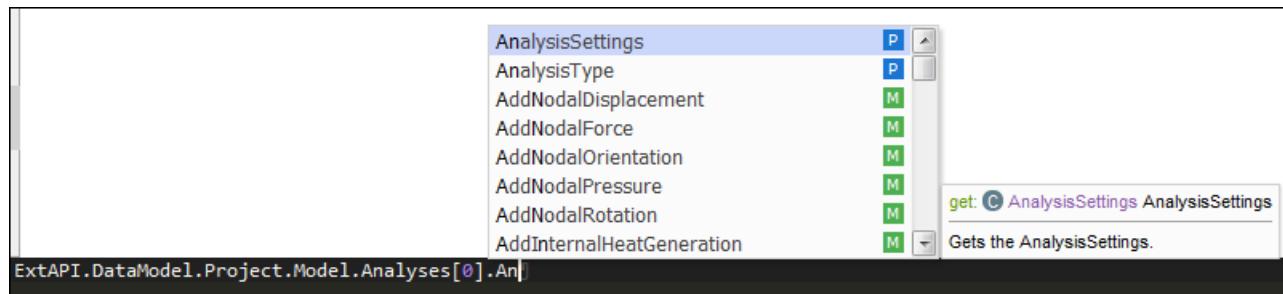
For a command history entry, you can access a drop-down menu and select an action to apply to it. When you click the icon with the three dots to the right of an entry, the following actions are available:

- **Copy:** Copy the command text to the clipboard.
- **Add snippet:** Add the command text to the **Snippets** pane. For more information, see [Using Snippets \(p. 258\)](#).
- **Replay:** Paste the command text into the command line so that it can be executed again later.



Using Autocompletion

The **ACT Console** provides "smart" autocompletion, which means it takes syntax elements such as brackets or quotations marks into account. As you type in the command line, a scrollable list of suggestions is displayed above the command line.



Each suggestion has an icon with a letter and color-coding to indicate the type of suggestion. A letter in a square icon describes the type of the member in the suggestion list. A letter in a circle icon describes the nature of the return type.

Listed alphabetically are types of members, which are shown in square icons:

- **E** = Enumeration
- **M** = Method
- **N** = Namespace
- **P** = Property
- **S** = Snippet

- **T** = Type
- **V** = Variable
- **-** = Unidentified member (generally a Python type that cannot be extracted)

Once a member is selected, the nature of the return types available for this member are shown in circle icons. Listed alphabetically are return types:

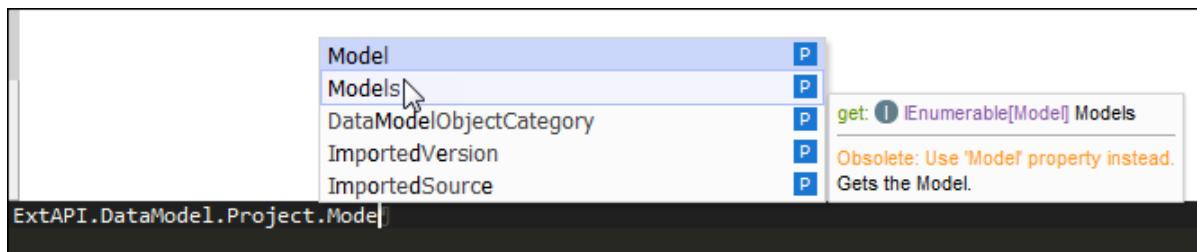
- **C** = Class
 - **E** = Enumeration
 - **F** = Field
 - **I** = Interface
 - **S** = Structure
-

Note

No icon displays for primitive types (integer, Boolean, string, and so on) or if no suggestion is returned.

Using Autocompletion Tooltips

When you place the mouse cursor over any property or method in the list of suggestions, the tooltip provides information about this item. The following image shows the tooltip for the property **Models**, which is available when using the **ACT Console** with a project model associated with ANSYS Mechanical.



Tooltips use color-coding to indicate the syntax:

- **green** = accessibility
- **purple** = type
- **orange** = warning
- **blue** = arguments

General formatting follows for properties and methods.

Properties:

- **get/set mode: ReturnType PropertyName**

- Description of the property.
- **Returns:** Description of what is returned (if any)
- **Remarks:** Additional information (if any)
- **Example:** Sample entry (if any)

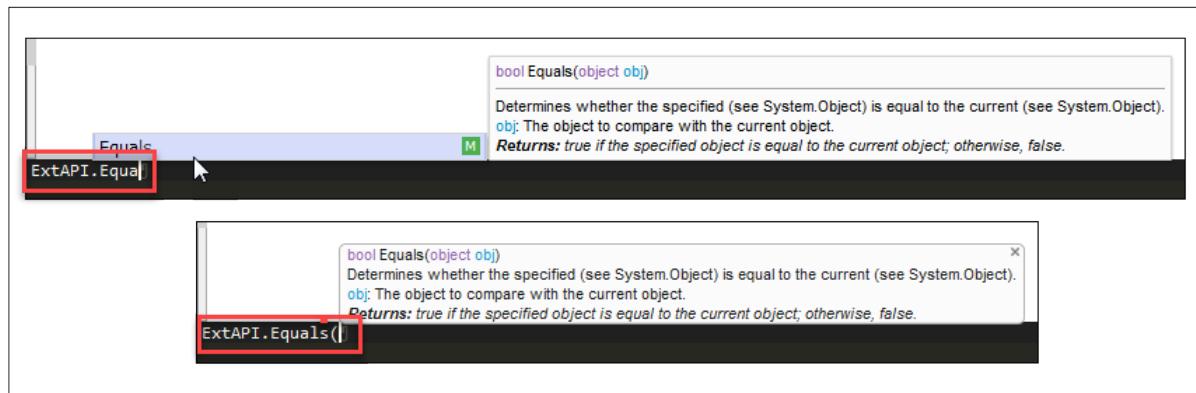
Methods:

- **ReturnType MethodName (ArgumentType ArgumentName)**
- Description of the method
- **ArgumentName:** Description of the argument
- **Returns:** Description of what is returned (if any)
- **Remarks:** Additional information (if any)
- **Example:** Sample entry (if any)

Tooltips can also provide:

- **Scope variables** (accessed by pressing **Ctrl + space** when no defined symbol appears under the text cursor).
- **.NET properties** where applicable.
- **Warning messages** when special attention is needed.
- **Prototype information** for methods when cursor is inside brackets and indexers when cursor is inside square brackets.
- **Overloaded methods**, including the following details:
 - **Number** of overloaded methods.
 - **Prototypes** for overloaded methods (accessed by pressing the arrows in the prototype tooltip).
 - **Members** for overloaded methods. (The tooltip for an overloaded method is a list of all members from all overloaded methods.)

The following image shows examples of two different tooltips for the method **Equals** in two different stages of using autocomplete.



Interacting with Autocompletion Suggestions

You can use the following key combinations to interact with autocompletion suggestions:

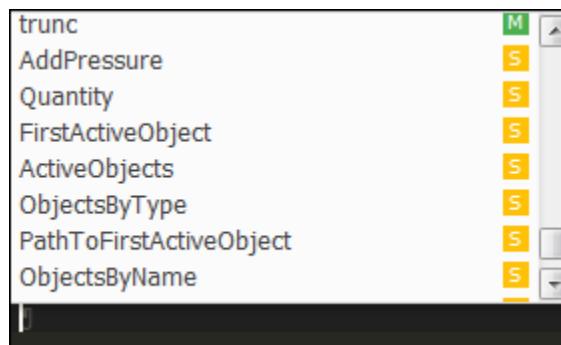
Key Combination	Action
Ctrl + space	Open the suggestion tooltip
Esc	Close the suggestion tooltip
Enter	Apply the suggestion
space or symbol Supported symbols: . () [] {} '' # , ; : ! / = ?	Apply the suggestion followed by the space or symbol

For available keyboard shortcuts, see [ACT Console Keyboard Shortcuts \(p. 263\)](#).

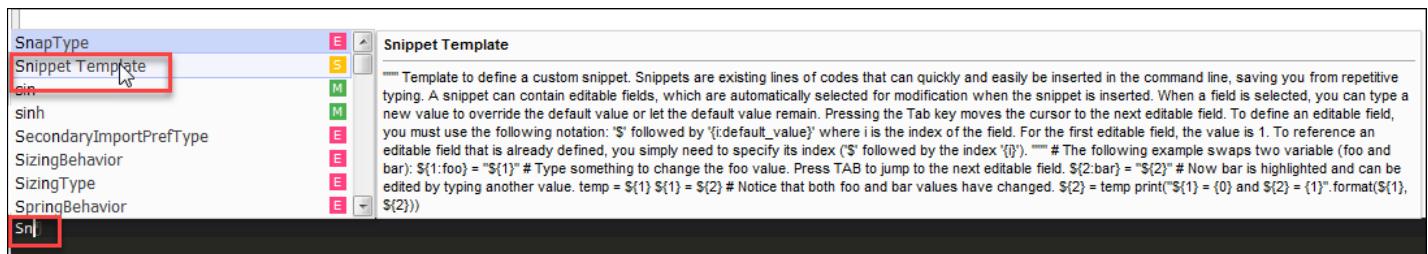
Using Snippets

Snippets are existing lines of code that can quickly and easily be inserted in the command line, saving you from repetitive typing. A snippet can contain editable fields, which are automatically selected for modification when the snippet is inserted. When a field is selected, you can type a new value to override the default value or let the default value remain. Pressing the **Tab** key moves the cursor to the next editable field.

When nothing is entered in the command line, pressing **Ctrl + space** opens a list of suggestions displaying all variables and snippets. When you scroll down to the snippets, you can see those predefined for standard Python commands as well as those in the **Snippets** panel.



If you know the name of the snippet, you can begin typing its name to find it in the list of suggestions. The predefined **Snippet Template** provides code for swapping two variables.



When you insert a snippet in the command line, all editable fields can easily be seen.

```
""" Template to define a custom snippet.#
Snippets are existing lines of codes that can quickly and easily be inserted in the command line, saving you from repetitive typing.#
A snippet can contain editable fields, which are automatically selected for modification when the snippet is inserted.#
When a field is selected, you can type a new value to override the default value or let the default value remain.#
Pressing the Tab key moves the cursor to the next editable field.#
To define an editable field, you must use the following notation: '$' followed by '{i:default_value}' where i is the index of the field.#
For the first editable field, the value is 1.#
To reference an editable field that is already defined, you simply need to specify its index ('$' followed by the index '{i}').#
"""
# The following example swaps two variables (foo and bar):#
foo = "foo" # Type something to change the foo value. Press TAB to jump to the next editable field.#
bar = "bar" # Now bar is highlighted and can be edited by typing another value.#
temp = foo#
foo = bar # Notice that both foo and bar values have changed.#
bar = temp#
print("foo = {0} and bar = {1}".format(foo, bar))
```

For example, in the **Snippet Template** snippet, the first editable field (**foo**) is highlighted.

```
foo = "foo" #
bar = "bar" #
temp = foo #
foo = bar # N
bar = temp #
```

Typing something changes the **foo** value to whatever you type (**value1**). Notice that both **foo** values change to **value1** in one operation.

```
value1 = "value1" #
bar = "bar" # Now
temp = value1 #
value1 = bar # Not
bar = temp #
```

Pressing the **Tab** key moves the cursor to the next editable field, causing **bar** to be highlighted.

```
value1 = "value1" #
bar = "bar" # Now
temp = value1 #
value1 = bar # Not
bar = temp #
```

Typing something changes the **bar** value to whatever you type (**value2**). Notice that both **bar** values change to **value2** in one operation.

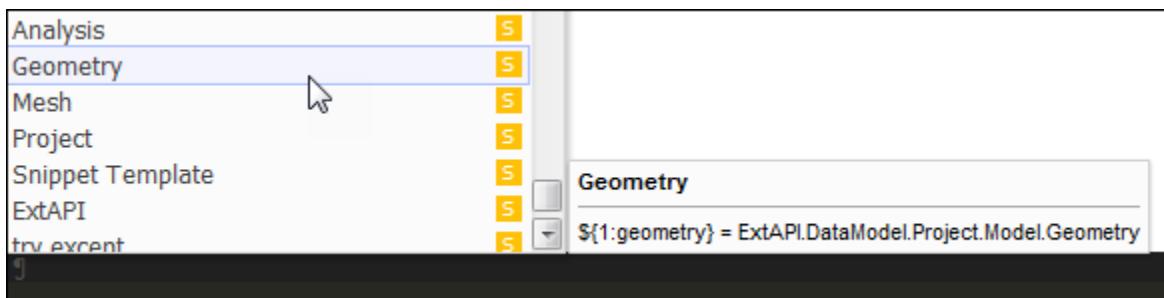
```

value1 = "value1"
value2 = "value2"
temp = value1
value1 = value2 #
value2 = temp

```

Pressing the **Tab** key again finalizes the code.

To define an editable field, you must use the following notation: `#{#:default_value}` where # is the index of the field. For the first editable field, the value is 1. To reference an editable field that is already defined, you simply need to specify its index (`#{#}`), as shown in the following figure for the **Geometry** snippet.



For the previous example for swapping two variables, you can define the following snippet:

```

temp = ${1:foo}
${1} = ${2:bar}
${2} = temp

```

Creating and Managing Snippets

From the **Snippets** pane, you can easily insert a snippet in the command line by clicking it. This pane displays all personal snippets, which are either those supplied by ANSYS for ACT-specific commands or those that you create. It does not display snippets for basic Python commands because these snippets cannot be removed. For example, the **Snippets** pane displays the supplied **Snippet Template** for swapping two variables, but it does not display the **lambda** snippet for this basic Python command.

Note

When typing in the command line, autocompletion displays all snippets in the **Snippet** pane as well as the snippets for basic Python commands.

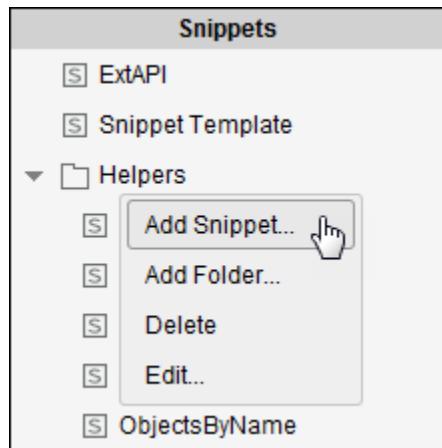
To make creating and managing snippets easy, the **Snippets** pane provides toolbar icons and a context menu.

Toolbar Icons:



1. Add a new snippet.
2. Add a new snippet folder.
3. Import an existing snippets collection.
4. Export the current snippets collection.
5. Show hidden icons; only displays when the **ACT Console** window is not wide enough to display all icons in the toolbar.

Context Menu:



Creating Snippets and Folders

You can create snippets and folders using any of the following methods:

- Click the icon with the three dots to the right of an entry in the command history and select **Add snippet**.
- Click the toolbar icon for adding a new snippet or folder.
- Right-click a snippet or folder in the **Snippets** pane and select **Add Snippet** or **Add Folder**. Where you right-click determines where the new snippet or new folder is placed in the **Snippets** pane.
 - When you right-click a folder, the new item is created inside this folder.
 - When you right-click a snippet, the new item is created at the bottom of the containing folder.

Once the snippet or folder is created, the **Snippet Properties** dialog box opens so that you can specify properties. A snippet has two properties, **Caption** and **Value**. A folder has only **Caption**.

- **Caption** specifies the text to display as the name for the snippet or folder.
- **Value** specifies the code that the snippet is to insert in the command line. When a snippet is created from an entry in the command history, **Value** displays the code for this entry by default.

After specifying the properties for the snippet or folder, you click **Apply** to save your changes or **Cancel** to discard the newly created snippet or folder.

Viewing and Editing Snippet or Folder Properties

To view or edit the properties of an existing snippet or folder in the **Snippets** pane, right-click it and selecting **Edit** to open the **Snippet Properties** dialog box. When finished, click **Apply** to save changes or **Cancel** to exit without saving changes.

Deleting Snippets and Folders

You can delete a snippet or folder from the **Snippets** pane by right-clicking it and selecting **Delete**.

Caution

If you delete a folder, all of its contents are also deleted. Deleting a snippet from the **Snippets** pane also removes it from the list of suggestions for autocompletion.

Organizing Snippets and Folders

When creating a snippet or folder, you can determine its location by using a particular creation method, as described in [Creating Snippets and Folders \(p. 261\)](#). To organize existing snippets and folders in the **Snippets** pane, you can drag them to the desired location in the hierarchical tree structure.

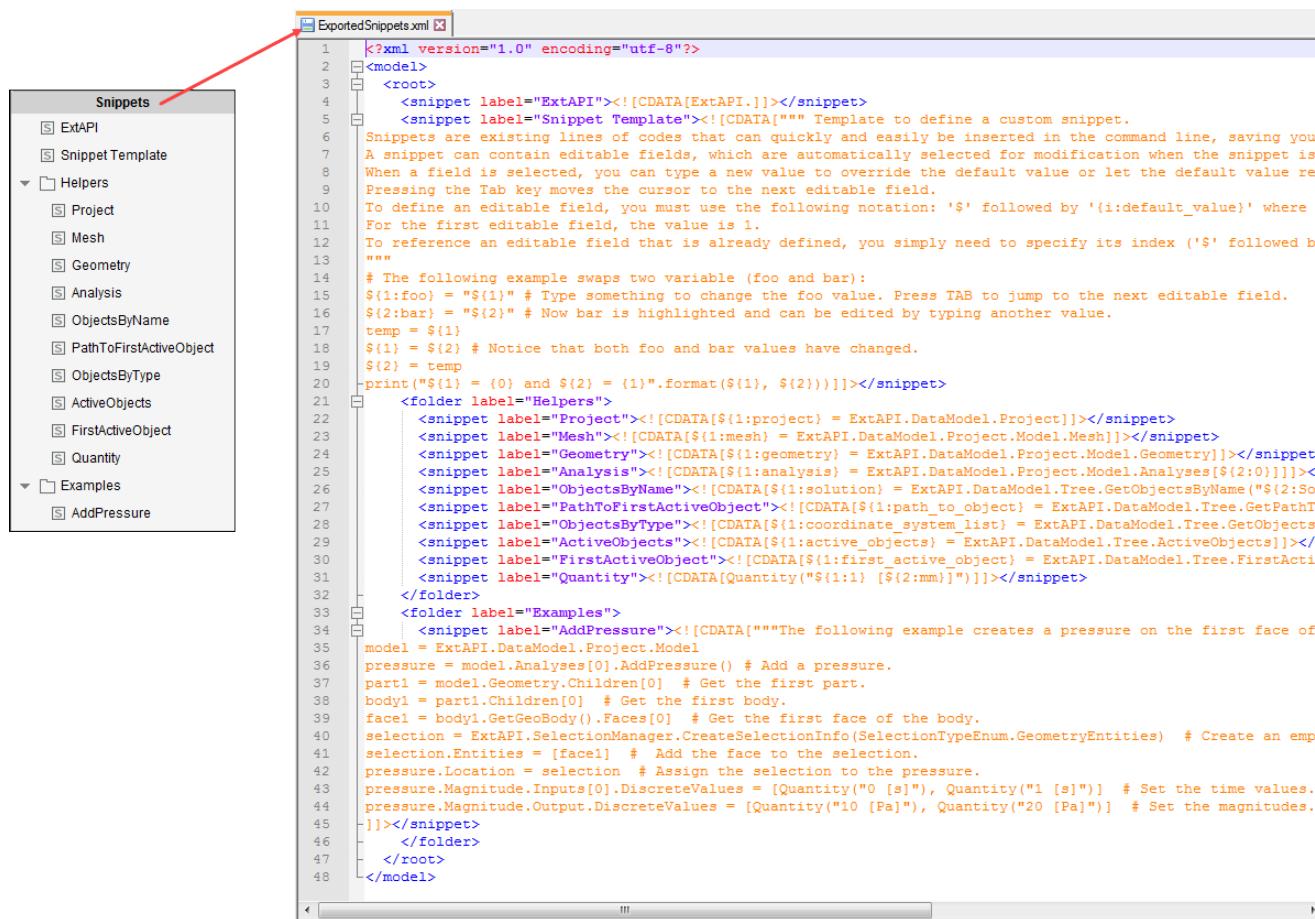
By default, the **Snippets** pane contains a **Helpers** folder and an **Examples** folder. The **Helpers** folder contains snippets for frequently used commands. The **Examples** folder contains snippets for use as templates. For example, the **AddPressure** snippet provides an example of how to create a pressure on the first face of the first body for the first part.

Importing and Exporting a Snippets Collection

You can import or export a collection of snippets as an XML file.

- Clicking the toolbar icon for importing a snippets collection () appends the snippets in the selected XML file to the list already shown in the **Snippets** pane.
- Clicking the toolbar icon for exporting a snippets collection () exports all snippets in the **Snippets** pane to the specified XML file.

The following figure shows the XML file that results from the export of a snippets collection.



```

ExportedSnippets.xml
1  <?xml version="1.0" encoding="utf-8"?>
2  <model>
3  |   <root>
4  |   |   <snippet label="ExtAPI"><![CDATA[ExtAPI.]]></snippet>
5  |   |   <snippet label="Snippet Template"><![CDATA["" Template to define a custom snippet.
6  |   |   Snippets are existing lines of codes that can quickly and easily be inserted in the command line, saving you
7  |   |   A snippet can contain editable fields, which are automatically selected for modification when the snippet is
8  |   |   When a field is selected, you can type a new value to override the default value or let the default value remain.
9  |   |   Pressing the Tab key moves the cursor to the next editable field.
10 |   |   To define an editable field, you must use the following notation: '$' followed by '{i:default_value}' where i
11 |   |   For the first editable field, the value is 1.
12 |   |   To reference an editable field that is already defined, you simply need to specify its index ('$' followed by
13 |   |   """
14 |   |   # The following example swaps two variables (foo and bar):
15 |   |   ${1:foo} = "${1}" # Type something to change the foo value. Press TAB to jump to the next editable field.
16 |   |   ${2:bar} = "${2}" # Now bar is highlighted and can be edited by typing another value.
17 |   |   temp = ${1}
18 |   |   ${1} = ${2} # Notice that both foo and bar values have changed.
19 |   |   ${2} = temp
20 |   |   print("${1} = {0} and ${2} = {1}".format(${1}, ${2}))]]></snippet>
21 |   |   <folder label="Helpers">
22 |   |       <snippet label="Project"><![CDATA[$1:project] = ExtAPI.DataModel.Project]]></snippet>
23 |   |       <snippet label="Mesh"><![CDATA[$1:mesh] = ExtAPI.DataModel.Project.Model.Mesh]]></snippet>
24 |   |       <snippet label="Geometry"><![CDATA[$1:geometry] = ExtAPI.DataModel.Project.Model.Geometry]]></snippet>
25 |   |       <snippet label="Analysis"><![CDATA[$1:analysis] = ExtAPI.DataModel.Project.Model.Analyses[$2:0]]]></snippet>
26 |   |       <snippet label="ObjectsByName"><![CDATA[$1:solution] = ExtAPI.DataModel.Tree.GetObjectsByName("${2:Sol}]]></snippet>
27 |   |       <snippet label="PathToFirstActiveObject"><![CDATA[$1:path_to_object] = ExtAPI.DataModel.Tree.GetPathToFirstActiveObject]]></snippet>
28 |   |       <snippet label="ObjectsByType"><![CDATA[$1:coordinate_system_list] = ExtAPI.DataModel.Tree.GetObjectsByType[$1:type]]></snippet>
29 |   |       <snippet label="ActiveObjects"><![CDATA[$1:active_objects] = ExtAPI.DataModel.Tree.ActiveObjects]]></snippet>
30 |   |       <snippet label="FirstActiveObject"><![CDATA[$1:first_active_object] = ExtAPI.DataModel.Tree.FirstActiveObject]]></snippet>
31 |   |       <snippet label="Quantity"><![CDATA[Quantity("${1:1} (${2:mm})")]]></snippet>
32 |   |   </folder>
33 |   |   <folder label="Examples">
34 |   |       <snippet label="AddPressure"><![CDATA[""The following example creates a pressure on the first face of
35 |   |       model = ExtAPI.DataModel.Project.Model
36 |   |       pressure = model.Analyses[0].AddPressure() # Add a pressure.
37 |   |       parti = model.Geometry.Children[0] # Get the first part.
38 |   |       body1 = parti.Children[0] # Get the first body.
39 |   |       face1 = body1.GetGeoBody().Faces[0] # Get the first face of the body.
40 |   |       selection = ExtAPI.SelectionManager.CreateSelectionInfo(SelectionTypeEnum.GeometryEntities) # Create an empty
41 |   |       selection.Entities = [face1] # Add the face to the selection.
42 |   |       pressure.Location = selection # Assign the selection to the pressure.
43 |   |       pressure.Magnitude.Inputs[0].DiscreteValues = [Quantity("0 [s]"), Quantity("1 [s]")] # Set the time values.
44 |   |       pressure.Magnitude.Output.DiscreteValues = [Quantity("10 [Pa]"), Quantity("20 [Pa]")] # Set the magnitudes.
45 |   |   -]]></snippet>
46 |   |   </folder>
47 |   | </root>
48 | </model>

```

ACT Console Keyboard Shortcuts

The following tables list keyboard shortcuts for the **ACT Console**.

Table 2: Line Operations

Key Combination	Action
Ctrl + D	Remove line
Alt + Shift + down-arrow	Copy lines down
Alt + Shift + up-arrow	Copy lines up
Alt + down-arrow	Move lines down
Alt + up-arrow	Move lines up
Alt + Backspace	Remove to line end
Alt + Delete	Remove to line start
Ctrl + Delete	Remove word left
Ctrl + Backspace	Remove word right

Table 3: Selection

Key Combination	Action
Ctrl + A	Select all
Shift + left-arrow	Select left

Key Combination	Action
Shift + right-arrow	Select right
Ctrl + Shift + left-arrow	Select word left
Ctrl + Shift + right-arrow	Select word right
Shift + Home	Select line start
Shift + End	Select line end
Alt + Shift + right-arrow	Select to line end
Alt + Shift + left-arrow	Select to line start
Shift + up-arrow	Select up
Shift + down-arrow	Select down
Shift + Page Up	Select page up
Shift + Page Down	Select page down
Ctrl + Shift + Home	Select to start
Ctrl + Shift + End	Select to end
Ctrl + Shift + D	Duplicate selection
Ctrl + Shift + P	Select to matching bracket

Table 4: Multi-Cursor

Key Combination	Action
Ctrl + Alt + up-arrow	Add multi-cursor above
Ctrl + Alt + down-arrow	Add multi-cursor below
Ctrl + Alt + right-arrow	Add next occurrence to multi-selection
Ctrl + Alt + left-arrow	Add previous occurrence to multi-selection
Ctrl + Alt + Shift + up-arrow	Move multi-cursor from current line to the line above
Ctrl + Alt + Shift + down-arrow	Move multi-cursor from current line to the line below
Ctrl + Alt + Shift + right-arrow	Remove current occurrence from multi-selection and move to next
Ctrl + Alt + Shift + left-arrow	Remove current occurrence from multi-selection and move to previous
Ctrl + Shift + L	Select all from multi-selection

Table 5: Go-To

Key Combination	Action
Page Up	Go to page up
Page Down	Go to page down
Ctrl + Home	Go to start
Ctrl + End	Go to end
Ctrl + L	Go to line

Key Combination	Action
Ctrl + P	Go to matching bracket

Table 6: Folding

Key Combination	Action
Alt + L, Ctrl + F1	Fold selection
Alt + Shift + L, Ctrl + Shift + F1	Unfold

Table 7: Other

Key Combination	Action
Tab	Indent
Shift + Tab	Outdent
Ctrl + Z	Undo
Ctrl + Shift + Y, Ctrl + Y	Redo
Ctrl + T	Transpose letters
Ctrl + Shift + U	Change to lower-case
Ctrl + U	Change to upper-case
Insert	Overwrite
Ctrl + R	Search command history for match, stepping backward through commands
Ctrl + Shift + R	Search command history for match, stepping forward through commands

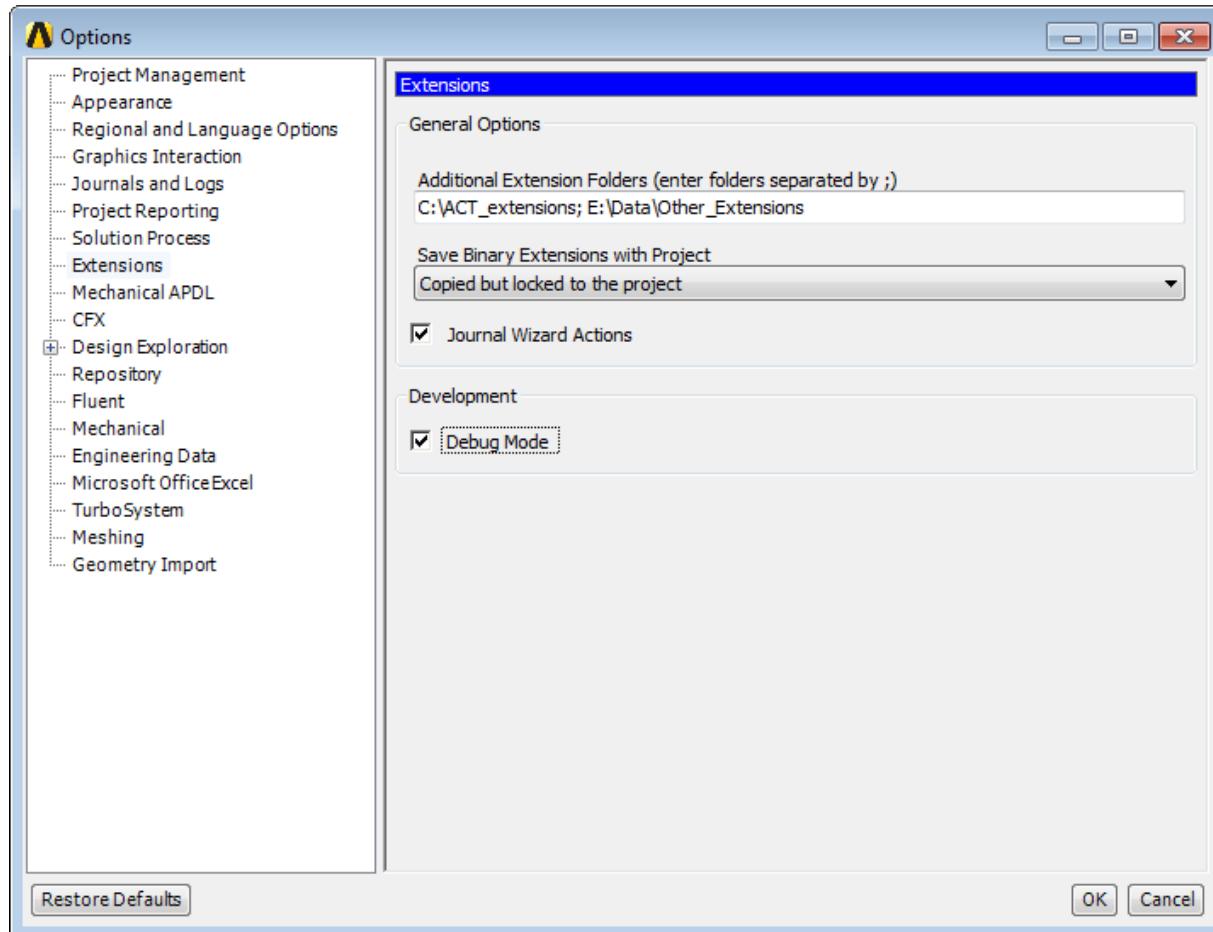
Binary Extension Builder

The binary extension builder is a tool for compiling a scripted extension into a binary extension that is then shared with end users. A separate compiler is not necessary. ACT provides its own process for encapsulating the scripted extension into a binary format.

The building of the binary extension encapsulates the scripted extension, generating a unique WBEX file that contains all the files or directories necessary for the extension. For access and usage information, see [Building a Binary Extension \(p. 18\)](#).

Debug Mode

When developing an extension, enabling the debug mode is recommended. To enable or disable this mode, you select **Tools > Options** from the main menu to open the **Options** dialog box. On the **Extensions** tab, you then select or clear the **Debug Mode** check box. For descriptions of other options on this tab, see [Configuring Extension Options \(p. 21\)](#).



When the debug mode is enabled, access points to debugging features are provided in Workbench, AIM, Mechanical, and DesignModeler:

Toolbar Buttons for Reloading Extensions

Extensions Log File

In Workbench, the debug mode is enabled as soon as you select the **Debug Mode** check box.

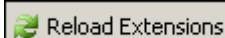
In Mechanical and DesignModeler, the debug mode is not enabled until the next time that the product is launched. If the product is already started when you select the **Debug Mode** check box, you must exit and restart it to make debugging functionality available.

Toolbar Buttons for Reloading Extensions

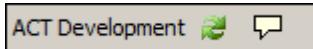
In the debug mode, you can use toolbar buttons to reload currently loaded extensions without having to exit the current product. This capability is particularly useful during the development of an extension because it provides an efficient method of interactively testing the changes in an XML extension definition file or IronPython script function. For major changes, you can ensure that an object acts as expected by deleting previously defined ACT objects and then recreating them.

You can reload extensions in any of the following ways:

- In the Workbench or AIM toolbar, click the **Reload Extensions** button. This button persists throughout all native Workbench or AIM tabs when debug mode is enabled.

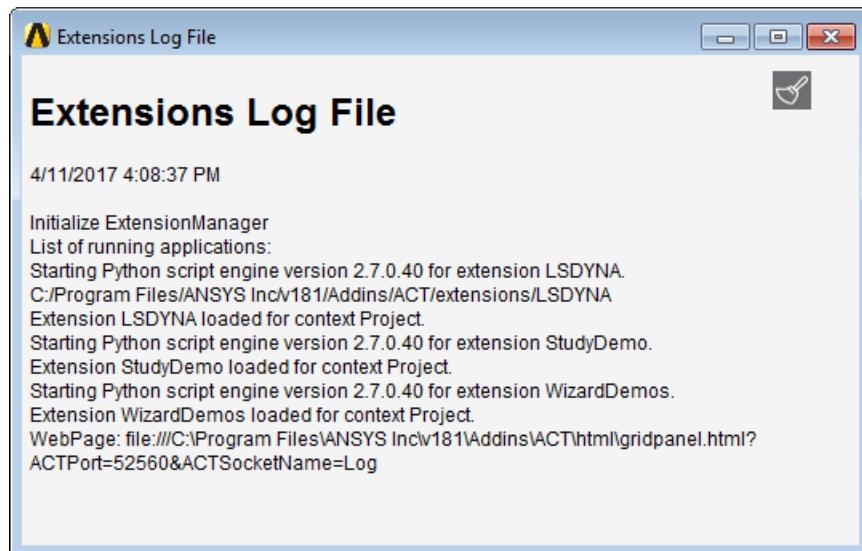


- In the **ACT Development** toolbar for Mechanical or DesignModeler, click the first button to reload extensions. Clicking the other button opens the [Extensions Log File \(p. 267\)](#).



Extensions Log File

The **Extensions Log File** displays the messages generated by extensions. If present, warnings are shown in orange text, and errors are shown in red text.

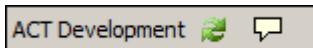


You can open the **Extensions Log File** in any of the following ways:

- From the **ACT Start Page**, click the corresponding toolbar button.



- From the main menu, select **Extensions > View Log File**.
- From the **ACT Development** toolbar for Mechanical or DesignModeler, click the second button. This toolbar is available only when debug mode is enabled.



- From the **ACT App Builder**, click the corresponding toolbar button.



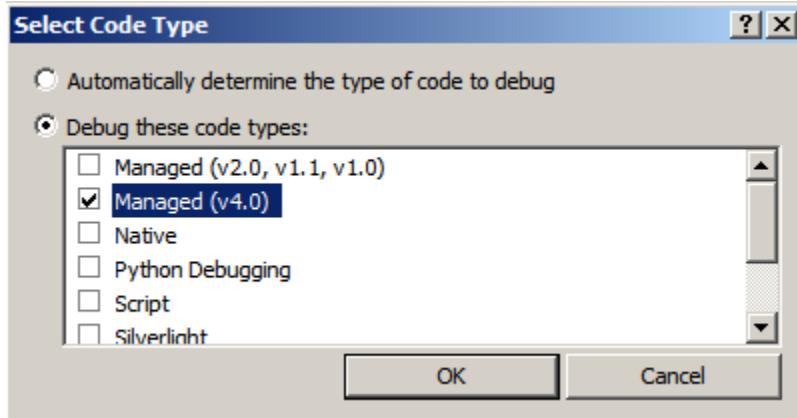
Once opened, the **Extensions Log File** window can be resized, dragged, and kept open as you switch between products. To clear this file, you click the brush icon in the upper right corner.

Debugging with Microsoft® Visual Studio

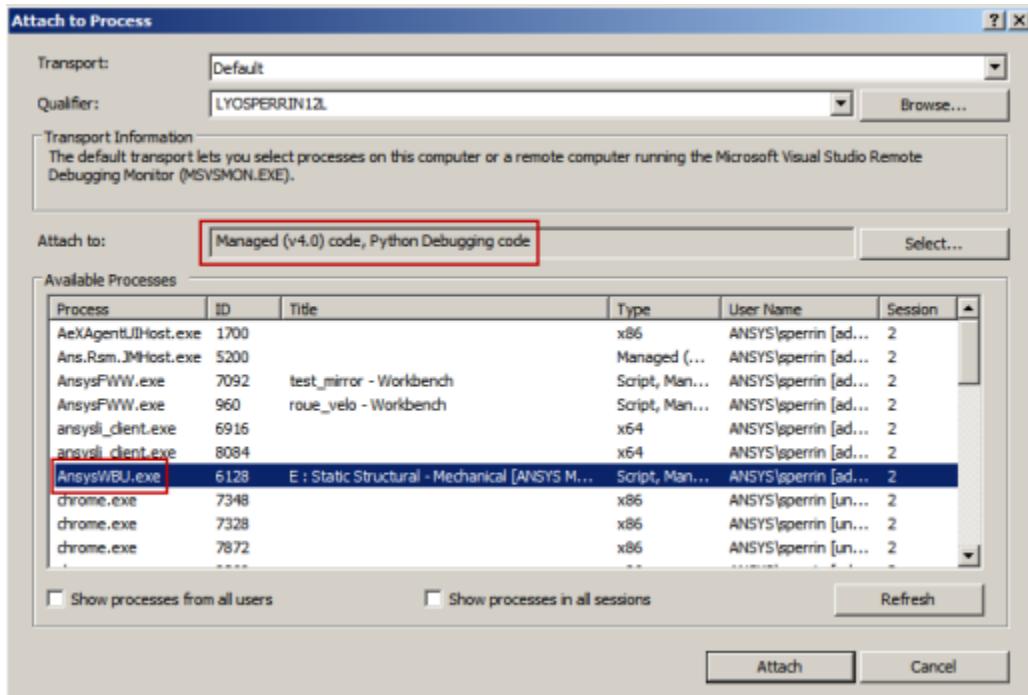
If you have Microsoft Visual Studio, you can use it to debug an IronPython script that is running.

For example, to debug an extension running in Mechanical:

1. Start Visual Studio.
2. For the **Attach to** field, verify that the **Managed (v.4.0)** code type is selected.



3. Attach the **AnsysWBU.exe** process.



4. After attaching the process, open your IronPython code and set some breakpoints into it.

Advanced Programming in C#

This section explains how to replace IronPython code with C# assemblies. C# is used in this document but any language that creates .NET assemblies can also be used for that purpose. This section assumes that you already know how to create an assembly in C#.

C# provides two major advantages over IronPython:

- Better performance
- Superior development environment that provides auto completion

Initialize the C# Project

Once you have created a C# project and associated the type "Class Library" with this project, add references to the **Ansys.ACT.Interfaces** and **Ansys.ACT.WB1** assemblies of ACT. Related DLLs for these assemblies are located at:

- <ANSYS_INSTALL_DIR>/Addins/ACT/bin/<Platform>/Ansys.ACT.Interfaces.dll.
- <ANSYS_INSTALL_DIR>/Addins/ACT/bin/<Platform>/Ansys.ACT.WB1.dll.

C# Implementation for a Load

The following XML extension definition file declares a load to be created in ANSYS Mechanical that requires C# implementation:

```
<extension version="1" name="CSharp">

  <author>ANSYS</author>
  <description>This extension demonstrates how to use CSharp to write extension.</description>

  <assembly src="CSharp.dll" namespace="CSharp" />

  <interface context="Mechanical">

    <images>images</images>

  </interface>

  <simdata context="Mechanical">

    <load name="CSharpLoad" caption="CSharp Load" version="1" icon="tload" unit="Temperature"
          color="#0000FF" class="CSharp.Load">
      <property name="Geometry" control="scoping">
        <attributes>
          <selection_filter>face</selection_filter>
        </attributes>
      </property>
    </load>

    <result name="CSharpResult" caption="CSharp Result" version="1" unit="Temperature" icon="tload"
           location="node" type="scalar" class="CSharp.Result">
      <property name="Geometry" control="scoping" />
    </result>
  </simdata>
</extension>
```

```
</simdata>  
</extension>
```

In the definition of the load object, the only change is the use of the attribute class. This attribute must be set to the name of the class to be used for the integration of the load.

A description of the class **CSharp.Load** follows.

```
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using Ansys.ACT.Interfaces.Mechanical;  
using Ansys.ACT.Interfaces.UserObject;  
namespace CSharp  
{  
    public class Load  
    {  
        private readonly IMechanicalExtAPI _api;  
        private readonly IMechanicalUserLoad _load;  
        public CustomLoadObjectController(IExtAPI api, IUserLoad load)  
        {  
            _api = (IMechanicalExtAPI) api;  
            _load = (IMechanicalUserLoad) load;  
        }  
  
        public virtual IEnumerable<double> getnodalvaluesfordisplay(IUserLoad load, IEnumerable<int> nodeIds)  
        {  
            var res = new List<double>();  
            IMeshData mesh = _load.Analysis.MeshData;  
            foreach (int nodeId in nodeIds)  
            {  
                INode node = mesh.NodeById(nodeId);  
                res.Add(Math.Sqrt(node.X * node.X + node.Y * node.Y + node.Z * node.Z));  
            }  
            return res;  
        }  
    }  
}
```

To implement a callback in C#, create a new method in your class with the name of the callback in lower case.

In the example, you implement the callback **<getnodalvaluesfordisplay>** by adding the method **getnodalvaluesfordisplay** to the class.

C# Implementation for a Result

The following XML file declares a result to be created in ANSYS Mechanical that requires C# implementation:

```
<extension version="1" name="CSharp">  
  
<author>ANSYS</author>  
<description>This extension demonstrates how to use CSharp to write extension.</description>  
  
<assembly src="CSharp.dll" namespace="CSharp" />  
  
<interface context="Mechanical">  
  
<images>images</images>  
  
</interface>  
  
<simdata context="Mechanical">
```

```

<load name="CSharpLoad" caption="CSharp Load" version="1" icon="tload" unit="Temperature"
      color="#0000FF" class="CSharp.Load">
  <property name="Geometry" control="scoping">
    <attributes>
      <selection_filter>face</selection_filter>
    </attributes>
  </property>
</load>

<result name="CSharpResult" caption="CSharp Result" version="1" unit="Temperature" icon="tload"
       location="node" type="scalar" class="CSharp.Result">
  <property name="Geometry" control="scoping" />
</result>

</simdata>

</extension>

```

For the load definition, the attribute class must be set to the name of the class to be used for the integration of the result.

A description of the class **CSharp.Result** follows.

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Ansys.ACT.Interfaces.Mechanical;
using Ansys.ACT.Interfaces.Post;
using Ansys.ACT.Interfaces.UserObject;

namespace CSharp
{
    public class Result
    {
        internal double[] res = new double[1];

        public Result(IMechanicalExtAPI extApi, IUserResult result)
        {
        }

        public void evaluate(IUserResult entity, IStepInfo stepInfo, IResultCollector collector)
        {
            foreach (var id in collector.Ids)
            {
                res[0] = id;
                collector.SetValues(id, res);
            }
        }
    }
}

```

As for the load definition, the implementation of a new callback simply requires you add a new method with the name of the callback.

Extension Examples

ACT supports product customization by exposing a set of interfaces for each supported ANSYS product. The examples in this section build upon the methods and techniques discussed in previous sections.

Examples are provided for the following areas:

[Mechanical Extension Examples](#)

[DesignXplorer Extension Examples](#)

[Custom ACT Workflows in Workbench Examples](#)

[Wizard Examples](#)

Note

Most of the examples are included in the package **ACT Developer's Guide Examples**, which you can download from the [ACT Resources](#) page on the [ANSYS Customer Portal](#).

- To display the page, select **Downloads > ACT Resources**.
- To download the package, expand the **Help & Support** section and click **ACT Developer's Guide Examples** under **Documentation**.
- To view extension installation information, in the same **Help & Support** section, click **ACT Extension Installation Procedure** under **Extension Resources**. Or, see [Installing and Uninstalling Extensions](#) (p. 32).

A few of the examples are included in template packages, which you can also download from the [ACT Resources](#) page. After expanding the **ACT Templates** section, click the template package that you want to download.

Mechanical Extension Examples

The following examples are provided for ANSYS Mechanical:

[Von-Mises Stress as a Custom Result](#)

[Edge-Node Coupling Tool](#)

Note

The examples in this section are included in the package **ACT Developer's Guide Examples**, which you can download from the [ACT Resources](#) page on the [ANSYS Customer Portal](#). For more information, see [Extension Examples](#) (p. 273).

Von-Mises Stress as a Custom Result

The extension Mises reproduces the result for the averaged Von Mises equivalent stress. While this result is already available in the standard results that can be selected in Mechanical, this example demonstrates how you can define a custom result.

First, consider the support required in the XML definition file for the custom result. The `<interface>` block contains information for adding a toolbar and a toolbar button to the user interface. The callback function `<Create_Mises_Result>` is invoked when the toolbar button is clicked. Next the `<sim-data>` block encapsulates the information needed for the custom result. One `result` block is declared. The callbacks `onstarteval` and `getvalue` invoke functions for computing and storing the custom result (Von Mises Stress). They are invoked when custom result values are queried for graphical display. The property `Geometry` defines one property to be added in the **Details** pane of the custom result. For this example, the property integrates a scoping method in the custom result.

```
<extension version="1" name="Mises">

<guid shortid="Mises">a1844c3c-b65c-444c-a5ad-13215a9f0413</guid>

<script src="main.py" />

<interface context="Mechanical">

<images>images</images>

<toolbar name="Von Mises Stress" caption="Von Mises Stress">
  <entry name="Von Mises Stress" icon="result">
    <callbacks>
      <onclick>Create_Mises_Result</onclick>
    </callbacks>
  </entry>
</toolbar>

</interface>

<simdata context="Mechanical">

  <result name="Von Mises Stress" version="1" caption="Von Mises Stress" unit="Stress" icon="result"
location="node" type="scalar">

    <callbacks>
      <onstarteval>Mises_At_Nodes_Eval</onstarteval>
      <getvalue>Mises_At_Nodes_GetValue</getvalue>
    </callbacks>

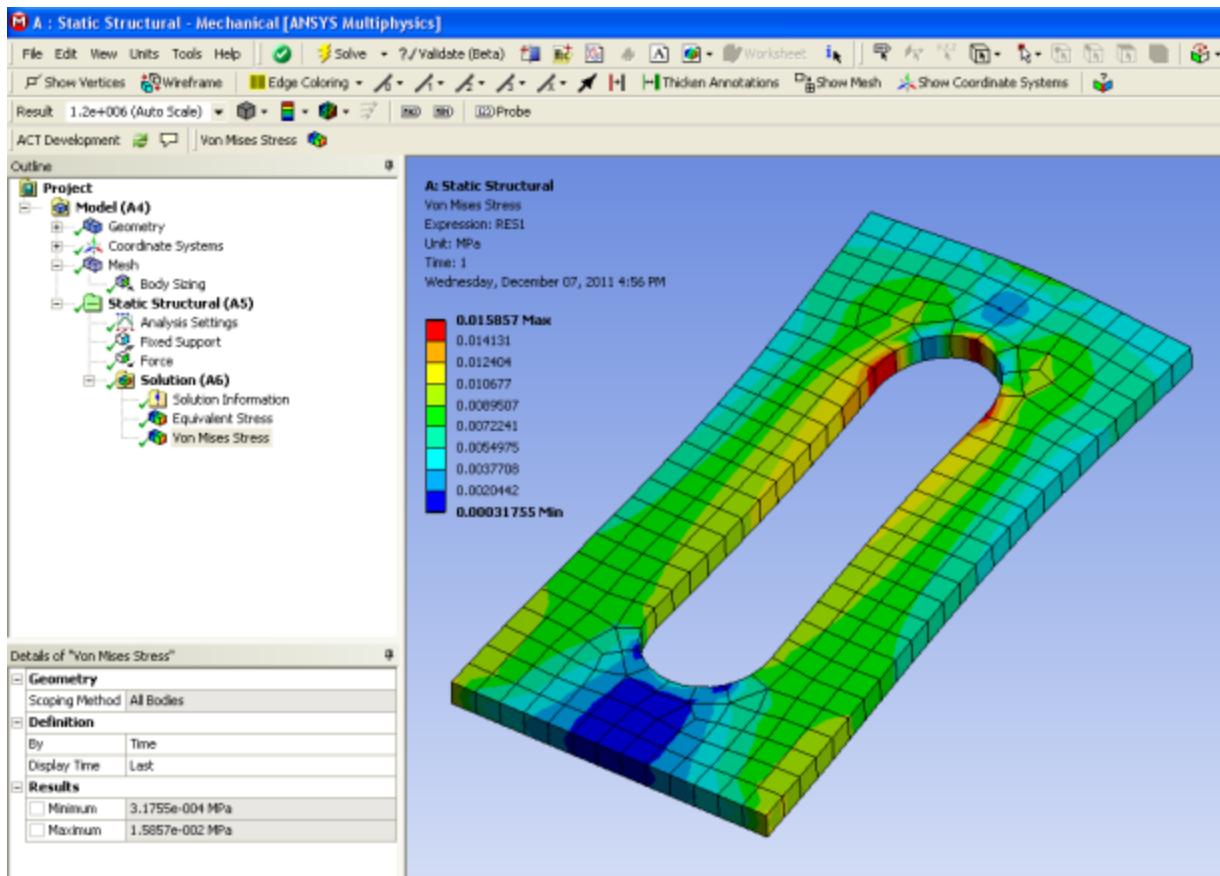
    <property name="Geometry" caption="Geometry" control="scoping"></property>

  </result>

</simdata>

</extension>
```

The following figure shows how the toolbar and result are added to the Mechanical user interface.



The IronPython script for this extension follows the next several descriptive paragraphs. This script defines a callback function named `<Create_Mises_Result>`. When activated by the **Von Mises Stress** toolbar button, the callback creates the result **Von Mises Stress**. The callback invokes the function `Create_Mises_Result` on the interface `simDataMgr`. The file `Mises.xml` provides the details needed to create both the **Von Mises Stress** toolbar button and the custom result.

In addition, the script `main.py` contains all the functions used by this extension.

At the top of the script is the list definition for the node adjacency tables. The list, named `link`, contains hash maps. Each hash map has a numeric hash key that corresponds to the type returned by the property `IElementType`. The values for the hash map are in arrays. The names of the arrays match the local node numbers for a typical element of the key (element-type). Finally, the content of each array provides the adjacent local node numbers for the corresponding elements node. This list of arrays makes the extension compatible with all the different element topologies that Mechanical potentially creates during the mesh generation.

The XML file for this extension specifies one other callback in the script:

```
<onstarteval>Mises_At_Nodes_Eval</onstarteval>
```

The callback `<onstarteval>` is associated with the function `Mises_At_Nodes_Eval`. It is invoked as the result of an event thrown by Mechanical. This function computes the appropriate stress results and stores them. The comments in the following script describe the algorithm used. It returns the nodal results set stored for the specified node ID. This callback is used by the graphics system to display the results.

The script also demonstrates the use of utility sub-functions. Using sub-functions to perform common repetitive task helps to make the script modular and more understandable. The utility sub-functions

used in here are **Mises** and **EigenValues**. **Mises** is called by the function **Mises_At_Nodes_Eval** and returns the computed equivalent stress for a given stress tensor (SX,SY,SZ,SXY,SXZ,SYZ). **EigenValues** is called by the function **Mises** to compute the Eigen vectors for a given stress tensor.

```
import units
import math

context = ExtAPI.Context
if context == "Mechanical":
    link = {}
    #kHex20
    link.Add(ElementTypeEnum.kHex20,{ 0:[3,1,4], 1:[0,2,5], 2:[1,3,6], 3:[2,0,7], 4:[5,0,7], 5:[1,4,6],
6:[5,7,2], 7:[6,3,4], 8:[0,1], 9:[1,2], 10:[2,3], 11:[3,0], 12:[4,5], 13:[5,6], 14:[6,7], 15:[7,4],
16:[0,4], 17:[1,5], 18:[2,6], 19:[3,7] })
    #kHex8
    link.Add(ElementTypeEnum.kHex8,{ 0:[3,1,4], 1:[0,2,5], 2:[1,3,6], 3:[2,0,7], 4:[5,0,7], 5:[1,4,6],
6:[5,7,2], 7:[6,3,4] })
    #kPyramid13
    link.Add(ElementTypeEnum.kPyramid13,{ 0:[3,1,4], 1:[0,2,4], 2:[1,3,4], 3:[2,0,4], 4:[0,1,2,3],
5:[0,1], 6:[1,2], 7:[2,3], 8:[3,0], 9:[0,4], 10:[1,4], 11:[2,4], 12:[3,4] })
    #kPyramid5
    link.Add(ElementTypeEnum.kPyramid5,{ 0:[3,1,4], 1:[0,2,4], 2:[1,3,4], 3:[2,0,4], 4:[0,1,2,3] })
    #kQuad4
    link.Add(ElementTypeEnum.kQuad4, { 0:[3,1], 1:[0,2], 2:[1,3], 3:[2,0] })
    #kQuad8
    link.Add(ElementTypeEnum.kQuad8, { 0:[3,1], 1:[0,2], 2:[1,3], 3:[2,0], 4:[0,1], 5:[1,2], 6:[2,3],
7:[3,0] })
    #kTet10
    link.Add(ElementTypeEnum.kTet10,{ 0:[2,1,3], 1:[0,2,3], 2:[1,0,3], 3:[0,1,2], 4:[0,1], 5:[1,2],
6:[2,0], 7:[0,3], 8:[1,3], 9:[2,3] })
    #kTet4
    link.Add(ElementTypeEnum.kTet4, { 0:[2,1,3], 1:[0,2,3], 2:[1,0,3], 3:[0,1,2] })
    #kTri3
    link.Add(ElementTypeEnum.kTri3, { 0:[2,1], 1:[0,2], 2:[1,0] })
    #kTri6
    link.Add(ElementTypeEnum.kTri6, { 0:[2,1], 1:[0,2], 2:[1,0], 3:[0,1], 4:[1,2], 5:[2,0] })
    #kWedge15
    link.Add(ElementTypeEnum.kWedge15,{ 0:[2,1,3], 1:[0,2,4], 2:[1,0,5], 3:[5,4,0], 4:[3,5,1], 5:[4,3,2],
6:[0,1], 7:[1,2], 8:[2,0], 9:[3,4], 10:[4,5], 11:[5,3], 12:[0,3], 13:[1,4], 14:[2,5] })
    #kWedge6
    link.Add(ElementTypeEnum.kWedge6,{ 0:[2,1,3], 1:[0,2,4], 2:[1,0,5], 3:[5,4,0], 4:[3,5,1], 5:[4,3,2] })

def Create_Mises_Result(analysis):
    analysis.CreateResultObject("Von Mises Stress")

mises_stress = {}

# This function evaluates the specific result (i.e. the Von-Mises stress) on each element required by
the geometry selection
# The input data "step" represents the step on which we have to evaluate the result
def Mises_At_Nodes_Eval(result, step):
    global mises_stress

    ExtAPI.Log.WriteMessage("Launch evaluation of the Mises result at nodes: ")
    # Reader initialization
    reader = result.Analysis.GetResultsData()
    istep = int(step)
    reader.CurrentResultSet = istep
    # Get the stress result from the reader
    stress = reader.GetResult("S")
    stress.SelectComponents(["X", "Y", "Z", "XY", "XZ", "YZ"])
    unit_stress = stress.GetComponentInfo("X").Unit
    conv_stress = units.ConvertUnit(1.,unit_stress,"Pa","Stress")

    # Get the selected geometry
    propGeo = result.Properties["Geometry"]
    refIds = propGeo.Value.Ids
```

```

nodal_stress = [0.] * 6

# Get the mesh of the model
mesh = result.Analysis.MeshData

# Loop on the list of the selected geometrical entities
for refId in refIds:
    # Get mesh information for each geometrical entity
    meshRegion = mesh.MeshRegionById(refId)
    nodeIds = meshRegion.NodeIds

    # Loop on the nodes related to the current geometrical refId
    for nodeId in nodeIds:

        for i in range(6):
            nodal_stress[i] = 0.

        elementIds = mesh.NodeById(nodeId).ConnectedElementIds
        num = 0
        # Loop on the elements related to the current node
        for elementId in elementIds:
            # Get the stress tensor related to the current element
            tensor = stress.GetElementValues(elementId)

            element = mesh.ElementById(elementId)
            # Look for the position of the node nodeId in the element elementId
            # cpt contains this position
            cpt = element.NodeIds.IndexOf(nodeId)

            # for corner nodes, cpt is useless.
            # The n corner nodes of one element are always the first n nodes of the list
            if cpt < element.CornerNodeIds.Count:
                for i in range(6):
                    nodal_stress[i] = nodal_stress[i] + tensor[6*cpt+i]
            else:
                # For midside nodes, cpt is used and the link table provides the two neighbouring corner
                # nodes for the midside node identified in the list by cpt.
                itoadd = link[element.Type][cpt]
                for ii in itoadd:
                    for i in range(6):
                        nodal_stress[i] = nodal_stress[i] + tensor[6*ii+i] / 2.

            num += 1

        # The average stress tensor is computed before to compute the Von-Mises stress
        # num is the number of elements connected with the current node nodeId
        for i in range(6):
            nodal_stress[i] *= conv_stress / num

        # Von-Mises stress computation
        vm_stress = Mises(nodal_stress)
        # Result storage
        mises_stress[nodeId] = [vm_stress]

# This function returns the values array. This array will be used by Mechanical to make the display available
def Mises_At_Nodes_GetValue(result,nodeId):
    global mises_stress
    try:    return mises_stress[nodeId]
    except: return [System.Double.MaxValue]

# This function computes the Von-Mises stress from the stress tensor
# The Von-Mises stess is computed based on the three eigenvalues of the stress tensor
def Mises(tensor):

    # Computation of the eigenvalues
    (S1, S2, S3) = EigenValues(tensor)
    return sqrt( ( (S1-S2)*(S1-S2) + (S2-S3)*(S2-S3) + (S1-S3)*(S1-S3) ) / 2. )

# This function computes the three eigenvalues of one [3*3] symetric tensor

```

```
EPSILON = 1e-4
def EigenValues(tensor):

    a = tensor[0]
    b = tensor[1]
    c = tensor[2]
    d = tensor[3]
    e = tensor[4]
    f = tensor[5]

    if ((abs(d)>EPSILON) or (abs(e)>EPSILON) or (abs(f)>EPSILON)):
        # Polynomial reduction
        A = -(a+b+c)
        B = a*b+a*c+b*c-d*e-f*f
        C = d*d*c+f*f*a+e*e*b-2*d*e*f-a*b*c

        p = B-A*A/3
        q = C-A*B/3+2*A*A*A/27
        R = sqrt(fabs(p)/3)
        if q < 0: R = -R

        z = q/(2*R*R*R)
        if z < -1.: z = -1.
        elif z > 1.: z = 1.
        phi = acos(z)

        S1 = -2*R*cos(phi/3)-A/3
        S2 = -2*R*cos(phi/3+2*math.pi/3)-A/3
        S3 = -2*R*cos(phi/3+4*math.pi/3)-A/3
    else:
        S1 = a
        S2 = b
        S3 = c

    return (S1, S2, S3)
```

Edge-Node Coupling Tool

The extension **Coupling** creates a tool that can be used to couple two set of nodes related to two edges. This example demonstrates how you can develop your own preprocessing feature (such as a custom load) to address one specific need.

The following XML extension definition file specifies the custom load. As in the previous example, the **<interface>** block contains information for adding a toolbar and a toolbar button to the user interface. The callback function **<CreateCoupling>** is invoked when the toolbar button is clicked. Next, the **<simdata>** block is defined. It encapsulates the information that defines the support. Of particular importance is that the value for the attribute **issupport** is set to **true**. The attribute **issupport** tells Mechanical which type of boundary condition to apply. Three result level callback functions are declared. The function **SolveCmd** is registered and called for an event that gets fired when the solver input is being written. The functions **ShowCoupling** and **HideCoupling** are registered and called for events used to synchronize tree view selections with content in the graphics pane. The details needed to define the inputs and behavior of this special load consist of three properties, **Source**, **Target**, and **Reverse**, along with their behavioral callbacks.

```
<extension version="1" name="Coupling">
<guid shortid="Coupling">e0d5c579d-0263-472a-ae0e-b3cbb9b74b6c</guid>
<script src="main.py" />
<interface context="Mechanical">
    <images>images</images>
    <toolbar name="Coupling" caption="Coupling">
        <entry name="Coupling" icon="support">

            <callbacks>
                <onclick>CreateCoupling</onclick>
            </callbacks>
        </entry>
    </toolbar>
</interface>
<simdata>
    <coupling>
        <source></source>
        <target></target>
        <reverse></reverse>
    </coupling>
</simdata>
<callback>
    <event>SolveCmd</event>
    <function>SolveCmd</function>
</callback>
<callback>
    <event>ShowCoupling</event>
    <function>ShowCoupling</function>
</callback>
<callback>
    <event>HideCoupling</event>
    <function>HideCoupling</function>
</callback>
</extension>
```

```

</toolbar>
</interface>

<simdata context="Mechanical">

<load name="Coupling" version="1" caption="Coupling" icon="support" issupport="true"
      color="#FF0000">
<callbacks>
<getsolvecommands>SolveCmd</getsolvecommands>
<onshow>ShowCoupling</onshow>
<onhide>HideCoupling</onhide>
</callbacks>

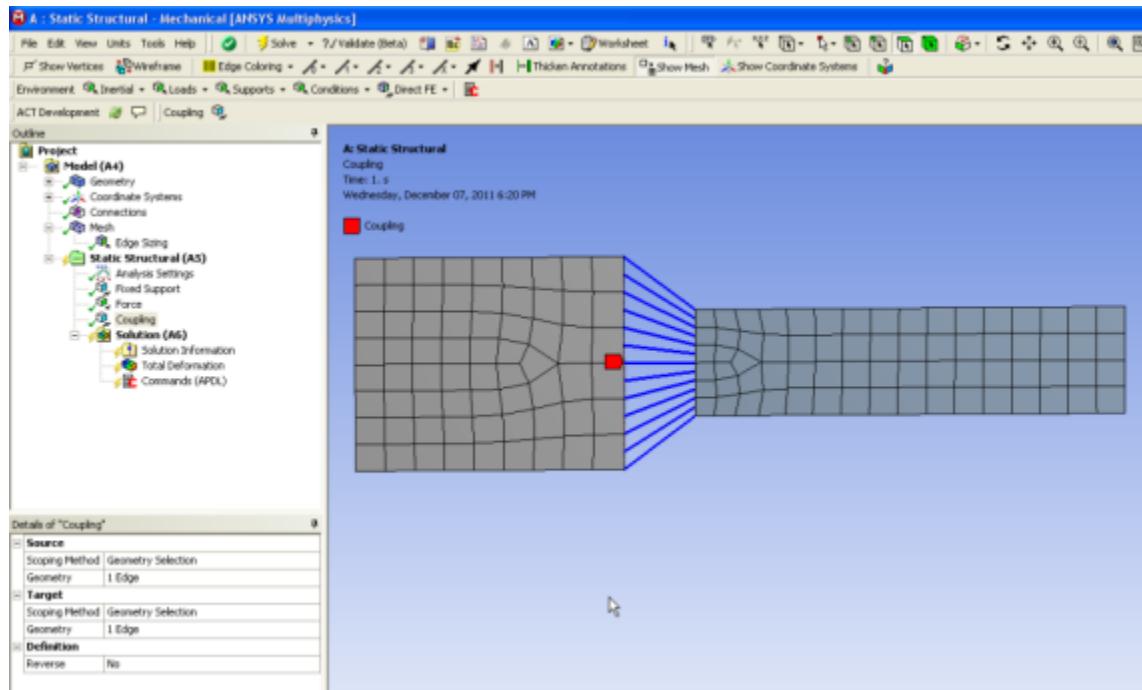
<property name="Source" caption="Source" control="scoping">
<attributes selection_filter="edge" />
<callbacks>
<isvalid>IsValidCoupledScoping</isvalid>
<onvalidate>OnValidateScoping</onvalidate>
</callbacks>
</property>

<property name="Target" caption="Target" control="scoping">
<attributes selection_filter="edge" />
<callbacks>
<isvalid>IsValidCoupledScoping</isvalid>
<onvalidate>OnValidateScoping</onvalidate>
</callbacks>
</property>

<property name="Reverse" caption="Reverse" control="select" default="No">
<attributes options="No, Yes" />
<callbacks>
<onvalidate>OnValidateReverse</onvalidate>
</callbacks>
</property>
</load>
</simdata>
</extension>

```

The following figure shows how a fully defined coupling appears in Mechanical.



The following IronPython script for the extension **Coupling** defines a callback function named **<CreateCoupling>**. When activated by the **Coupling** toolbar button, this callback creates the load **Coupling**. The callback invokes the function **CreateLoadObject** for the current analysis. The function **SolveCmd** is invoked when the solver input is being generated. **SolveCmd** invokes **GetListNodes** to obtain two lists of node IDs corresponding to the edges **Target** and **Source**. These node IDs are then used to write APDL CP commands to the solver input. **GetListNodes** is also invoked by the callback function **<ShowCoupling>**. In **<ShowCoupling>**, the interface **IGraphics** is used to create a graphics context. Using the object returned, the inter-nodal lines are drawn to provide a visual representation of the coupling.

The graphics context associated with this custom load and the validation of the user inputs impose to manage more various situations than for the first example described in the previous section. This explains why more functions and sub-functions are required for this example.

```
import graphics

def CreateCoupling(analysis):
    analysis.CreateLoadObject("Coupling")

#-----
#   Callbacks
#-----

def OnValidateReverse(load, prop):
    ShowCoupling(load)

def OnValidateScoping(load, prop):
    ShowCoupling(load)

def IsValidScoping(load, prop):
    if not prop.Controller.isvalid(load, prop):
        return False

    selection = prop.Value
    if selection == None: return False
    if selection.Ids.Count != 1: return False
    return True

def IsValidCoupledScoping(load, prop):
    sProp = load.Properties["Source"]
    tProp = load.Properties["Target"]

    if not IsValidScoping(load, sProp):
        return False
    if not IsValidScoping(load, tProp):
        return False

    sIds = sProp.Value.Ids
    tIds = tProp.Value.Ids

    try:
        mesh = load.Analysis.MeshData
        sNum = mesh.MeshRegionById(sIds[0]).NodeCount
        tNum = mesh.MeshRegionById(tIds[0]).NodeCount
        if sNum == 0 or tNum == 0: return False
    except:
        return False

    return sNum == tNum

#-----
#   Show / Hide
#-----

graphicsContext = {}
```

```

def getContext(entity):
    global graphicsContext
    if entity.Id in graphicsContext : return graphicsContext[entity.Id]
    else : return None

def setContext(entity, context):
    global graphicsContext
    graphicsContext[entity.Id] = context

def delContext(entity):
    context = getContext(entity)
    if context != None : context.Visible = False
    context = None
    setContext(entity, None)

def ShowCoupling(load):
    delContext(load)
    ctxCoupling = ExtAPI.Graphics.CreateAndOpenDraw3DContext()

    sourceColor = load.Color
    targetColor = 0x00FF00
    lineColor = 0x0000FF

    sProp = load.Properties["Source"] ; sSel = sProp.Value
    tProp = load.Properties["Target"] ; tSel = tProp.Value

    ctxCoupling.LineWeight = 1.5
    if sSel != None:
        ctxCoupling.Color = sourceColor
        for id in sSel.Ids:
            graphics.DrawGeoEntity(ExtAPI, load.Analysis.GeoData, id, ctxCoupling)
    if tSel != None:
        ctxCoupling.Color = targetColor
        for id in tSel.Ids:
            graphics.DrawGeoEntity(ExtAPI, load.Analysis.GeoData, id, ctxCoupling)

    if IsValidSelections(load):

        ctxCoupling.Color = lineColor
        ctxCoupling.LineWeight = 1.5

        mesh = load.Analysis.MeshData
        sList, tList = GetListNodes(load)

        for sId, tId in zip(sList, tList):
            sNode = mesh.NodeById(sId)
            tNode = mesh.NodeById(tId)
            ctxCoupling.DrawLine([sNode.X,sNode.Y,sNode.Z,tNode.X,tNode.Y,tNode.Z])

    ctxCoupling.Close()
    ctxCoupling.Visible = True
    setContext(load, ctxCoupling)

def HideCoupling(load):
    delContext(load)

#-----
# Commands
#-----

def SolveCmd(load, s):
    s.WriteLine("! Coupling - CP")
    sList, tList = GetListNodes(load)
    for sId, tId in zip(sList, tList):
        s.WriteLine("CP,NEXT,ALL,{0},{1}", sId, tId)

#-----
# Utils
#-----

def IsValidSelections(load):
    return load.Properties["Source"].IsValid and load.Properties["Target"].IsValid

```

```

def GetListNodes(load):

    if IsValidSelections(load):

        sProp = load.Properties["Source"] ; sIds = sProp.Value.Ids
        tProp = load.Properties["Target"] ; tIds = tProp.Value.Ids

        geometry = ExtAPI.DataModel.GeoData
        mesh = load.Analysis.MeshData

        sList = GetSubListNodes(geometry, mesh, sIds[0])
        tList = GetSubListNodes(geometry, mesh, tIds[0])

        rev = False
        r = load.Properties["Reverse"].Value
        if r == "Yes": rev = True

        sList = sorted(sList, key=sList.get)
        tList = sorted(tList, key=tList.get, reverse=rev)

    return (sList, tList)

def GetSubListNodes(geometry, mesh, refId):

    entity = geometry.GeoEntityById(refId)
    region = mesh.MeshRegionById(refId)

    result = {}
    pt = System.Array.CreateInstance(System.Double, 3)

    for nodeId in region.NodeIds:
        node = mesh.NodeById(nodeId)
        pt[0], pt[1], pt[2] = (node.X, node.Y, node.Z)
        result[nodeId] = entity.ParamAtPoint(pt)

    return result

```

DesignXplorer Extension Examples

For ANSYS DesignXplorer, the following extension examples are provided:

[DOE Extension Examples](#)

[Optimization Extension Examples](#)

These examples are included in the package **ACT Templates for DesignXplorer**. You can download this template package from the [ACT Resources](#) page on the [ANSYS Customer Portal](#). For more information, see [Extension Examples \(p. 273\)](#).

DOE Extension Examples

In the package **ACT Templates for DesignXplorer**, two extension examples show how to generate design point samples. Each of these extensions uses a different programming language, allowing you to see how you can accomplish this objective using IronPython or C#.

Example Name	Description
PythonSampling	<p>Fully implemented in IronPython (no build required), this example is your sandbox. It demonstrates many sampling extension features and is definitely the best example to start with.</p> <p>PythonSampling is running a simple random exploration of the parametric space, generating the number of points requested by the user.</p>

CSharpSampling	<p>This example demonstrates how an extension can be completely set up when the interface ISamplingMethod is implemented in C#.</p> <p>Note that the optimization algorithm is also implemented in C#, but one could use only the ISamplingMethod implementation as an adapter to an existing implementation in C# or other languages.</p> <p>The project is provided for Microsoft Visual Studio 2010.</p>
----------------	---

Optimization Extension Examples

In the package **ACT Templates for DesignXplorer**, three extension examples show how to optimize the design space. Each of these extensions uses a different programming language, allowing you to see how you can accomplish this objective using IronPython, C#, or C/C++.

Example Name	Description
PythonOptimizer	<p>Fully implemented in IronPython (no build required), this example is your sandbox. It demonstrates many optimization extension features and is definitely the best example to start with.</p> <p>PythonOptimizer is running a simple random exploration of the parametric space, generating the number of points requested by the user and returning the best candidates found.</p>
CSharpOptimizer	<p>This example demonstrates how an extension can be completely set up when the interface IOptimizationMethod is implemented in C#.</p> <p>Note that the optimization algorithm is also implemented in C#, but one could use only the IOptimizationMethod implementation as an adapter to an existing implementation in C# or other languages.</p> <p>The project is provided for Microsoft Visual Studio 2010.</p>
CppOptimizer	<p>This example demonstrates how an extension can be implemented from existing C/C++. The interface IOptimizationMethod is implemented in IronPython as a wrapper to the C++ symbols, using the ctypes foreign function library for Python.</p>

Custom ACT Workflows in Workbench Examples

The following examples are provided for custom workflows in the Workbench **Project** tab:

[Global Workflow Callbacks](#)

[Custom User-Specified GUI Operation](#)

[Custom, Lightweight, External Application Integration with Parameter Definition](#)

[Custom, Lightweight, External Application Integration with Custom Data and Remote Job Execution](#)

[Generic Material Transfer](#)

[Generic Mesh Transfer](#)

[Custom Transfer](#)

Parametric

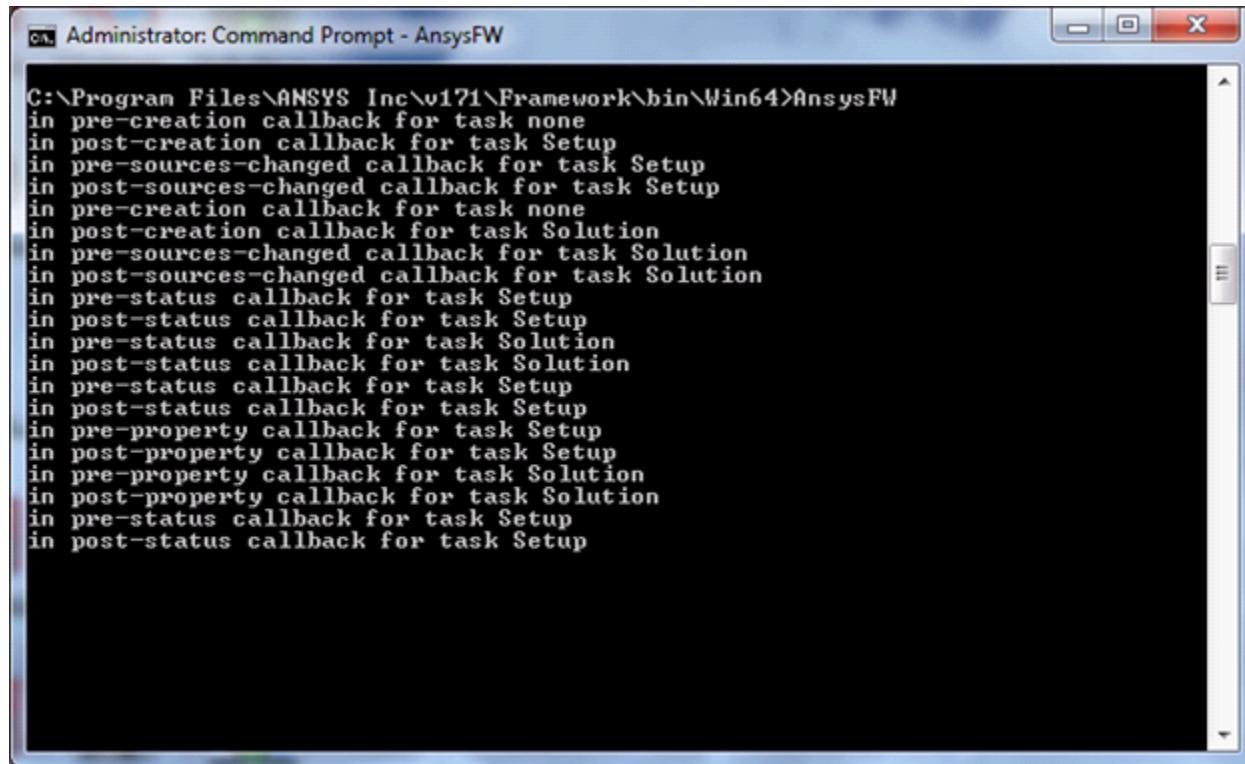
Note

All of these examples are included in the package **ACT Developer's Guide Examples**, which you can download from the [ACT Resources](#) page on the [ANSYS Customer Portal](#). For more information, see [Extension Examples \(p. 273\)](#).

Global Workflow Callbacks

The extension **WorkflowCallbacksDemo** demonstrates the use of global callbacks to implement a custom process or action before or after a Workbench **Project Schematic** operation. Specifically, it defines pre- and post-operation callbacks for each available operation. Each task, or argument passed to the invoked method, defines an action: writing a message in the command line window either before or after execution of the operation.

This example also shows how to specify that a callback method (in this case, `onAfterUpdate`) is processed only for a specific task or template.



```
C:\>Administrator: Command Prompt - AnsysFW
C:\Program Files\ANSYS Inc\v171\Framework\bin\Win64>AnsysFW
in pre-creation callback for task none
in post-creation callback for task Setup
in pre-sources-changed callback for task Setup
in post-sources-changed callback for task Setup
in pre-creation callback for task none
in post-creation callback for task Solution
in pre-sources-changed callback for task Solution
in post-sources-changed callback for task Solution
in pre-status callback for task Setup
in post-status callback for task Setup
in pre-status callback for task Solution
in post-status callback for task Solution
in pre-status callback for task Setup
in post-status callback for task Setup
in pre-property callback for task Setup
in post-property callback for task Setup
in pre-property callback for task Solution
in post-property callback for task Solution
in pre-status callback for task Setup
in post-status callback for task Setup
```

XML Extension Definition File

The XML extension definition file `WorkflowCallbacksDemo.xml` performs the following actions:

- References the IronPython script `main.py`.
- For the `<interface>` block, specifies that the attribute `context` is set to `Project`, which means that the extension is executed on the Project tab.
- For the `<workflow>` block:

- Defines the attributes **name** and **context**.
- Defines pre- and post-operation global callbacks for each available **Project Schematic** operation.

```
<extension version="1" name="WorkflowCallbacksDemo">
<guid shortid="WorkflowCallbacksDemo">96d0195b-e138-4841-a13a-de12238c83f2</guid>
<script src="main.py" />
<interface context="Project">
<images>images</images>
</interface>
<workflow name="WorkflowDemol" context="Project" version="1">
<callbacks>
<onbeforetaskreset>onBeforeReset</onbeforetaskreset>
<onaftertaskreset>onAfterReset</onaftertaskreset>
<onbeforetaskrefresh>onBeforeRefresh</onbeforetaskrefresh>
<onaftertaskrefresh>onAfterRefresh</onaftertaskrefresh>
<onbeforetaskupdate>onBeforeUpdate</onbeforetaskupdate>
<onaftertaskupdate>onAfterUpdate</onaftertaskupdate>
<onbeforetaskduplicate>onBeforeDuplicate</onbeforetaskduplicate>
<onaftertaskduplicate>onAfterDuplicate</onaftertaskduplicate>
<onbeforetasksourceschanged>onBeforeSourcesChanged</onbeforetasksourceschanged>
<onaftertasksourceschanged>onAfterSourcesChanged</onaftertasksourceschanged>
<onbeforetaskcreation>onBeforeCreate</onbeforetaskcreation>
<onaftertaskcreation>onAfterCreate</onaftertaskcreation>
<onbeforetaskdeletion>onBeforeDelete</onbeforetaskdeletion>
<onaftertaskdeletion>onAfterDelete</onaftertaskdeletion>
<onbeforetaskcanusetransfer>onBeforeCanUseTransfer</onbeforetaskcanusetransfer>
<onaftertaskcanusetransfer>onAfterCanUseTransfer</onaftertaskcanusetransfer>
<onbeforetaskcanduplicate>onBeforeCanDuplicate</onbeforetaskcanduplicate>
<onaftertaskcanduplicate>onAfterCanDuplicate</onaftertaskcanduplicate>
<onbeforetaskstatus>onBeforeStatus</onbeforetaskstatus>
<onaftertaskstatus>onAfterStatus</onaftertaskstatus>
<onbeforetaskpropertyretrieval>onBeforePropertyRetrieval</onbeforetaskpropertyretrieval>
<onaftertaskpropertyretrieval>onAfterPropertyRetrieval</onaftertaskpropertyretrieval>
</callbacks>
</workflow>
</extension>
```

IronPython Script

The IronPython script `main.py` contains the code defining the methods invoked by the global callbacks. This script performs the following actions:

- Defines 23 methods:
 - One method for each of the 22 global callback defined in the XML extension definition file.
 - One method to format a message to write to the command line window.
- Specifies (via the method arguments) that the methods **onBeforeUpdate** and **onAfterUpdate** should be executed only for updates of **Engineering Data** tasks.

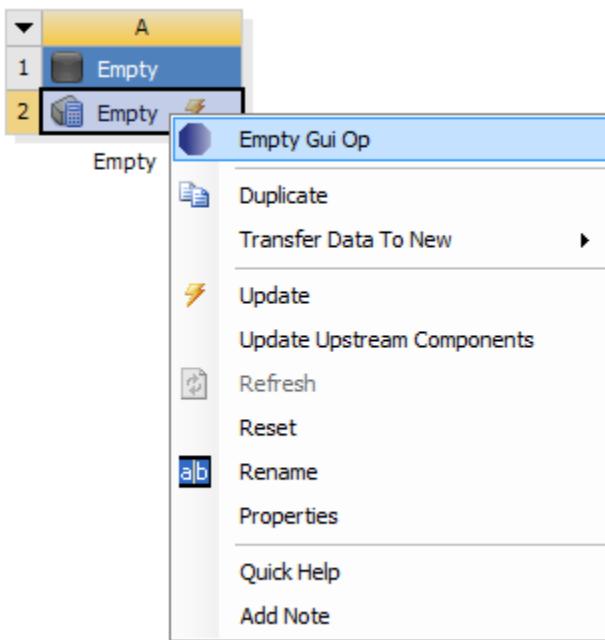
The script `main.py` contains the following code:

```
def onBeforeReset(task):
    msg = getPrintMessage('pre-reset', task)
    print msg
def onAfterReset(task):
    msg = getPrintMessage('post-reset', task)
    print msg
def onBeforeRefresh(task):
    msg = getPrintMessage('pre-refresh', task)
    print msg
def onAfterRefresh(task):
    msg = getPrintMessage('post-refresh', task)
    print msg
def onBeforeUpdate(task):
```

```
if task.Name == "Engineering Data":
    msg = getPrintMessage('post-update', task)
    print msg
else:
    print("ignored")
def onAfterUpdate(task):
    if task.Name == "Engineering Data":
        msg = getPrintMessage('post-update', task)
        print msg
    else:
        print("ignored")
def onBeforeDuplicate(task):
    msg = getPrintMessage('pre-duplicate', task)
    print msg
def onAfterDuplicate(task):
    msg = getPrintMessage('post-duplicate', task)
    print msg
def onBeforeSourcesChanged(task):
    msg = getPrintMessage('pre-sources-changed', task)
    print msg
def onAfterSourcesChanged(task):
    msg = getPrintMessage('post-sources-changed', task)
    print msg
def onBeforeCreate(task):
    msg = getPrintMessage('pre-creation', task)
    print msg
def onAfterCreate(task):
    msg = getPrintMessage('post-creation', task)
    print msg
def onBeforeDelete(task):
    msg = getPrintMessage('pre-deletion', task)
    print msg
def onAfterDelete(task):
    msg = getPrintMessage('post-deletion', task)
    print msg
def onBeforeCanUseTransfer(sourceTask, targetTask):
    msg = 'in pre-can-use-transfer with source task ' + sourceTask.Name +
    ' and target task ' + targetTask.Name
    print msg
def onAfterCanUseTransfer(sourceTask, targetTask):
    msg = 'in post-can-use-transfer with source task ' + sourceTask.Name +
    ' and target task ' + targetTask.Name
    print msg
def onBeforeCanDuplicate():
    msg = getPrintMessage('pre-can-use-transfer', None)
    print msg
def onAfterCanDuplicate():
    msg = getPrintMessage('post-can-use-transfer', None)
    print msg
def onBeforeStatus(task):
    msg = getPrintMessage('pre-status', task)
    print msg
def onAfterStatus(task):
    msg = getPrintMessage('post-status', task)
    print msg
def onBeforePropertyRetrieval(task):
    msg = getPrintMessage('pre-property', task)
    print msg
def onAfterPropertyRetrieval(task):
    msg = getPrintMessage('post-property', task)
    print msg
def getPrintMessage(msg, task):
    taskName = 'none'
    if task != None:
        taskName = task.Name
    return 'in ' + msg + ' callback for task ' + taskName
```

Custom User-Specified GUI Operation

The extension `EmptyGUI` implements a custom GUI operation for a task. Specifically, it adds a custom context menu. This functionality is valuable when you require menu entries beyond the default **Edit** menu created by the definition of the callback `<onedit>`.



XML Extension Definition File

The XML extension definition file `EmptyGUI.xml` performs the following actions:

- References the IronPython script `empty_gui.py`.
- Defines a single task in the `<tasks>` block. Within this block, a single context menu is defined in the `<contextmenus>` block. The callback `<onclick>` is defined, as required.
- Defines the inputs and outputs in the `<inputs>` and `<outputs>` blocks. Note that an empty input and an output are defined.
- Defines a single task group with a single task in the `<taskgroups>` block.

The script `EmptyGUI.xml` contains the following code:

```

<extension version="1" name="EmptyGUI">
    <guid shortid="EmptyGUI">69d0095b-e138-4841-a13a-de12238c83f6</guid>
    <script src="empty_gui.py" />
    <interface context="Project">
        <images>images</images>
    </interface>
    <workflow name="wf3" context="Project" version="1">
        <tasks>
            <task name="Empty" caption="Empty" icon="Generic_cell" version="1">
                <callbacks>
                    <onupdate>update</onupdate>
                </callbacks>
                <inputs>
                    <input/>
                </inputs>
                <outputs/>
            </task>
        </tasks>
    </workflow>
</extension>

```

```

<contextmenus>
    <entry name="Empty Gui Op" type="ContextMenuEntry" priority="1.0" icon="default_op">
        <callbacks>
            <onclick>click</onclick>
        </callbacks>
    </entry>
</contextmenus>
</task>
</tasks>
<taskgroups>
    <taskgroup name="Empty" caption="Empty" icon="Generic" category="ACT Custom Workflows"
abbreviation="MT" version="1">
        <includeTask name="Empty" caption="Empty"/>
    </taskgroup>
</taskgroups>
</workflow>
</extension>

```

IronPython Script

The IronPython script `empty_gui.py` contains the code that the task executes from the context menu and update callbacks.

Because the XML extension definition file has both the callbacks `<onupdate>` and `<onclick>`, the update and click methods are defined in the script.

The file `empty_gui.py` contains the following code:

```

import clr
clr.AddReference("Ans.UI.Toolkit")
clr.AddReference("Ans.UI.Toolkit.Base")
import Ansys.UI.Toolkit

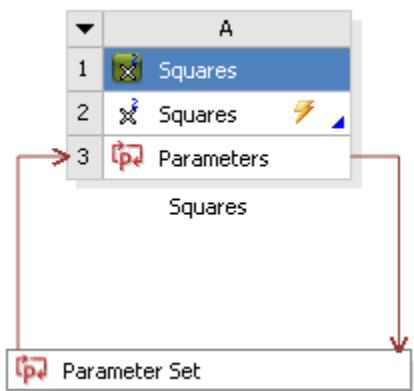
def click(task):
    Ansys.UI.Toolkit.MessageBox.Show("Empty Test!")
def update(task):
    ExtAPI.Log.WriteMessage('empty update')

```

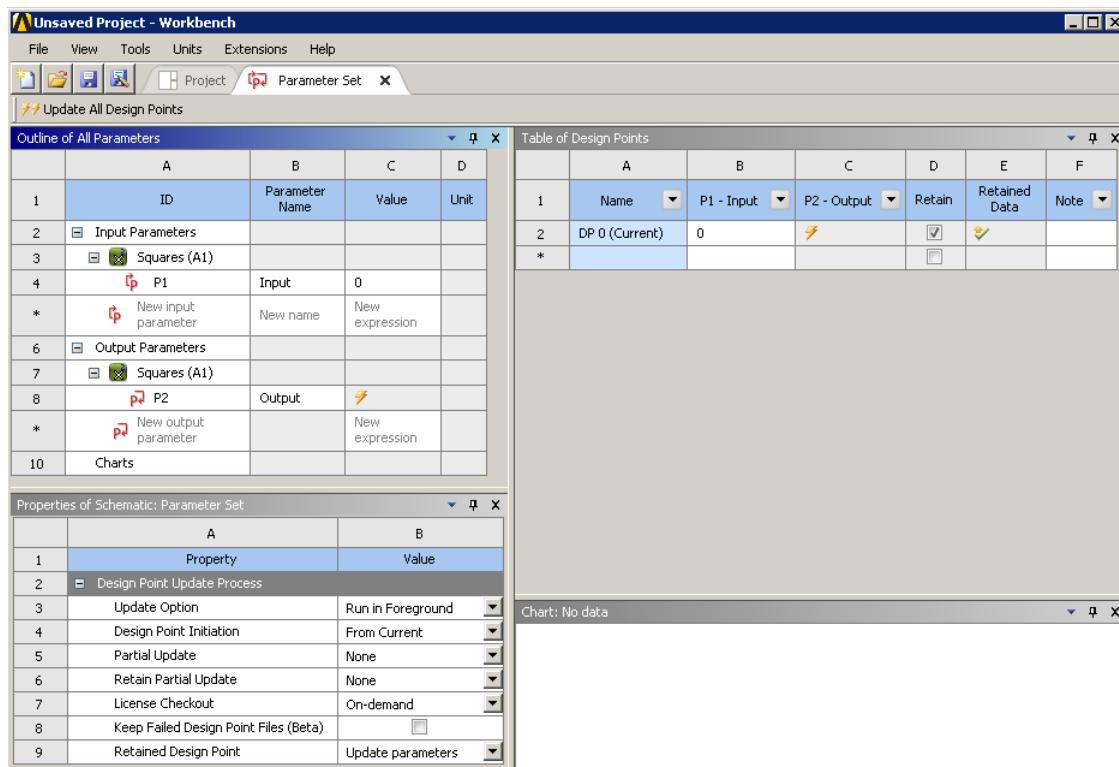
Custom, Lightweight, External Application Integration with Parameter Definition

The extension `Squares` demonstrates the use of parameter definitions to integrate an external application.

Driven by parameters defined in the XML extension definition file, the external application squares the value of an input number, which is displayed in the **Parameter Set** bar.



The external application updates the output parameter to the computed square value. The use of parameters in this example enables you to leverage DesignXplorer functionality.



XML Extension Definition File

The XML extension definition file `Squares.xml` performs the following actions:

- References the IronPython script `Squares.py`.
- Defines a single task in the `<tasks>` block.
- Defines the inputs and outputs in the `<inputs>` and `<outputs>` blocks. Note that an empty input is defined.
- Defines two parameters in the `<parameters>` block.
- Defines a single task group with a single task in the `<taskgroups>` block.

The file `Squares.xml` contains the following code:

```

<extension version="1" name="Squares">
    <guid shortid="Squares">69d0095b-e138-4841-a13a-de12238c83f4</guid>
    <script src="squares.py" />
    <interface context="Project">
        <images>images</images>
    </interface>
    <workflow name="wf1" context="Project" version="1">
        <tasks>
            <task name="Squares" caption="Squares" icon="squares_component" version="1">
                <callbacks>
                    <onupdate>update</onupdate>
                </callbacks>
                <inputs>
                    <input/>
                </inputs>
            
```

```
<outputs/>
<parameters>
    <parameter name="Input" caption="Input" usage="Input" control="Double" version="1"/>
    <parameter name="Output" caption="Output" usage="Output" control="Double" version="1"/>
</parameters>
</task>
</tasks>
<taskgroups>
    <taskgroup name="Squares" caption="Squares" icon="squares" category="ACT Custom Workflows"
abbreviation="SQRS" version="1">
        <includeTask name="Squares" caption="Squares"/>
    </taskgroup>
</taskgroups>
</workflow>
</extension>
```

IronPython Script

The IronPython script `Squares.py` performs the following actions:

- Obtains the parameters.
- Prepares the inputs.
- Writes the input file.
- Runs the external solver.
- Reads the output file.
- Sets the parameters to the calculated solver values.

Because the XML extension definition file has the callback `<onupdate>`, the method `update` is defined in the script. All ACT workflow callbacks get a `container` (for the task) and a `context`.

The script `Squares.py` contains the following code:

```
import System

import clr
clr.AddReference("Ans.Utilities")
import Ansys.Utilities

#convenience method to look up parameters based on a predetermined ID
def GetParameterByName(parameters, id):
    match = None
    for param in parameters:
        if param.ParameterName == id:
            match = param
            break
    return match

def update(task):
    container = task.InternalObject
    context = ExtAPI.DataModel.Context
    activeDir = container.GetActiveDirectory()
    extensionDir = ExtAPI.ExtensionManager.CurrentExtension.InstallDir
    exeName = "ExampleAddinExternalSolver.exe"
    solverPath = System.IO.Path.Combine(extensionDir, exeName)

    #get parameters owned by container

    params = None
    lock = context.ContainerReadLock(container)
    params = context.Project.GetDataReferencesByType(container, "ParameterAdapter")
    lock.Dispose()
```

```

#isolate specific parameters

inputParam = GetParameterByName(params, "Input")
outputParam = GetParameterByName(params, "Output")

#prep i/o file paths

inputFileName = "input.txt"
outputFileName = "output.txt"
dpInputFile = System.IO.Path.Combine(activeDir, inputFileName)
dpOutputFile = System.IO.Path.Combine(activeDir, outputFileName)

#write input file

if inputParam != None and outputParam != None:
    val = inputParam.Value
    #write input file
    f = open(dpInputFile, "w")
    f.write('input=' + val.ToString(System.Globalization.NumberFormatInfo.InvariantInfo))
    f.close()

#run exe

runInMono = Ansys.Utilities.ApplicationConfiguration.DefaultConfiguration.IsRuntimeMono
monoPath = "mono"
monoArgs = System.String.Format("{0} \\"{1}" \\"{2}\\"", solverPath, dpInputFile, dpOutputFile)
info = None
if runInMono:
    info = System.Diagnostics.ProcessStartInfo(monoPath, monoArgs)
else:
    info = System.Diagnostics.ProcessStartInfo(solverPath, System.String.Format("\\"{0}\\" \\"{1}\\"", dpInputFile, dpOutputFile))
info.CreateNoWindow = True
info.WindowStyle = System.Diagnostics.ProcessWindowStyle.Minimized
p = System.Diagnostics.Process.Start(info)
p.WaitForExit()

#read output file

outValue = None
f = open(dpOutputFile, "r")
currLine = f.readline()
while currLine != "":
    valuePair = currLine.split('=')
    outValue = System.Int32.Parse(valuePair[1], System.Globalization.NumberFormatInfo.InvariantInfo)
    currLine = f.readline()
f.close()

#set output value

if outValue == None:
    raise Exception("Error in update - no output value detected!")
else:
    outputParam.Value = outValue

```

Custom, Lightweight, External Application Integration with Custom Data and Remote Job Execution

The extension **DataSquares** demonstrates:

- The use of custom task properties to integrate an external application
- The specification that the defined task should be submitted via ANSYS Remote Solve Manager (RSM) for remote job execution

Driven by the custom task properties defined in the XML extension definition file, this external application squares the value of an input number, which is displayed in the **Parameter Set** bar. Because the task

defines an RSM job, the user can choose to either run the task update locally or submit the calculations via RSM for remote job execution. Once the job has completed, Workbench merges the job back into the active session and retrieves results. The custom task defined in the extension then updates the custom property values to the externally-computed square value.

This example also demonstrates progress and project reporting functionality. The following figure shows the **Data Squares** task group in the **Project Schematic**.



XML Extension Definition File

The XML extension definition file `DataSquares.xml` performs the following actions:

- References the IronPython script `datasquares_complete.py`.
- Defines a single task in the `<tasks>` block. Within this task:
 - Defines the callback `<onreport>`, which is called when the user generates a project report. This allows the task to access the report object and add its own task-specific reporting content.
 - Defines two property groups with the `<propertygroup>` blocks. Within the two property groups, `<property>` tags define the properties `Input` and `Output`.
 - Defines the inputs and outputs in the `<inputs>` and `<outputs>` blocks. Note that an empty input is defined here, although this is not required.
 - Specifies via the `<rsmJob>` block that the task can be sent via RSM as a job for remote execution. To specify RSM support, you add the `<rsmJob>` block to the task in the existing XML extension definition file. In the RSM job specification:
 - Attributes specify the name and indicate the files to manage upon successful completion or cancellation of the job.
 - The `<platform>` tags specify supported platforms.
 - The `<argument>` tags specify the external application's command line parameters, including input and output file paths.
 - When using referenced file paths, the same `ID` can be used across categories but not within categories. An input and an output can both have an `ID` of 1, but two inputs cannot have an `ID` of 1. During the execution, all input and output file names are transformed into fully qualified file paths.
 - The `<program>` block informs the extension about the task's executable, supported platforms, and corresponding command line arguments. Environment variables can be used for the executable path and previously defined arguments can be referenced by ID.

- The callbacks invoke methods defined in the IronPython script to create the job, check job status, provide the capability to cancel the job, and access output files and results.
- Defines a single task group in the **<taskgroups>** block that includes the task **DataSquares**

The file **DataSquares.xml** contains the following code:

```

<extension version="1" name="DataSquares">
    <guid shortid="DataSquares">69d0095b-e138-4841-a13a-de12238c83f2</guid>
    <script src="datasquares_complete.py" />
    <interface context="Project">
        <images>images</images>
    </interface>
    <workflow name="MyWorkflow" context="Project" version="1">
        <tasks>
            <task name="DataSquares" caption="Data Squares" icon="dsquares_component" version="1">
                <callbacks>
                    <onupdate>update</onupdate>
                    <onstatus>status</onstatus>
                    <onreport>report</onreport>
                    <onreset>reset</onreset>
                </callbacks>
                <propertygroup name="Inputs">
                    <property name="Input" caption="Input" control="double" default="0.0" readonly="False" needupdate="true" visible="True" persistent="True" isparameter="True" />
                </propertygroup>
                <propertygroup name="Outputs">
                    <property name="Output" caption="Output" control="double" default="0.0" readonly="True" visible="True" persistent="True" isparameter="True" />
                </propertygroup>
                <inputs>
                    <input/>
                </inputs>
                <outputs/>
                <rsmjob name="squares" deletefiles="True" version="1">
                    <inputfile id="1" name="input1.txt"/>
                    <outputfile id="1" name="output1.txt"/>
                    <program>
                        <platform name="Win64" path="%AWP_ROOT171%\Addins\ACT\extensions\Datasquares\ExampleAddinExternalSolver.exe" />
                        <platform name="Linux64" path="%AWP_ROOT171%\Addins\ACT\extensions\Datasquares\ExampleAddinExternalSolver.exe" />
                            <argument name="" value="inputFile:1" separator=""/>
                            <argument name="" value="outputFile:1" separator=""/>
                    </program>
                    <callbacks>
                        <oncreatejobinput>createJobInput</oncreatejobinput>
                        <onjobstatus>getJobStatus</onjobstatus>
                        <onjobcancellation>cancelJob</onjobcancellation>
                        <onjobreconnect>reconnectJob</onjobreconnect>
                    </callbacks>
                </rsmjob>
            </task>
        </tasks>
        <taskgroups>
            <taskgroup name="DataSquares" caption="Data Squares" icon="dsquares" category="ACT Custom Workflows" abbreviation="DSQRS" version="1">
                <includeTask name="DataSquares" caption="Data Squares" />
            </taskgroup>
        </taskgroups>
    </workflow>
</extension>

```

The following callbacks reference methods in the IronPython script. The first three callbacks invoke standard task actions. The subsequent callbacks invoke actions related to the RSM job specification.

<onupdate>

Invokes the method **update**. Called when the user updates the task.

<onstatus>

Invokes the method **status**. Called when the product (such as Workbench) asks the task for its current state.

<onreport>

Invokes the method **report**. Called when the user generates a project report. This allows the extension to access the report object and add their own task-specific reporting content.

<oncreatejobinput>

Invokes the method **createJobInput**. Called when the RSM job requires the generation of all specified input files. Must prepare and generate the input files specified for the RSM job.

As a method argument, ACT passes in the fully qualified input file paths. For this example, only one input file is generated. It contains the input value for the squares solver. ACT retrieves the first entry from the input file list, opens the file, writes the input value line, and closes the file. ACT requires no further action.

<onjobstatus>

Invokes the method **getJobStatus**. Must tell Workbench if the job has finished. Polled by Workbench several times during the RSM job execution to determine if the update execution has completed. ACT supplies the fully qualified output file list as a method argument.

For the example, **True** is returned when the presence of the solver's sole output file is detected. Otherwise, **False** is returned.

<onjobcancellation>

Invokes the method **cancelJob**. Called when the user stops the remote update before the job completes. Must conduct any clean-up actions required due to a user-initiated stoppage. ACT supplies the fully qualified lists of both input and output files as method arguments.

For this example, no other action is performed. Only file cleanup is required. Setting the method **DeleteFiles** to **True** automatically handles the file cleanup.

<onjobreconnect>

Invokes the method **reconnectJob**. Called when the job status reports a completed job. Must retrieve remotely solved information from output files and update the task's data, rendering it up-to-date.

For this example, the output file generated by the solver is read and the output property that is defined on the task is updated.

The next topic provides additional information about these methods.

IronPython Script

The IronPython script **dataquares_complete.py** defines methods that are invoked by callbacks in the XML extension definition file. The methods **update**, **status**, and **report** are standard methods defined for the task. The methods **createJobInput**, **getJobStatus**, **cancelJob**, and **reconnectJob** are related to the RSM job specification.

```
import clr
clr.AddReference("Ans.Core")

def update(task):
    container = task.InternalObject
    context = ExtAPI.DataModel.Context
    activeDir = container.GetActiveDirectory()
```

```

extensionDir = ExtAPI.ExtensionManager.CurrentExtension.InstallDir
exeName = "ExampleAddinExternalSolver.exe"
solverPath = System.IO.Path.Combine(extensionDir, exeName)

monitor = context.ProgressMonitor
monitor.BeginTask("Data Squares Solver", 3, None)
monitor.TaskDetails = "Preparing solver input..."
System.Threading.Thread.Sleep(2000)
monitor.UpdateTask(1, None)

#get param values
inputValue = task.Properties["Inputs"].Properties["Input"].Value

#prep i/o file paths

inputFileName = "input.txt"
outputFileName = "output.txt"
dpInputFile = System.IO.Path.Combine(activeDir, inputFileName)
dpOutputFile = System.IO.Path.Combine(activeDir, outputFileName)

#write input file
f = open(dpInputFile, "w")
f.write('input=' + inputValue.ToString(System.Globalization.NumberFormatInfo.InvariantInfo))
f.close()

monitor.UpdateTask(1, None)

monitor.TaskDetails = "Executing Solver..."
System.Threading.Thread.Sleep(2000)

#run exe

runInMono = Ansys.Utilities.ApplicationConfiguration.DefaultConfiguration.IsRuntimeMono
monoPath = "mono"
monoArgs = System.String.Format("{0} \"{1}\" \"{2}\"", solverPath, dpInputFile, dpOutputFile)
info = None
if runInMono:
    info = System.Diagnostics.ProcessStartInfo(monoPath, monoArgs)
else:
    info = System.Diagnostics.ProcessStartInfo(solverPath, System.String.Format("\\"{0}" \"{1}\"", dpInputFile, dpOutputFile))
info.CreateNoWindow = True
info.WindowStyle = System.Diagnostics.ProcessWindowStyle.Minimized
p = System.Diagnostics.Process.Start(info)
p.WaitForExit()

monitor.UpdateTask(1, None)

monitor.TaskDetails = "Retrieving results from solver..."
System.Threading.Thread.Sleep(2000)

#read output file

outputValue = None
f = open(dpOutputFile, "r")
currLine = f.readline()
while currLine != "":
    valuePair = currLine.split('=')
    outputValue = System.Double.Parse(valuePair[1], System.Globalization.NumberFormatInfo.InvariantInfo)
    currLine = f.readline()
f.close()

monitor.UpdateTask(1, None)

#set output value

if outputValue == None:
    raise Exception("Error in update - no output value detected!")
else:
    task.Properties["Outputs"].Properties["Output"].Value = outputValue
monitor.TaskDetails = "Solve completed..."
System.Threading.Thread.Sleep(2000)

```

```
monitor.EndTask(None)

def createJobInput(task, inputFilePaths):
    ExtAPI.Log.WriteMessage('creating job input')
    inputFilePath = inputFilePaths[0]
    #get param values
    inputValue = task.Properties["Inputs"].Properties["Input"].Value

    #write input file
    ExtAPI.Log.WriteMessage("Writing input value ("+str(inputValue)+" to file (" + inputFilePath + ")")
    f = open(inputFilePath, "w")
    f.write('input=' + inputValue.ToString(System.Globalization.NumberFormatInfo.InvariantInfo))
    f.close()

def reconnectJob(task, outputFilePaths):
    ExtAPI.Log.WriteMessage('reconnecting job')
    outputValue = None
    outputPath = outputFilePaths[0] #I know we only have one specified based on our definition...so work
    off of the first entry
    f = open(outputFilePath, "r")
    currLine = f.readline()
    while currLine != "":
        valuePair = currLine.split('=')
        outputValue = System.Double.Parse(valuePair[1], System.Globalization.NumberFormatInfo.InvariantInfo)
        currLine = f.readline()
    f.close()
    #set output value
    ExtAPI.Log.WriteMessage("Retrieved value (" + str(outputValue) + ") from file (" + outputPath + ")")
    if outputValue == None:
        raise Exception("Error in update - no output value detected!")
    else:
        task.Properties["Outputs"].Properties["Output"].Value = outputValue

def getJobStatus(task, outputFiles):
    ExtAPI.Log.WriteMessage('checking job status')
    outputPath = outputFiles[0]
    finished = System.IO.File.Exists(outputPath)
    return finished

def cancelJob(task, inputFiles, outputFiles):
    #nothing to do...just print a message for now.
    ExtAPI.Log.WriteMessage('performing cancellation clean up')

import clr
clr.AddReference("Ans.ProjectSchematic")
clr.AddReference("ReportUtility.Interop")
import Ansys.ReportUtility.Interop
import Ansys.ProjectSchematic

def status(task):
    status = Ansys.ProjectSchematic.Queries.ComponentState(Ansys.ProjectSchematic.State.Unfulfilled,
    "This is unfulfilled!")
    return None

def report(task, report):
    root = report.GetRootSection()
    section = Ansys.ReportUtility.Interop.ReportSection("My Custom ACT Task Report Content")
    text = Ansys.ReportUtility.Interop.ReportText("", "Sample text from the data squares component")
    section.AddChild(text)
    root.AddChild(section)

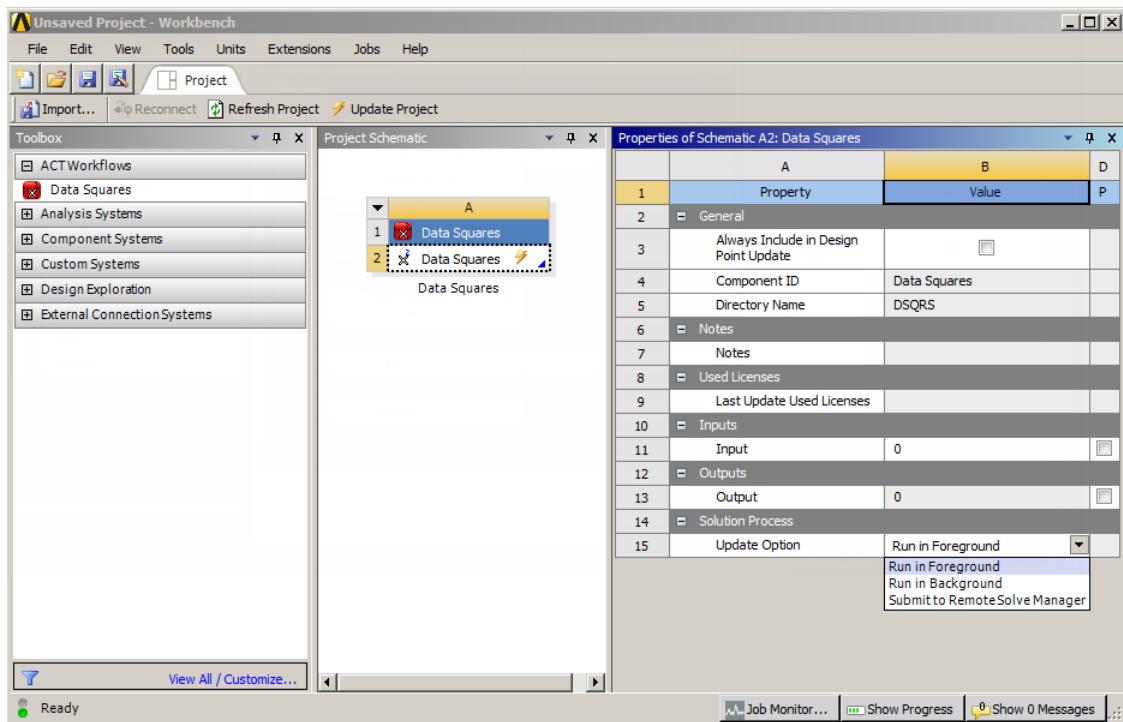
def reset(task):
    task.Properties["Inputs"].Properties["Input"].Value = 0
    task.Properties["Outputs"].Properties["Output"].Value = 0
```

Using RSM Job Submission Capabilities

Once you've defined an extension with an RSM job specification, you can submit your task as an RSM job for remote execution:

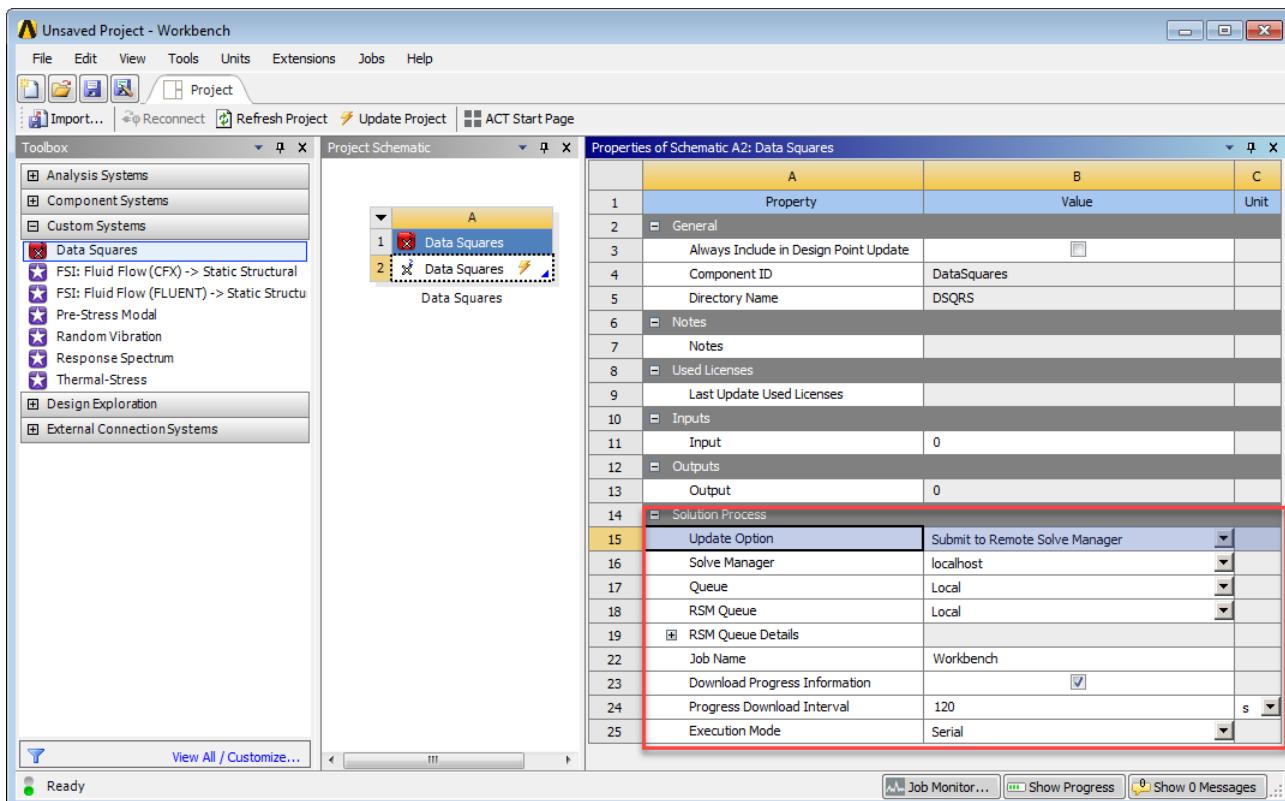
1. Install and load the extension to Workbench as usual.
2. Add a **Data Squares** task group to the **Project Schematic**.

Solution Process properties become available for the task group.



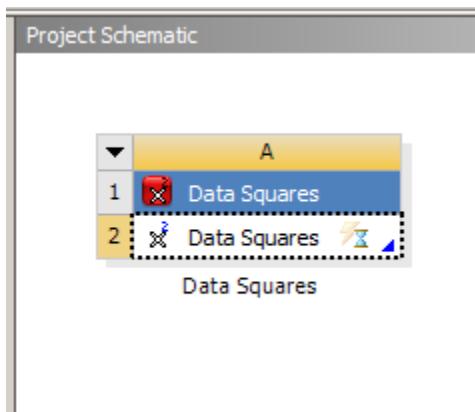
- To specify that your task is to be executed remotely, set **Update Option** to **Submit to Remote Solve Manager**.

Additional **Solution Process** properties become available.

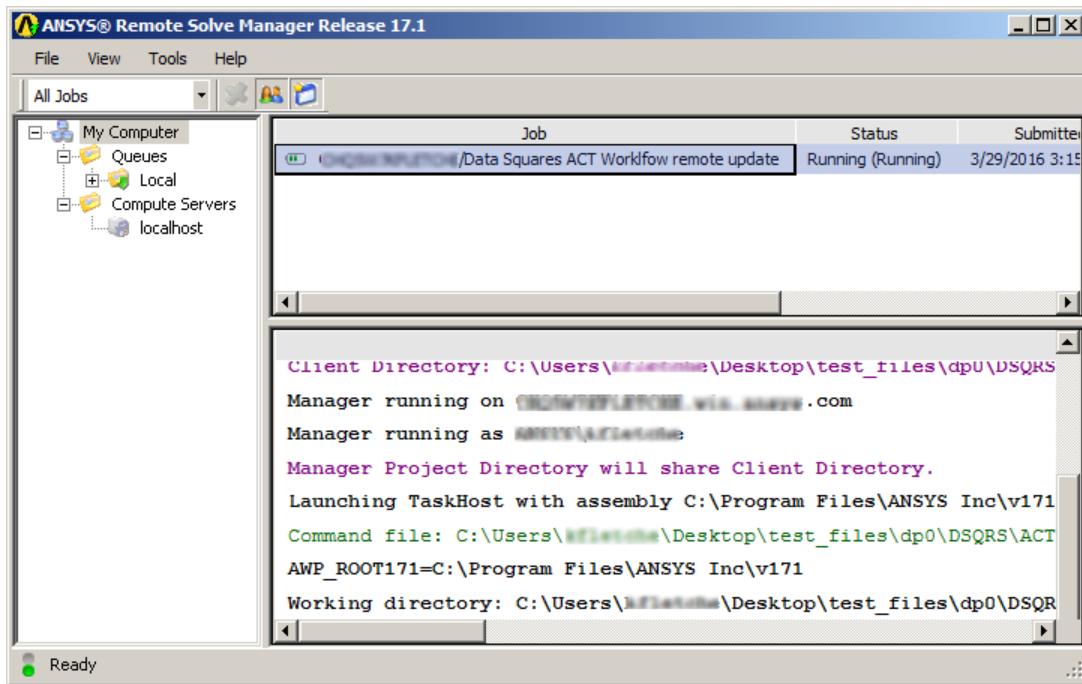


- Set your **Solution Process** properties. Then, save and update your project.

The task submits its process to RSM as a new job. Once RSM accepts the job, the task transitions to the pending state.



- To view the status of your RSM job while it is being updated, open RSM.



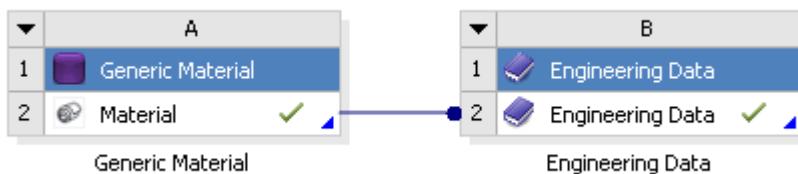
Note

To run your jobs locally in the background, set **Update Option** to **Run in Background**. Updates are then treated in a similar manner to true remote updates. Workbench prepares the update and invokes the same callbacks as in the example, but the update proceeds as a separate local update instead of engaging RSM.

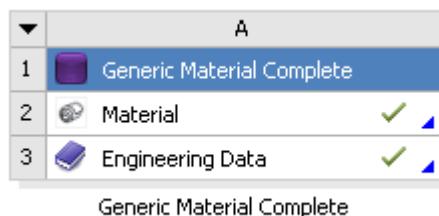
Generic Material Transfer

The extension **GenericMaterialTransfer** implements two custom material transfer task groups, each passing material data to a downstream task group or task.

- The first task group references a single custom task that passes MatML-formatted material data to a downstream external **Engineering Data** task group.



- The second task group references both a custom task and an external task. The custom **Material** task passes MatML-formatted material data to the downstream external **Engineering Data** task.



This example also demonstrates input and output specification and file management capabilities.

Screenshot of the ANSYS Workbench interface showing the 'Engineering Data' schematic:

- Outline of Schematic B2: Engineering Data:**
 - Row 1: Contents of Engineering Data (Source)
 - Row 2: Material (Sample Material from Driver)
 - Row 3: Sample Material (Structural Steel, Fatigue Data at zero mean stress comes from 1998 ASME BPV Code, Section 8, Div 2, Table 5-110.1)
 - Row 4: Structural Steel
 - Row *: Click here to add a new material
- Table of Properties Row 6: Sample Property:**

	A	B
1	Strain (m m ⁻¹)	Stress (Pa)
2	0.1338	494.15
3	0.2675	912.8
4	0.3567	1172.5
5	0.6242	1941.5
6	0.8917	2803.8
7	1.1592	3869.1
8	1.4268	5245.4
9	2.051	10379
10	2.586	18193
11	3.0318	28439
12	3.7898	57755
- Properties of Outline Row 3: Sample Material:**

	A	B	C	D	E
1	Property	Value	Unit	(X)	(P)
2	Driver Link Details				
3	Data Link Version	1			
4	Model Type	Linear;Isotropic			
5	Sample Property	Value			
6	Sample Property	Tabular			
7	Scale	1			
8	Offset	0	Pa		
- Chart of Properties Row 6: Sample Property:**

Stress (x10³) [Pa] vs Strain [m m⁻¹]

Graph showing a linear relationship between Stress and Strain, starting from (0,0) and ending at approximately (7, 5).

XML Extension Definition File

The XML extension definition file `GenericMaterialTransfer.xml` specifies the custom task groups that are to appear in your custom **ACT Workflows** task group in the Workbench **Toolbox**.

This XML file performs the following actions:

- References the IronPython script `generic_material_transfer.py`.
- Defines a single task in the `<tasks>` block.
- Specifies that the callback `<onupdate>` invokes the IronPython function `producer_update`, which accesses a materials file.
- Defines the inputs and outputs in the `<inputs>` and `<outputs>` blocks. Note that an empty input and an output are defined. The attribute `outputtype` is set to `MatML31`, specifying the kind of data exposed and generated by this task.
- Defines two custom task groups in the `<taskgroups>` block.
 - The first task group, `GenericMaterialTransfer`, references the custom task `Material` and transfers material data to an external task group `Engineering Data` that has been added downstream.
 - The second task group, `GenericMaterialTransferComplete`, references the custom task `Material` and also, by using the attribute `external`, the external task `Engineering Data`. The custom task transfers material data to the downstream external task.

The file `GenericMaterialTransfer.xml` contains the following code:

```
<extension version="1" name="GenericMaterialTransfer">
    <guid shortid="GenericMaterialTransfer">69d0095b-e138-4841-a13a-de12238c83f8</guid>
    <script src="generic_material_transfer.py" />
    <interface context="Project">
        <images>images</images>
    </interface>
    <workflow name="wf5" context="Project" version="1">
        <tasks>
            <task name="Material" caption="Material" icon="material_cell" version="1">
                <callbacks>
                    <onupdate>update</onupdate>
                </callbacks>
                <inputs>
                    <input/>
                </inputs>
                <outputs>
                    <output format="" type="MatML31"/>
                </outputs>
            </task>
        </tasks>
        <taskgroups>
            <taskgroup name="GenericMaterialTransfer" caption="Generic Material" icon="material_system" category="ACT Custom Workflows" abbreviation="GenMatXfer" version="1">
                <includeTask name="Material" caption="Material"/>
            </taskgroup>
            <taskgroup name="GenericMaterialTransferComplete" caption="Generic Material Complete" icon="material_system" category="ACT Custom Workflows" abbreviation="GenMatXferComplete" version="1">
                <includeTask name="Material" caption="Material"/>
                <includeTask external="True" name="EngDataCellTemplate" caption="Engineering Data"/>
            </taskgroup>
        </taskgroups>
    </workflow>
</extension>
```

IronPython Script

The IronPython script `generic_material_transfer.py` contains the code for passing the MatML-formatted material data to a downstream `Engineering Data` task group or task. This includes the method `update` (invoked by the callback `<onupdate>` in the XML file), which accesses the file `SampleMaterials.xml`.

The script `generic_material_transfer.py` contains the following code:

```
def update(task):
    container = task.InternalObject
    extensionDir = ExtAPI.ExtensionManager.CurrentExtension.InstallDir
    matFilePath = System.IO.Path.Combine(extensionDir, "Sample_Materials.xml")
    matFileRef = None
    isRegistered = IsFileRegistered(FilePath=matFilePath)
    if isRegistered == True:
        matFileRef = GetRegisteredFile(matFilePath)
    else:
        matFileRef = RegisterFile(FilePath=matFilePath)
        AssociateFileWithContainer(matFileRef, container)
    outputRefs = container.GetOutputData()
    matOutputSet = outputRefs["MatML31"]
    matOutput = matOutputSet[0]
    matOutput.TransferFile = matFileRef
```

Material File

The file `Sample_Materials.xml` that is accessed by the IronPython method `update` contains the MatML-formatted material data:

```
<?xml version="1.0" encoding="UTF-8"?>
<EngineeringData version="16.1">
    <Notes />
    <Materials>
        <MatML_Doc>
            <Material>
                <BulkDetails>
                    <Name>Sample Material</Name>
                    <Description>Sample material from Driver</Description>
                    <PropertyData property="pr0">
                        <Data format="string">-</Data>
                        <ParameterValue parameter="pa0" format="float">
                            <Data>494.1474492,912.7972764,1172.453938,1941.495468,2803.754154,3869.063522,5245.395513,
                            10378.82012,18192.58268,28438.67868,57755.1982,94951.87682,135751.6191,178064.7612,216504.4272,261538.9311,
                            304701.5076,333300.2826,364061.2544,397079.5705,432533.1159,457543.8578,483751.5301</Data>
                            <Qualifier name="Variable Type">Dependent,Dependent,Dependent,Dependent,Dependent,Dependent,
                            Dependent,Dependent,Dependent,Dependent,Dependent,Dependent,Dependent,Dependent,Dependent,
                            Dependent,Dependent,Dependent,Dependent,Dependent,Dependent,Dependent,Dependent,Dependent</Qualifier>
                        </ParameterValue>
                        <ParameterValue parameter="pal" format="float">
                            <Data>0.1338,0.2675,0.3567,0.6242,0.8917,1.1592,1.4268,2.051,2.586,3.0318,3.7898,4.3694,4.8153,
                            5.172,5.4395,5.707,5.9299,6.0637,6.1975,6.3312,6.465,6.5541,6.6433</Data>
                            <Qualifier name="Variable Type">Independent,Independent,Independent,Independent,Independent,
                            Independent,Independent,Independent,Independent,Independent,Independent,Independent,Independent,
                            Independent,Independent,Independent,Independent,Independent,Independent,Independent,Independent,
                            Independent,Independent,Independent,Independent,Independent,Independent,Independent,Independent,
                            Independent</Qualifier>
                        </ParameterValue>
                    </PropertyData>
                    <PropertyData property="prDriver">
                        <Data format="string">-</Data>
                        <Qualifier name="Data Link Version">1</Qualifier>
                        <Qualifier name="Model Type">Linear;Isotropic</Qualifier>
                        <Qualifier name="Sample Property">Value</Qualifier>
                    </PropertyData>
                </BulkDetails>
            </Material>
            <Metadata>
                <ParameterDetails id="pa0">
                    <Name>Stress</Name>
                    <Units>
                        <Unit>
                            <Name>Pa</Name>
                        </Unit>
                    </Units>
                </ParameterDetails>
                <ParameterDetails id="pal">
                    <Name>Strain</Name>
                </ParameterDetails>
```

```

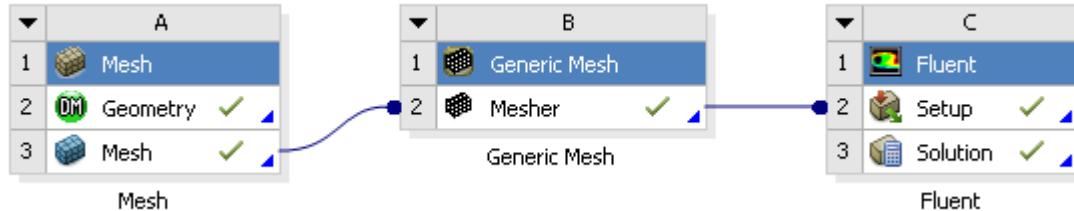
<Units>
  <Unit>
    <Name>m</Name>
  </Unit>
  <Unit power="-1">
    <Name>m</Name>
  </Unit>
</Units>
</ParameterDetails>
<PropertyDetails id="pr0">
  <Unitless />
  <Name>Sample Property</Name>
</PropertyDetails>
<PropertyDetails id="prDriver">
  <Unitless />
  <Name>Driver Link Details</Name>
</PropertyDetails>
</Metadata>
</MatML_Doc>
</Materials>
<Loads />
<BeamSections />
</EngineeringData>

```

Generic Mesh Transfer

The extension **GenericMeshTransfer** implements two custom mesh transfer task groups, each with "consuming" and "providing" connections.

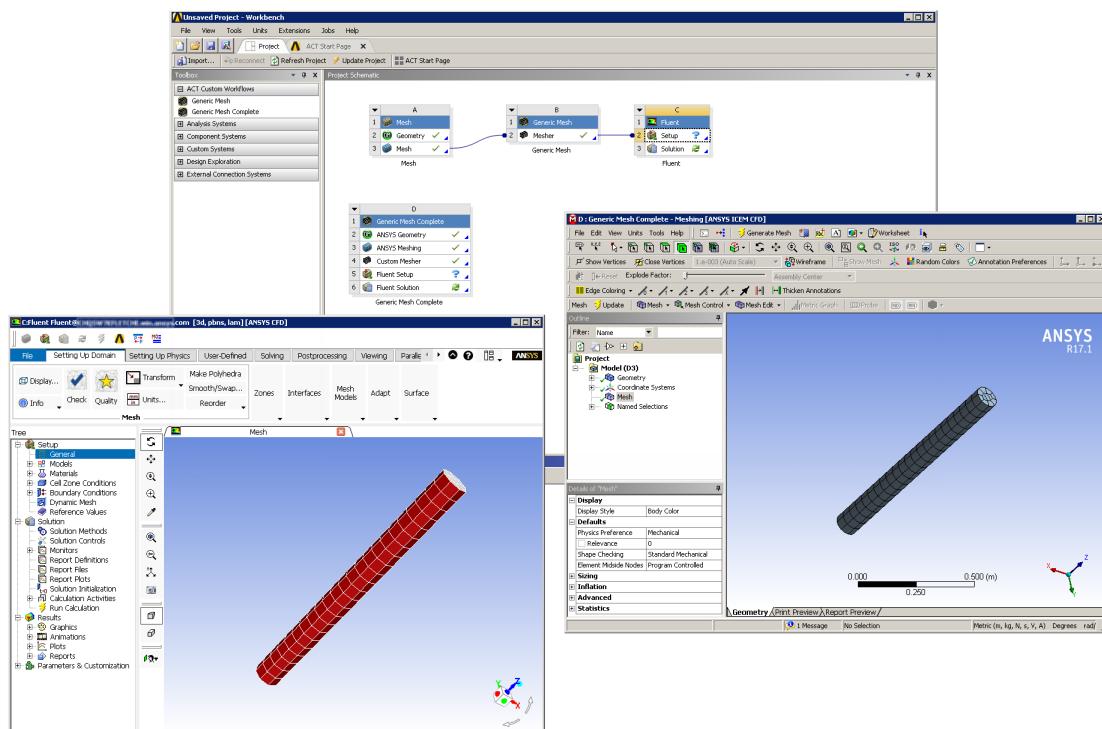
- The first custom task group references a single custom task. It consumes a mesh from an upstream external task group and passes it to a downstream Workbench task group.



- The second custom task group references both a custom task and multiple external tasks. The custom task consumes a mesh from an upstream external task and passes it to a downstream external task.



This example also demonstrates input and output specification, file management capabilities, and the default **Edit** context menu.



XML Extension Definition File

The XML extension definition file `GenericMeshTransfer.xml` specifies the two custom task groups that are to appear in your custom **ACT Workflows** task group in the Workbench **Toolbox**. This XML file performs the following actions:

- References the IronPython script `generic_mesh_transfer.py`.
- Defines a single task in the `<tasks>` block.
- Defines the callback `<onedit>` in the task block, which automatically creates a default **Edit** context menu for the task.
- Defines an input and an output in the `<inputs>` and `<outputs>` blocks.
 - The input has `type` set to **MeshingMesh**, indicating that the input data type is a mesh.
 - Both the input and output have `format` set to **FluentMesh**, specifying that the input and output files have the same **FluentMesh** format.
- Defines two custom task groups in the `<taskgroups>` block.
 - The first custom task group, **GenericMeshTransfer**, references the custom task **Mesh**. It consumes a mesh from an upstream **Mesh** task group and passes it to a downstream **Fluent** task group.
 - The second custom task group, **GenericMeshTransferComplete**, references the custom task **Material** and also, by using the attribute `external`, the external tasks **ANSYS Geometry**, **ANSYS Meshing**, **Fluent Setup**, and **Fluent Solution**. The task **Material** consumes a mesh from the upstream task **ANSYS Meshing** and passes it to the downstream task **Fluent Setup**.

The file `GenericMeshTransfer.xml` contains the following code:

```
<extension version="1" name="GenericMeshTransfer">
<guid shortid="GenericMeshTransfer">69d0095b-e138-4841-a13a-de12238c83f7</guid>
<script src="generic_mesh_transfer.py" />
<interface context="Project">
    <images>images</images>
</interface>
<workflow name="wf6" context="Project" version="1">
    <tasks>
        <task name="Mesher" caption="Mesher" icon="GenericMesh_cell" version="1">
            <callbacks>
                <onupdate>update</onupdate>
                <onedit>edit</onedit>
            </callbacks>
            <inputs>
                <input format="FluentMesh" type="MeshingMesh" count="1"/>
                <input/>
            </inputs>
            <outputs>
                <output format="FluentMesh" type="SimulationGeneratedMesh" />
            </outputs>
        </task>
    </tasks>
    <taskgroups>
        <taskgroup name="GenericMeshTransfer" caption="Generic Mesh" icon="GenericMesh" category="ACT Custom Workflows" abbreviation="GenMeshXfer" version="1">
            <includeTask name="Mesher" caption="Mesher"/>
        </taskgroup>
        <taskgroup name="GenericMeshTransferComplete" caption="Generic Mesh Complete" icon="GenericMesh" category="ACT Custom Workflows" abbreviation="GenMeshXferComplete" version="1">
            <includeTask external="True" name="GeometryCellTemplate" caption="ANSYS Geometry"/>
            <includeTask external="True" name="SimulationMeshingModelCellTemplate" caption="ANSYS Meshing"/>
            <includeTask name="Mesher" caption="Custom Mesher"/>
            <includeTask external="True" name="FluentSetupCellTemplate" caption="Fluent Setup"/>
            <includeTask external="True" name="FluentResultsCellTemplate" caption="Fluent Solution"/>
        </taskgroup>
    </taskgroups>
    </workflow>
</extension>
```

IronPython Script

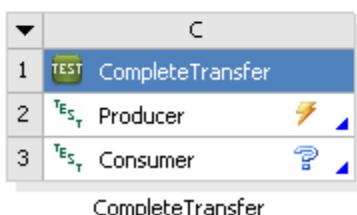
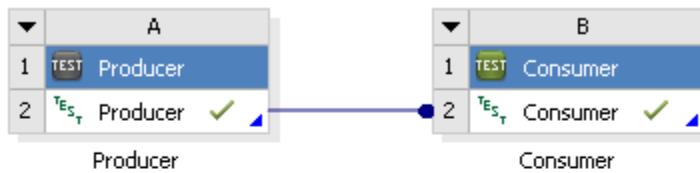
The IronPython script generic_mesh_transfer.py contains the code for passing the mesh data to the downstream task group or task.

```
import clr
clr.AddReference("Ans.UI.Toolkit")
clr.AddReference("Ans.UI.Toolkit.Base")
import Ansys.UI.Toolkit

def update(task):
    container = task.InternalObject
    ExtAPI.Log.WriteMessage('in generic_mesh_transfer.py update method')
    #obtain input data
    upstreamData = container.GetInputDataByType(InputType="MeshingMesh")
    meshFileRef = None
    upstreamDataCount = upstreamData.Count
    if upstreamDataCount > 0:
        meshFileRef = upstreamData[0]
        #set our output so that we are just a pass through.
        outputRefs = container.GetOutputData()
        meshOutputSet = outputRefs["SimulationGeneratedMesh"]
        meshOutput = meshOutputSet[0]
        #meshOutput.MeshFile = meshFileRef
        meshOutput.TransferFile = meshFileRef
        ExtAPI.Log.WriteMessage(str(meshFileRef))
    #if no new data...nothing to process from upstream sources.
def edit(task):
    Ansys.UI.Toolkit.MessageBox.Show("Test!")
```

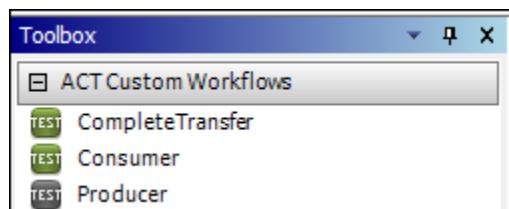
Custom Transfer

The extension **CustomTransfer** implements a custom transfer from a producing task group to a consuming task group, creating connections between custom tasks (no ANSYS installed products). It also demonstrates the creation of single-task task group versus a multi-task task group.



XML Extension Definition File

The XML extension definition file **CustomTransfer.xml** defines task groups named **Producer**, **Consumer**, and **CompleteTransfer**, all of which appear in the Workbench **Toolbox**.



This XML file performs the following actions:

- References the IronPython script `customtransfer.py`.
- Defines two tasks in the `<tasks>` block: **Producer** and **Consumer**.
- Defines three task groups in the `<taskgroups>` block: **Producer**, **Consumer**, and **CompleteTransfer**. The task groups **Producer** and **Consumer** each contain a single task. The task group **CompleteTransfer** contains two tasks.

The file **CustomTransfer.xml** contains the following code:

```

<extension version="1" name="CustomTransfer">
    <guid shortid="CustomTransfer">69d0095b-e138-4841-a13a-de12238c83f3</guid>
    <script src="customtransfer.py" />
    <interface context="Project">
        <images>images</images>
    </interface>
    <workflow name="wf4" context="Project" version="1">
        <tasks>
            <task name="Producer" caption="Producer" icon="test_component" version="1">
                <callbacks>

```

```
<onupdate>producer_update</onupdate>
</callbacks>
<inputs>
    <input/>
</inputs>
<outputs>
    <output format="" type="MyData"/>
</outputs>
</task>
<task name="Consumer" caption="consumer" icon="test_component" version="1">
    <callbacks>
        <onupdate>consumer_update</onupdate>
    </callbacks>
    <inputs>
        <input/>
        <input format="" type="MyData"/>
    </inputs>
    <outputs/>
</task>
</tasks>
<taskgroups>
    <taskgroup name="Producer" caption="Producer" icon="producer_system"
category="ACT Custom Workflows" abbreviation="Producer" version="1">
        <includeTask name="Producer" caption="Producer"/>
    </taskgroup>
    <taskgroup name="Consumer" caption="Consumer" icon="consumer_system"
category="ACT Custom Workflows" abbreviation="Consumer" version="1">
        <includeTask name="Consumer" caption="Consumer"/>
    </taskgroup>
    <taskgroup name="CompleteTransfer" caption="CompleteTransfer" icon="consumer_system"
category="ACT Custom Workflows" abbreviation="CompleteTransfer" version="1">
        <includeTask name="Producer" caption="Producer"/>
        <includeTask name="Consumer" caption="Consumer"/>
    </taskgroup>
</taskgroups>
</workflow>
</extension>
```

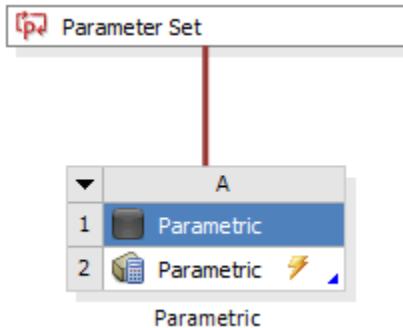
IronPython Script

The IronPython script `customtransfer.py` contains the code that provides update instructions for the producing task group and for the consuming task to obtain the output data from the upstream producer.

```
def consumer_update(task):
    container = task.InternalObject
    #obtain input data
    upstreamData = container.GetInputDataByType( InputType="MyData" )
    fileRef = None
    upstreamDataCount = upstreamData.Count
    if upstreamDataCount > 0:
        fileRef = upstreamData[0]
        AssociateFileWithContainer(fileRef, container)
        ExtAPI.Log.WriteMessage("Recieved file "+fileRef.Location+" from producer.")
    #if no new data...nothing to process from upstream sources.
def producer_update(task):
    container = task.InternalObject
    extensionDir = ExtAPI.ExtensionManager.CurrentExtension.InstallDir
    filePath = System.IO.Path.Combine(extensionDir, "Sample_Materials.xml")
    fileRef = None
    isRegistered = IsFileRegistered(FilePath=filePath)
    if isRegistered == True:
        fileRef = GetRegisteredFile(filePath)
    else:
        fileRef = RegisterFile(FilePath=filePath)
        AssociateFileWithContainer(fileRef, container)
    outputRefs = container.GetOutputData()
    outputSet = outputRefs["MyData"]
    myData = outputSet[0]
    myData.TransferFile = fileRef
```

Parametric

The extension **Parametric** demonstrates the creation of a parametric task group using the attribute **isparametricgroup**. When set to **true**, this attribute indicates that the task group operates only on design points. As such, the task group is added below the **Parameter Set** bar. The focus on parameters in this example enables you to incorporate DesignXplorer-like functionality.



XML Extension Definition File

The XML extension definition file `Parametric.xml` performs the following actions:

- References the IronPython script `parametric.py`.
- Defines a single task in the `<tasks>` block.
- Defines the inputs and outputs in the `<inputs>` and `<outputs>` blocks. Note that an empty input is defined.
- Defines a single task group with a single task in the `<taskgroups>` block. Note that the attribute **isparametricgroup** is set to **true**.

The file `Parametric.xml` follows:

```

<extension version="1" name="Parametric">
    <guid shortid="Parametric">69d0095b-e138-4841-a13a-de12238c83f5</guid>
    <script src="parametric.py" />
    <interface context="Project">
        <images>images</images>
    </interface>
    <workflow name="wf2" context="Project" version="1">
        <tasks>
            <task name="Parametric" caption="Parametric" icon="parametric_component" version="1">
                <callbacks>
                    <onupdate>update</onupdate>
                </callbacks>
                <inputs>
                    <input/>
                </inputs>
                <outputs/>
            </task>
        </tasks>
        <taskgroups>
            <taskgroup name="Parametric" caption="Parametric" icon="parametric" category="ACT Custom Workflows" abbreviation="PARAMS" isparametricgroup="True" version="1">
                <includeTask name="Parametric" caption="Parametric"/>
            </taskgroup>
        </taskgroups>
    </workflow>
</extension>

```

IronPython Script

The IronPython script parametric.py performs an update on the parameters. Because the XML extension definition file has the callback <onupdate>, the method **update** is defined in the script.

```
def update(task):
    ExtAPI.Log.WriteMessage("test")
```

Wizard Examples

This section provides examples for each type of simulation wizard:

[Project Wizard \(Workbench Project Tab\)*](#)
[Project Wizard \(AIM Project Tab\)](#)
[DesignModeler Wizard*](#)
[SpaceClaim Wizard*](#)
[Mechanical Wizard*](#)
[Mixed Wizard*](#)
[Fluent Wizard \(MSH Input File\)](#)
[Fluent Wizard \(CAS Input File\)](#)
[Electronics Desktop Wizard](#)
[SpaceClaim Wizard](#)
[AIM Custom Template \(Single-Step\)](#)
[AIM Custom Template \(Multiple-Step\)](#)
[Wizard Custom Layout Example](#)

Note

- The wizard examples marked above with an asterisk are all defined in the same extension, **WizardDemos**. All other wizard examples are defined in separate extensions.
 - All but the Fluent wizard examples are included in the package **ACT Developer's Guide Examples**, which you can download from the [ACT Resources](#) page on the [ANSYS Customer Portal](#). For more information, see [Extension Examples \(p. 273\)](#).
 - The two Fluent wizard examples and their sample input files are included in the package **ACT Wizards Templates**, which you can also download from the [ACT Resources](#) page.
-

Project Wizard (Workbench Project Tab)*

The extension **ProjectWizard** contains a project wizard named **wizardDemos** that runs from the Workbench Project tab.

XML Extension Definition File

Definitions follow for codes blocks in the XML extension definition file **WizardDemos.xml**.

Extension Definition

The **<extension>** block is the top-most level in the hierarchy.

- The attributes **name** and **version** specify the extension name and version.

- The attribute **icon** specifies the image file to display as the icon for the extension. You specify the location of the **images** folder for all images used by the extension in the **<interface>** block.

Guid Definition

The **<guid>** block specifies a unique identifier for the extension.

Script Definition

The **<script>** blocks specify the IronPython scripts that are referenced by the extension. This extension references four scripts: `main.py`, `ds.py`, `dm.py`, and `sc.py`. The callbacks for steps in the wizard **ProjectWizard** reference the script `main.py`.

Wizard Interface Definition

The **<interface>** blocks define user interfaces for various wizards in the extension. The **WizardDemos** extension has two interface blocks.

- The first **<interface>** block has the attribute **context** set to **Project | Mechanical | SpaceClaim**, which indicates that wizards with their context set to any of these ANSYS products use the interface defined by this block. The wizard **ProjectWizard** wizard uses this first **<interface>** block because its attribute **context** is set to **Project**.
- The second **<interface>** block has the attribute **context** set to **DesignModeler**, which indicates that wizards with their contexts set to this ANSYS product use the interface defined by this block.

The attribute **images** specifies the **images** folder for all images that the extension is to use. You use the attribute **icon** for the extension, wizard, step, and so on to specify an image file in this folder to display as the icon.

Wizard Layout Definition

The **<uidefinition>** block defines a set of layouts for the user interfaces of the wizards. For the sake of this example, the contents of this block have been removed from the excerpt of the XML extension definition file that is shown after the code block definitions. For more information, see [Customizing the Layout of a Wizard \(p. 162\)](#) and the example [Wizard Custom Layout Example \(p. 391\)](#).

Wizard Definition

The wizard **ProjectWizard** and its steps are defined in the **<wizard>** block named **ProjectWizard**.

- The attributes **name** and **version** specify the wizard name and version.
- The attribute **context** specifies the ANSYS product in which the wizard is executed. Because this wizard runs from the Workbench Project tab, **context** is set to **Project**.
- The attribute **icon** specifies the filename of the image to display as the icon for the wizard: `wizard_icon`. The file `wizard_icon.png` is stored in the **images** folder specified for the attribute **images** in the **<interface>** block.
- The attribute **description** specifies the text to display in the **About** window for the extension when it is accessed from the **Extension Manager** and in the **About** window for the wizard when it is accessed from the **Wizards** page.

Step Definition

The **<step>** blocks define the steps of the wizard. There are six steps: **Geometry**, **Mechanical**, **Fluent**, **ReportView**, **CustomStep**, and **Charts**.

For each step:

- The attributes **name**, **version**, and **caption** specify the name, version, and display text for the step.
- The attribute **context** specifies the ANSYS product in which the specific step is executed. You only need to set the context for the step if it differs from the context for the wizard.
- The attribute **HelpFile** specifies the HTML file to display as the custom help for the step.
- The **<callbacks>** block specifies callbacks to functions defined in the script `main.py`.
 - For the step **Geometry**, the callback **<onupdate>** executes the function **EmptyAction**. The callback **<onreset>** executes the function **DeleteGeometry**. The callback **<oninit>** executes the function **InitTabularData**.
 - For the step **Mechanical**, the callback **<onupdate>** executes the function **EmptyAction**. The callback **<onreset>** executes the function **DeleteMechanical**.
 - For the step **Fluent**, the callback **<onrefresh>** executes the function **CreateDialog**. The callback **<onupdate>** executes the function **EmptyAction**. The callback **<onreset>** executes the function **EmptyReset**.
 - For the step **ReportView**, the callback **<onrefresh>** executes the function **RefreshReport**. The callback **<onreset>** executes the function **EmptyReset**.
 - For the step **CustomStep**, the callback **<onrefresh>** executes the function **RefreshMechanical**. The callback **<onupdate>** executes the function **LogReport**. The callback **<onreset>** executes the function **EmptyReset**.
 - For the step **Charts**, the callback **<onrefresh>** executes the function **RefreshCharts**.
- Any **<propertygroup>** blocks specify groupings of properties within the step. Any **<property>** blocks specify properties and property attributes. For properties requiring validation, the callback **<onvalidate>** executes the appropriate validation function.

An excerpt from the file `WizardDemos.xml` follows.

```
<extension version="2" minorversion="1" name="WizardDemos">
<guid shortid="WizardDemos">7fdb141e-3383-433a-a5af-32cb19971771</guid>
<author>ANSYS Inc.</author>
<description>Simple extension to test wizards in different contexts.</description>

<script src="main.py" />
<script src="ds.py" />
<script src="dm.py" />
<script src="sc.py" />

<interface context="Project|Mechanical|SpaceClaim">Caption property
<images>images</images>
</interface>

<interface context="DesignModeler">
<images>images</images>

<toolbar name="Deck" caption="Deck">
<entry name="Deck" icon="deck">
<callbacks>
<onclick>CreateDeck</onclick>
</callbacks>
</entry>
<entry name="Support" icon="Support">
<callbacks>
<onclick>CreateSupport</onclick>
</callbacks>
</entry>
</toolbar>
</interface>
```

```

</entry>
</toolbar>

</interface>

...

<wizard name="ProjectWizard" version="1" context="Project" icon="wizard_icon">
  <description>Simple wizard for demonstration in Project page.</description>

  <step name="Geometry" caption="Geometry" version="1" HelpFile="help/geometry.html"
    layout="DefaultTabularDataLayout">
    <description>Create a geometry component.</description>

    <componentStyle component="Submit">
      <background-color>#b6d7a8</background-color>
    </componentStyle>

    <componentData component="TabularData">
      <CanAddDelete>false</CanAddDelete>
    </componentData>

    <callbacks>
      <onupdate>EmptyAction</onupdate>
      <onreset>DeleteGeometry</onreset>
      <oninit>InitTabularData</oninit>
    </callbacks>

    <propertygroup display="caption" name="definition" caption="Basic properties" >
      <property name="filename" caption="Geometry file name" control="fileopen" />
      <property name="myint" caption="Integer value" control="integer" />
      <property name="mytext" caption="Text value" control="text" />
      <property name="myquantity" caption="Quantity value" control="float" unit="Pressure" />
      <property name="myreadonly" caption=" Readonly value" control="text" readonly="true"
      default="My value" />
      <propertygroup display="property" name="myselect" caption="List of choice" control="select"
      default="Option1">
        <attributes options="Option1,Option2" />
        <property name="option1" caption="Option1 value" control="text" visibleon="Option1" />
        <property name="option2first" caption="Option2 first value" control="float" unit="Pressure"
        visibleon="Option2" />
        <property name="option2second" caption="Option2 second value" control="float" unit="Length"
        visibleon="Option2" />
      </propertygroup>
    </propertygroup>

    <propertytable name="Table" caption="Table" control="custom" display="worksheet"
      class="Worksheet.TabularDataEditor.TabularDataEditor">
      <property name="Frequency" caption="Frequency" unit="Frequency" control="float"
      isparameter="true"></property>
      <property name="Damping" caption="Damping" control="float" isparameter="true"></property>
      <property name="TestFileopen" caption="fileopen" control="fileopen"></property>
    </propertytable>

    <propertytable name="TableB" caption="Table B" control="tabulardata" display="worksheet">
      <property name="Integer" caption="Integer" control="integer"></property>
      <property name="FileOpen" caption="Fileopen" control="fileopen"></property>
      <property name="Number" caption="Number A" unit="Pressure" control="float"></property>
      <property name="ReadOnly" caption="ReadOnly" unit="Pressure" control="float" readonly="true"
      default="4 [Pa]" />
      <propertygroup display="property" name="myselect" caption="List of choice" control="select"
      default="Option1">
        <attributes options="Option1,Option2" />
        <property name="TestStr" caption="Text" control="text" visibleon="Option1"></property>
        <property name="Number B" caption="Number B" unit="Temperature" control="float"></property>
      </propertygroup>
    </propertytable>

  </step>

  <step name="Mechanical" caption="Mechanical" enabled="true" version="1" HelpFile="help/mechanical.html">
    <description>Create a mechanical component.</description>

```

```
<callbacks>
<onupdate>EmptyAction</onupdate>
<onreset>DeleteMechanical</onreset>
</callbacks>

<property name="name" caption="System name" control="text" />

</step>

<step name="Fluent" caption="Fluent" version="1" HelpFile="help/fluent.html">
<description>Create a fluent component.</description>

<callbacks>
<onrefresh>CreateDialog</onrefresh>
<onupdate>EmptyAction</onupdate>
<onreset>EmptyReset</onreset>
</callbacks>

<property name="name" caption="System name" control="text" />
<property name="dialog" caption="Dialog" control="text">
<callbacks>
<onvalidate>ValidateDialog</onvalidate>
</callbacks>
</property>
<property name="dialog2" caption="DialogProgress" control="text">
<callbacks>
<onvalidate>ValidateDialogProgress</onvalidate>
</callbacks>
</property>
<property name="nextstep" caption="Next Step" control="select" >
<callbacks>
<onvalidate>ValidateNextStep</onvalidate>
</callbacks>
</property>

</step>

<step name="ReportView" caption="ReportView" version="1" layout="ReportView@WizardDemos">
<description>Simple example to demonstrate how report can be displayed.</description>

<callbacks>
<onrefresh>RefreshReport</onrefresh>
<onreset>EmptyReset</onreset>
</callbacks>

</step>

<step name="CustomStep" caption="CustomStep" enabled="true" version="1" layout="MyLayout@WizardDemos">
<description>Create a mechanical component.</description>

<callbacks>
<onrefresh>RefreshMechanical</onrefresh>
<onupdate>LogReport</onupdate>
<onreset>EmptyReset</onreset>
</callbacks>

<property name="name" caption="System name" control="text" />

</step>

<step name="Charts" caption="Charts" enabled="true" version="1" layout="GraphLayout@WizardDemos">
<description>Demonstrate all chart capabilities.</description>

<callbacks>
<onrefresh>RefreshCharts</onrefresh>
</callbacks>

</step>

</wizard>
```

```
...
```

```
</extension>
```

IronPython Script

The IronPython script `main.py` follows. It defines all of the functions executed by callbacks in the extension's XML file. Each step defined in the XML file can include multiple actions.

```
geoSystem = None
dsSystem = None
fluentSystem = None

def EmptyAction(step):
    pass

def InitTabularData(step):
    table = step.Properties["TableB"]
    for i in range(1, 10):
        table.AddRow()
        table.Properties["Integer"].Value = i
        table.Properties["FileOpen"].Value = "super"
        table.Properties["Number"].Value = 45.21
        table.Properties["ReadOnly"].Value = 777
        table.Properties["myselect"].Properties["TestStr"].Value = 777
        table.Properties["myselect"].Properties["Number B"].Value = 777
    table.SaveActiveRow()

def CreateGeometry(step):
    global geoSystem
    template1 = GetTemplate(TemplateName="Geometry")
    geoSystem = template1.CreateSystem()
    geometry1 = geoSystem.GetContainer(ComponentName="Geometry")
    geometry1.SetFile(FilePath=step.Properties["definition/filename"].Value)

def DeleteGeometry(step):
    global geoSystem
    geoSystem.Delete()

def RefreshMechanical(step):
    tree = step.UserInterface.GetComponent("Tree")
    root = tree.CreateTreeNode("Root")
    node1 = tree.CreateTreeNode("Node1")
    node2 = tree.CreateTreeNode("Node2")
    node3 = tree.CreateTreeNode("Node3")
    root.Values.Add(node1)
    root.Values.Add(node2)
    node2.Values.Add(node1)
    node2.Values.Add(node3)
    root.Values.Add(node3)
    tree.SetTreeRoot(root)
    chart = step.UserInterface.GetComponent("Chart")
    chart.Plot([1,2,3,4,5],[10,4,12,13,8],"b","Line1")
    chart.Plot([1,2,3,4,5],[5,12,7,8,11],"r","Line2")

def CreateMechanical(step):
    global dsSystem, geoSystem
    template2 = GetTemplate(
        TemplateName="Static Structural",
        Solver="ANSYS")
    geometryComponent1 = geoSystem.GetComponent(Name="Geometry")
    dsSystem = template2.CreateSystem(
        ComponentsToShare=[geometryComponent1],
        Position="Right",
        RelativeTo=geoSystem)
    if step.Properties["name"].Value=="error":
        raise UserErrorMessageException("Invalid system name. Please try again.")
    dsSystem.DisplayText = step.Properties["name"].Value

def DeleteMechanical(step):
```

```
global dsSystem
dsSystem.Delete()

def CreateFluent(step):
    global dsSystem, fluentSystem
    template3 = GetTemplate(TemplateName="Fluid Flow")
    geometryComponent2 = dsSystem.GetComponent(Name="Geometry")
    solutionComponent1 = dsSystem.GetComponent(Name="Solution")
    componentTemplate1 = GetComponentTemplate(Name="CFDPostTemplate")
    fluentSystem = template3.CreateSystem(
        ComponentsToShare=[geometryComponent2],
        DataTransferFrom=[Set(FromComponent=solutionComponent1, TransferName=None,
        ToComponentTemplate=componentTemplate1)],
        Position="Right",
        RelativeTo=dsSystem)
    if step.Properties["name"].Value=="error":
        raise Exception("Invalid system name. Please try again.")
    fluentSystem.DisplayText = step.Properties["name"].Value

def CreateDialog(step):
    comp = step.UserInterface.Panel.CreateOKCancelDialog("MyDialog", "MyTitle", 400, 150)
    comp.SetOkButton("Ok")
    comp.SetMessage("My own message")
    comp.SetCallback(cbDialog)
    prop = step.Properties["nextstep"]
    prop.Options.Clear()
    s = step.NextStep
    val = s.Caption
    while s!=None:
        prop.Options.Add(s.Caption)
        s = s.NextStep
    prop.Value = val

def cbDialog(sender, args):
    dialog = CurrentWizard.CurrentStep.UserInterface.GetComponent("MyDialog").Hide()

def ValidateDialog(step, prop):
    dialog = step.UserInterface.GetComponent("MyDialog")
    dialog.Show()

def worker(step):
    progressDialog = step.UserInterface.GetComponent("Progress")
    progress = progressDialog.GetFirstOrDefaultComponent()
    progress.Reset()
    stopped = progress.UpdateProgress("Start progress...", 0, True)
    progressDialog.Show()
    for i in range(100):
        System.Threading.Thread.Sleep(100)
        stopped = progress.UpdateProgress("Start progress...", i+1, True)
        if stopped:
            break
    progressDialog.Hide()

def ValidateDialogProgress(step, prop):
    thread = System.Threading.Thread(System.Threading.ParameterizedThreadStart(worker))
    thread.Start(step)

def ValidateNextStep(step, prop):
    prop = step.Properties["nextstep"]
    s = step.NextStep
    v = False
    while s!=None:
        if prop.Value==s.Caption:
            v = True
        s.IsEnabled = v
        s = s.NextStep
    steps = step.UserInterface.GetComponent("Steps")
    steps.UpdateData()
    steps.Refresh()

def RefreshReport(step):
    report = step.UserInterface.GetComponent("Report")
```

```

report.SetHtmlContent(System.IO.Path.Combine(ExtAPI.Extension.InstallDir,"help","report.html"))
report.Refresh()

def EmptyReset(step):
    pass

def LogReport(step):
    ExtAPI.Log.WriteMessage("Report:")
    for s in step.Wizard.Steps.Values:
        ExtAPI.Log.WriteMessage("Step "+s.Caption)
        for prop in s.AllProperties:
            ExtAPI.Log.WriteMessage(prop.Caption+": "+prop.DisplayString)

import random
import math

def RefreshCharts(step):
    graph = step.UserInterface.GetComponent("Graph")
    graph.Title("Line Bar Graph")
    graph.ShowLegend(False)
    graph.Plot([-1, 0, 1, 2, 3, 4], [0.5, -0.5, 0.5, -0.5, 0.5, 0.5], key="Variable A", color='g')
    graph.Bar([-1, 0, 1, 2, 3, 4], [10, 20, 30, 10, 5, 20], key="Variable B")

    graphB = step.UserInterface.GetComponent("GraphB")
    graphB.Title("Plot Graph")
    graphB.YTickFormat("0.2f")

    xValues = []
    yValues = []
    for i in range(0, 100):
        xValues.append(i)
        yValues.append(abs(math.sin(i*0.2))*i/100.0)
    graphB.Plot(xValues, yValues, key="y = a*sin(bx)", color="c")
    graphB.Plot(xValues, yValues, key="y = x", color="m")

    xValues = []
    yValues = []
    for i in range(0, 100):
        xValues.append(i)
        yValues.append(i/100.0)
    graphB.Plot(xValues, yValues, key="y = x", color="m")

    graphB.Plot([0, 10, 20, 30, 100], [0.2, 0.2, 0.2, 0.3, 0.3], key="Smth", color="r")
    graphB.Plot([0, 10, 20, 30, 100], [0.2, 0.1, 0.3, 0.5, 0.7], key="Smth", color="r")
    graphB.Plot([0, 10, 20, 30, 100], [0.2, 0.2, 0.2, 0.3, 0.3])

    graphC = step.UserInterface.GetComponent("GraphC")
    graphC.Title("Pie Graph")
    graphC.Pie([1, 2, 3])
    graphC.Pie([20, 30, 5, 15, 12], [0, "Banana", 2, 3, "42"])

    graphD = step.UserInterface.GetComponent("GraphD")
    graphD.Title("Bar Graph")
    graphD.Bar(["Banana"], [70], key="key")
    graphD.Bar([0, "Banana", 2, 3, 4], [20, 30, 5, 15, 12], key="key")

    graphE = step.UserInterface.GetComponent("GraphE")
    graphE.Title("Bubble Graph")
    graphE.XTickFormat("f")
    graphE.YTickFormat("f")
    keys = ["one", "two", "three", "four", "five"]
    colors = ["#BB3333", "#33BB33", "#3333BB", "#BBBB33", "#BB33BB"]
    for c in range(0, 5):
        xValues = []
        yValues = []
        sizeValues = []
        for i in range(0, (c+1)*20):
            rad = random.randrange(c+1, c+2) + (random.random()*2-1)
            angle = random.random() * 2 * math.pi
            xValues.append(math.cos(angle) * rad)
            yValues.append(math.sin(angle) * rad)
            sizeValues.append(random.random() * 2.0 + 0.5)

```

```

graphE.Bubble(xValues, yValues, sizeValues, key=keys[c], color=colors[c])

def CreateStaticStructural(step):
    template1 = GetTemplate(
        TemplateName="Static Structural",
        Solver="ANSYS")
    system1 = template1.CreateSystem()
    system1.DisplayText = "toto"

    nextStep = step.NextStep
    if nextStep!=None:
        nextStep.SystemName = system1.Name
        nextStep.ComponentName = "Geometry"
    nextStep = nextStep.NextStep
    if nextStep!=None:
        nextStep.SystemName = system1.Name
        nextStep.ComponentName = "Geometry"
    nextStep = nextStep.NextStep
    if nextStep!=None:
        nextStep.SystemName = system1.Name
        nextStep.ComponentName = "Geometry"
    nextStep = nextStep.NextStep
    if nextStep!=None:
        nextStep.SystemName = system1.Name
        nextStep.ComponentName = "Model"
    nextStep = nextStep.NextStep
    if nextStep!=None:
        nextStep.SystemName = system1.Name
        nextStep.ComponentName = "Model"

def OnSelectContext(step, prop):
    firstDMStep = step.NextStep
    secondDMStep = firstDMStep.NextStep
    firstSCStep = secondDMStep.NextStep
    secondSCStep = firstSCStep.NextStep

    firstGeoStep = step.NextStep
    if prop.Value == "DesignModeler":
        firstDMStep.IsEnabled = True
        secondDMStep.IsEnabled = True
        firstSCStep.IsEnabled = False
        secondSCStep.IsEnabled = False
    elif prop.Value == "SpaceClaim":
        firstDMStep.IsEnabled = False
        secondDMStep.IsEnabled = False
        firstSCStep.IsEnabled = True
        secondSCStep.IsEnabled = True

    panel = step.UserInterface.Panel.GetComponent("Steps")
    panel.UpdateData()
    panel.Refresh()

def RefreshResultsProject(step):
    step.Properties["Res"].Value = ExtAPI.Extension.Attributes["result"]
    panel = step.UserInterface.Panel.GetComponent("Properties")
    panel.UpdateData()
    panel.Refresh()

```

Project Wizard (AIM Project Tab)

The extension **PressureLoss** contains a project wizard with the same name that is run from the AIM Project tab. This example calculates the maximum velocity and pressure loss for a CFD system with one inlet and three outlets.

Note

- The XML syntax used to define a project wizard run in AIM is the same as that used to define a project wizard run in Workbench.
- No **<uidefinition>** block is defined for an AIM wizard (custom template). The layout for AIM is predefined.

XML Extension Definition File

Definitions follow for codes blocks in the XML extension definition file **PressureLoss.xml**.

Extension Definition

The **<extension>** block is the top-most level in the hierarchy.

- The attributes **name** and **version** specify the extension name and version.
- The attribute **icon** specifies the image file to display as the icon for the extension. You specify the location of the **images** folder for all images used by the extension in the **<interface>** block.

Guid Definition

The **<guid>** block specifies a unique identifier for the extension.

Script Definition

The **<script>** block specifies the IronPython script that is referenced by the extension. This extension references **pressureLoss.py**.

Wizard Interface Definition

The **<interface>** block defines the user interface for the wizard.

- The attribute **context** is set to **Project** because the extension is executed on the AIM Project tab.
- The attribute **images** specifies the **images** folder for all images that the extension is to use. You use the attribute **icon** for the extension, wizard, step, and so on to specify an image file in this folder to display as the icon.

Wizard Definition

The wizard **PressureLoss** and its steps are defined in the **<wizard>** block named **PressureLoss**.

- The attributes **name** and **version** specify the wizard name and version.
- The attribute **context** specifies the ANSYS product in which the wizard is executed. Because this wizard runs from the AIM Project tab, **context** is set to **Project**.
- The attribute **icon** specifies the filename of the image to display as the icon for the wizard: **images\loss.png**. The file **loss.png** is stored in the **images** folder specified for the attribute **images** in the **<interface>** block.

- The attribute **description** specifies the text to display in the **About** window for the extension when it is accessed from the **Extension Manager** and in the **About** window for the wizard when it is accessed from the **Wizards** page.

Step Definition

The **<step>** blocks are used to define the steps of the wizard. There are four steps: **Step1**, **Step2**, **Step3**, and **Step4**.

For each step:

- The attributes **name**, **version**, and **caption** specify the name, version, and display text for the step.
- The attribute **context** specifies the context in which the specific step is executed. You only need to set the context for the step if it differs from the context for the wizard.
- The attribute **HelpFile** specifies the HTML file to display as the custom help for the step. In this example, the help provided for each step indicates where each property is displayed in the **Study** panel.
- The **<callbacks>** blocks specify callbacks to functions defined in the IronPython script.
 - For step **Step1**, the callback **<onupdate>** executes the function **importGeometry**. This step creates the CFD workflow, importing and then generating the specified geometry when the **Next** button is clicked.
 - For step **Step2**, the callback **<onupdate>** executes the function **refineMesh**. This step allows the user to specify the mesh resolution and then refines the mesh accordingly when the **Next** button is clicked.
 - For step **Step3**, the callback **<onrefresh>** executes the function **initLocations**. The callback **<onupdate>** executes the function **setup**. This step enables the user to specify the loads applied to the inlet and outlet, selecting a location and specifying a load (**Velocity** for the **Inlet** and **Gauge Pressure** for the **Outlet**).

The callback **<isvalid>** on the property **OutletLocation** executes the action **isValid**, validating the value selected for the outlet location.

- For step **Step4**, no callbacks are required. This step takes the value calculated for the property **CalculatedMaximum** and populates it to the **Calculated maximum** field of the **Velocity** object.
- Any **<propertygroup>** blocks specify groupings of properties within the step. Any **<property>** blocks specify properties and property attributes.

The XML extension definition file **PressureLoss.xml** follows.

```
<extension version="1" name="PressureLoss" icon="images\loss.png">
<guid shortid="PressureLoss">74182c0a-16ea-4169-8a55-00bad98afb6c</guid>
<author>ANSYS Inc.</author>
<description>Demonstration of a pressure loss in AIM.</description>

<script src="pressureLoss.py" />

<interface context="Project">
  <images>images</images>
</interface>

<wizard name="PressureLoss" version="1" context="Project" icon="loss">
  <description>This wizard is for demonstration of ACT wizard capability in AIM.</description>
  <step name="Step1" caption="Import the geometry" version="1" context="Project" HelpFile="help/geometry.html">
```

```

<description>Import the geometry file and create the workflow.</description>

<callbacks>
  <onupdate>importGeometry</onupdate>
</callbacks>

<property name="geometryfile" caption="geometry file" control="fileopen"
default="E:\Geometry\TubeSelectionSet.agdb"/>
</step>

<step name="Step2" caption="Refine the mesh" version="1" context="Project" HelpFile="help/mesh.html">
  <description>Refine the mesh from Low to High.</description>

  <callbacks>
    <onupdate>refineMesh</onupdate>
  </callbacks>

  <property name="MeshResolution" caption="mesh resolution" control="integer" default="1"/>
</step>

<step name="Step3" caption="Define the loads" version="1" context="Project" HelpFile="help/loads.html">
  <description>Specify the loads to applied on the geometry.</description>

  <callbacks>
    <onrefresh>initLocations</onrefresh>
    <onupdate>setup</onupdate>
  </callbacks>

  <propertygroup name="Inlet" caption="Inlet">
    <property name="Velocity" caption="Velocity" control="float" unit="Velocity" default="0.1 [m s^-1]"/>
    <property name="InletLocation" caption="InletLocation" control="select" />
  </propertygroup>

  <propertygroup name="Outlet" caption="Outlet">
    <property name="GaugePressure" caption="Gauge Pressure" control="float" unit="Pressure"
default="0 [Pa]"/>
    <property name="OutletLocation" caption="OutletLocation" control="select" >
      <callbacks>
        <isvalid>isValid</isvalid>
      </callbacks>
    </property>
  </propertygroup>
</step>

<step name="Step4" caption="Export the maximum velocity" version="1" context="Project"
HelpFile="help/result.html">
  <description>Here we are just exposing the value of the maximum velocity and the pressure
loss.</description>

  <property name="MaximumVelocity" caption="Maximum Velocity" control="float" unit="Velocity"
readonly = "True"/>
  <property name="PressureLoss" caption="Pressure Loss" control="float" unit="Pressure" readonly = "True"/>
</step>
</wizard>

</extension>

```

IronPython Script

The IronPython script `pressureLoss.py` follows. This script defines all functions executed by the callbacks in the XML extension definition file. Each step defined in the XML file can include multiple actions.

```

meshingComponent1 = None
study1 = None
physicsDefinitionComponent1 = None
resultsEvaluationComponent1 = None
solvePhysicsComponent1 = None
physicsRegion1 = None
vectorResult1 = None

```

```

singleValueResult1 = None
results1 = None
materialAssignment1 = None
currentStep = None
clr.AddReference("Ans.UI")

def getSelectionSetsForProject():
    context = __scriptingEngine__.CommandContext
    project = context.Project
    containers = project.GetContainers()
    dataEntity = "SelectionSet"

    for container in containers:
        if container.Name == "Study":
            try:
                lockObject = context.ContainerReadLock(container)
                dataReferences = project.GetDataReferencesByType(container, dataEntity)
            finally:
                lockObject.Dispose()
            break
    return dataReferences

def initLocations(step):
    list = getSelectionSetsForProject()
    propIn = step.Properties["Inlet/InletLocation"]
    propIn.Options.Clear()
    propOut = step.Properties["Outlet/OutletLocation"]
    propOut.Options.Clear()
    for sel in list:
        ExtAPI.Log.WriteMessage("OPTION: "+sel.DisplayText)
        propIn.Options.Add(sel.DisplayText)
        propOut.Options.Add(sel.DisplayText)
    comp = step.UserInterface.GetComponent("Properties")
    comp.UpdateData()
    comp.Refresh()

def isValid(step, property):
    if property.Value == step.Properties["Inlet/InletLocation"].Value:
        ExtAPI.Log.LogWarning("Inlet and Outlet locations must be different.")
        return False
    return True

def importGeometry(step):
    global meshingComponent1, study1, results1, vectorResult1, singleValueResult1, physicsDefinitionComponent1,
    resultsEvaluationComponent1, solvePhysicsComponent1, physicsRegion1, materialAssignment1
    with Transaction():
        system1 = GetSystem(Name="Study")
        physicsDefinitionComponent1 = Study.CreateTask( Type="Physics Definition", System=system1)
        study1 = system1.GetContainer(ComponentName="Study")
        physicsDefinition1 = physicsDefinitionComponent1.GetTaskObject()
        physicsRegion1 = study1.CreateEntity( Type="PhysicsRegion", Association=physicsDefinition1)
        solverSettings1 = study1.CreateEntity( Type="SolverSettings", Association=physicsDefinition1)
        solvePhysicsComponent1 = Study.CreateTask( Type="Solve Physics", System=system1,
Input=physicsDefinitionComponent1)
        solvePhysicsComponent1.Refresh()
        resultsEvaluationComponent1 = Study.CreateTask( Type="Results Evaluation", System=system1,
Input=solvePhysicsComponent1)
        resultsEvaluationComponent1.Refresh()
        physicsDefinition1.CalculationType = "Static"
        physicsRegion1.PhysicsType = "Fluid"
        physicsRegion1.Location = "AllBodies()"
        materialAssignment1 = study1.CreateEntity( Type="MaterialAssignment", Association=physicsDefinition1)
        material1 = study1.CreateEntity( Type="Material", Association=physicsDefinition1)
        material1.ImportEngineeringData(Name="Air")
        materialAssignment1.Material = material1

        materialAssignment1.Location = [physicsRegion1]
        results1 = resultsEvaluationComponent1.GetTaskObject()
        vectorResult1 = study1.CreateEntity( Type="VectorResult", Association=results1)
        vectorResult1.Variable = "Velocity"
        vectorResult1.DisplayText = "Velocity"
        transcript1 = study1.CreateEntity( Type="Transcript", Association=physicsDefinition1)

```

```

transcript1.DisplayText = "Fluid Flow Output 1"
physicsSolutionGroup1 = Study.CreateGroup(Name="Physics Solution")
physicsSolutionGroup1.Add(Component=physicsDefinitionComponent1)
physicsSolutionGroup1.Add(Component=solvePhysicsComponent1)
meshingComponent1 = Study.CreateTask( Type="Meshing", System=system1, Output=physicsDefinitionComponent1)
physicsDefinitionComponent1.Refresh()
solvePhysicsComponent1.Refresh()
resultsEvaluationComponent1.Refresh()
importComponent1 = Study.CreateTask( Type="Import", System=system1, Output=meshingComponent1)
meshingComponent1.Refresh()
physicsDefinitionComponent1.Refresh()
solvePhysicsComponent1.Refresh()
resultsEvaluationComponent1.Refresh()
import1 = importComponent1.GetTaskObject()
geometryImportSource1 = import1.AddGeometryImportSourceOperation()
geometryImportSource1.FilePath = step.Properties["geometryfile"].Value
importComponent1.Update(AllDependencies=True)
meshingComponent1.Refresh()
physicsDefinitionComponent1.Refresh()
solvePhysicsComponent1.Refresh()
resultsEvaluationComponent1.Refresh()

def refineMesh(step):
    global meshingComponent1, study1, results1, vectorResult1, physicsDefinitionComponent1,
singleValueResult1, resultsEvaluationComponent1, solvePhysicsComponent1, physicsRegion1
    meshing1 = meshingComponent1.GetTaskObject()
    meshing1.MeshResolution = step.Properties["MeshResolution"].Value

def setup(step):
    global meshingComponent1, study1, results1, vectorResult1, physicsDefinitionComponent1, singleValueResult1,
resultsEvaluationComponent1, solvePhysicsComponent1, physicsRegion1
    with Transaction():
        meshing1 = meshingComponent1.GetTaskObject()
        meshControlLocalInflation1 = study1.CreateEntity( Type="MeshControlLocalInflation", Association=meshing1)
        meshing1.EngineeringIntent = "FluidFlow"
        AddSourceToComponentInSystem( SourceComponent=physicsDefinitionComponent1,
TargetComponent=resultsEvaluationComponent1)
        resultsEvaluationComponent1.Refresh()
    #meshControlLocalInflation1.Location = ["FACE8", "FACE2", "FACE6"]
    Study.Delete(Items=[meshControlLocalInflation1])
    with Transaction():
        meshingComponent1.Update(AllDependencies=True)
        physicsDefinitionComponent1.Refresh()
        solvePhysicsComponent1.Refresh()
        resultsEvaluationComponent1.Refresh()

    with Transaction():
        inletBoundary1 = study1.CreateEntity( Type="InletBoundary", Association=physicsRegion1)

        inlet_location = step.Properties["Inlet/InletLocation"].Value
        ExtAPI.Log.WriteMessage("my inlet location property value is : " + inlet_location)
        inlet_selection_set = None
        selection_sets = getSelectionSetsForProject()
        for selection_set in selection_sets:
            if selection_set.DisplayText == inlet_location:
                inlet_selection_set = selection_set
        if inlet_selection_set == None :
            ExtAPI.Log.WriteMessage("inlet selection set does not exist")

        inletBoundary1.Location = [inlet_selection_set]

        inletBoundary1.Flow.Velocity.Magnitude = step.Properties["Inlet/Velocity"].DisplayString

        outletBoundary1 = study1.CreateEntity( Type="OutletBoundary", Association=physicsRegion1)

        outlet_location = step.Properties["Outlet/OutletLocation"].Value

        outlet_selection_set = None
        selection_sets = getSelectionSetsForProject()
        for selection_set in selection_sets:
            if selection_set.DisplayText == outlet_location:
                outlet_selection_set = selection_set

```

```
if outlet_selection_set == None :
    ExtAPI.Log.WriteMessage("outlets selection set does not exist")

outletBoundary1.Location = [outlet_selection_set]
outletBoundary1.Flow.Pressure.GaugeStaticPressure = step.Properties["Outlet/GaugePressure"].DisplayString

wallBoundary1 = study1.CreateEntity( Type="WallBoundary", Association=physicsRegion1)

# Creation of the pressure loss expression.
singleValueResult1 = study1.CreateEntity( Type="SingleValueResult", Association=results1)
singleValueResult1.Method = "UserDefinedExpressionMethod"
singleValueResult1.Expression = "Average(Pressure, GetBoundary('@Inlet 1'), Weight='Area') - Average(Pressure, GetBoundary('@Outlet 1'), Weight='Area')"

with Transaction():
    physicsDefinitionComponent1.Update(AllDependencies=True)
    solvePhysicsComponent1.Update(AllDependencies=True)
    resultsEvaluationComponent1.Refresh()
    resultsEvaluationComponent1.Refresh()
    resultsEvaluationComponent1.Update(AllDependencies=True)

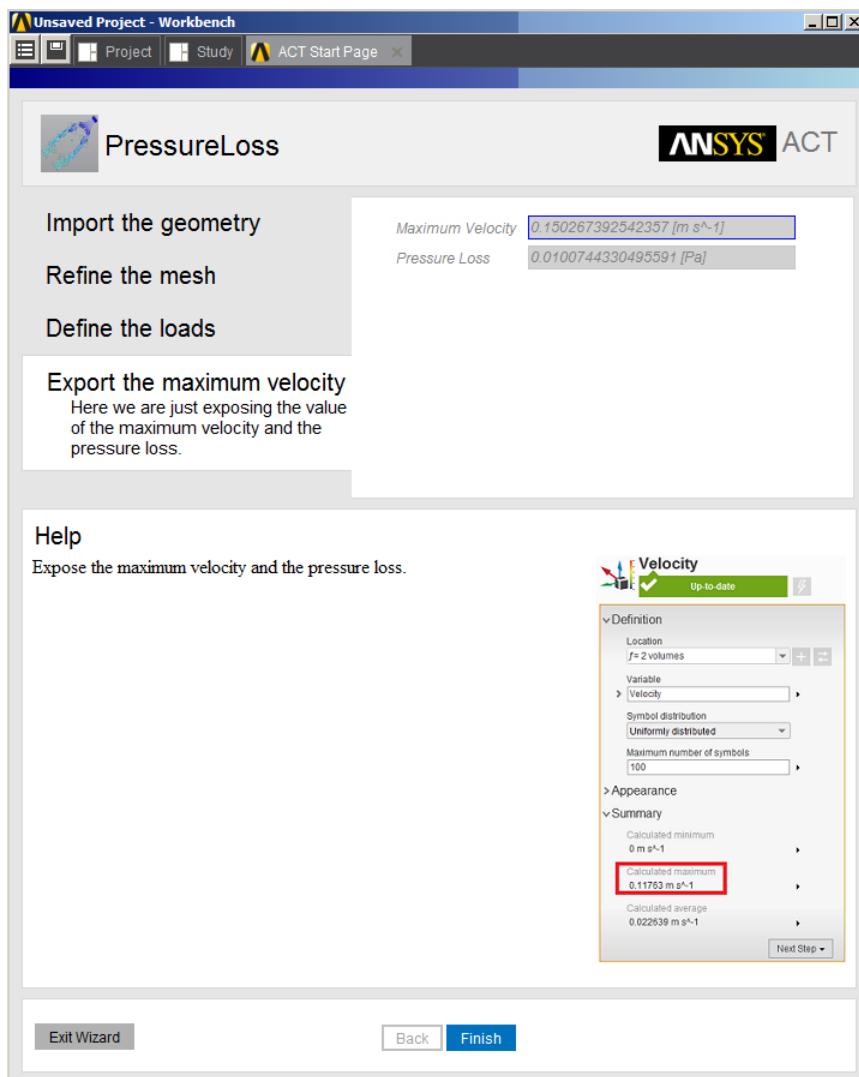
vectorResult1.Legend.Coloring = "Banded"
vectorResult1.Legend.NumberOfColors = "10"
vectorResult1.Distribution = "Mesh"
vectorResult1.Evaluate()
maximum_velocity = vectorResult1.Summary.Max
step.NextStep.Properties["MaximumVelocity"].Value = maximum_velocity.ToString()
step.NextStep.Properties["PressureLoss"].Value = singleValueResult1.Value.ToString()

def solve(step):
    global meshingComponent1, study1, results1, vectorResult1, physicsDefinitionComponent1, singleValueResult1,
    resultsEvaluationComponent1, solvePhysicsComponent1, physicsRegion1
    with Transaction():
        physicsDefinitionComponent1.Update(AllDependencies=True)
        solvePhysicsComponent1.Update(AllDependencies=True)
        resultsEvaluationComponent1.Refresh()
        resultsEvaluationComponent1.Refresh()
        resultsEvaluationComponent1.Update(AllDependencies=True)

    vectorResult1.Legend.Coloring = "Banded"
    vectorResult1.Legend.NumberOfColors = "10"
    vectorResult1.Distribution = "Mesh"
    vectorResult1.Evaluate()
    maximum_velocity = vectorResult1.Summary.Max
    step.NextStep.Properties["MaximumVelocity"].Value = maximum_velocity.ToString()
    step.NextStep.Properties["PressureLoss"].Value = singleValueResult1.Value.ToString()
```

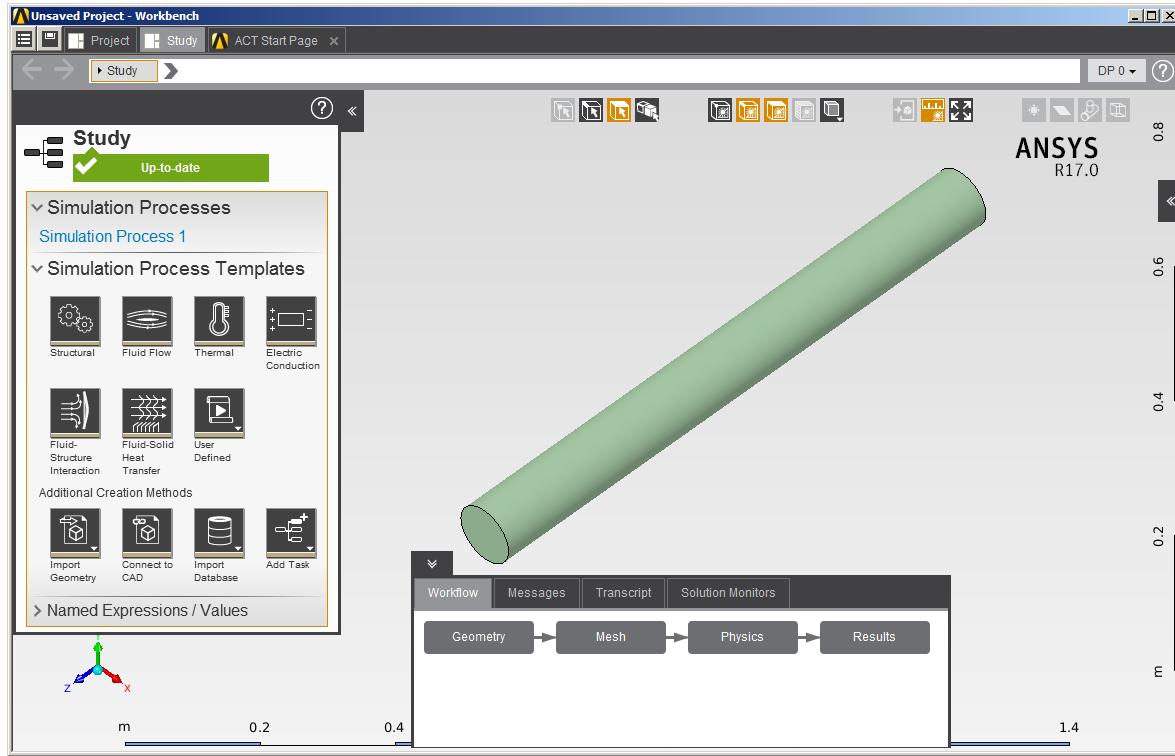
Reviewing Wizard Results

When the wizard has completed, the final page shows the calculated **Maximum Velocity**. The **Help** panel indicates where the result is shown in the AIM study.

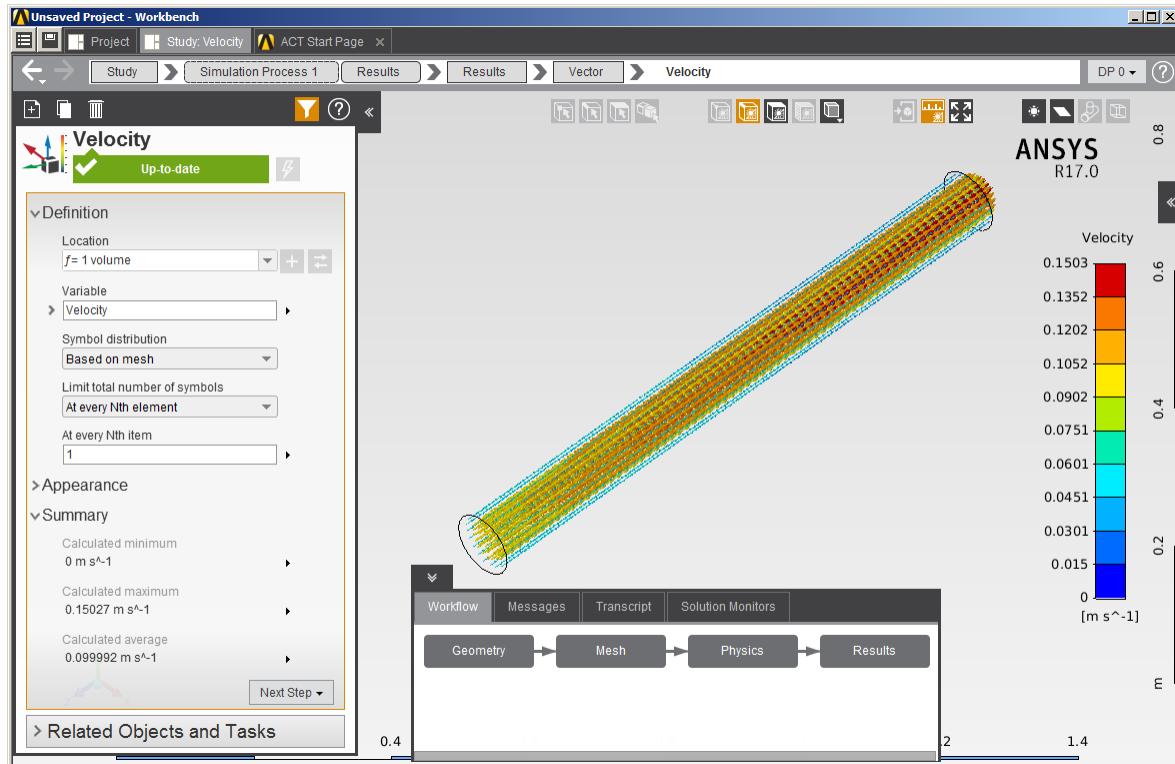


Clicking the **Finish** button returns the user to the **ACT Start Page**.

The user can then open the **Results** task to view the final results obtained by the wizard-driven simulation.



On the **Study** panel, the user can see that the wizard created the workflow in AIM and then automatically executed the steps.



DesignModeler Wizard*

The extension **WizardDemos** contains a target product wizard for DesignModeler named **CreateBridge**. This two-step wizard is for building a bridge.

Note

The extension **WizardDemos** contains two wizards named **CreateBridge**. The first one is for DesignModeler, and the second one is for SpaceClaim. This topic describes the wizard for DesignModeler. The next topic describes the wizard for SpaceClaim.

XML Extension Definition File

Definitions follow for codes blocks in the XML extension definition file `WizardDemos.xml`.

Extension Definition

The `<extension>` block is the top-most level in the hierarchy.

- The attributes `name` and `version` specify the extension name and version.
- The attribute `icon` specifies the image file to display as the icon for the extension. You specify the location of the `images` folder for all images used by the extension in the `<interface>` block.

Guid Definition

The `<guid>` block specifies a unique identifier for the extension.

Script Definition

The `<script>` blocks specify the IronPython scripts that are referenced by the extension. This extension references four scripts: `main.py`, `ds.py`, `dm.py`, and `sc.py`. The callbacks for steps in the wizard **CreateBridge** for DesignModeler reference the script `dm.py`.

Wizard Interface Definition

The `<interface>` blocks define user interfaces for various wizards in the extension. The extension **WizardDemos** has two interface blocks.

- The first `<interface>` block has the attribute `context` set to **Project | Mechanical | SpaceClaim**, which indicates that wizards with their contexts set to any of these ANSYS products use the interface defined by this block.
- The second `<interface>` block has the attribute `context` set to **DesignModeler**, which indicates that wizards with their context set to this ANSYS product use the interface defined by this block. The wizard **CreateBridge** for DesignModeler uses this `<interface>` block. It has a `<toolbar>` block that defines two toolbar buttons for exposure in DesignModeler. When the buttons are clicked, the callback `<onclick>` executes the functions `CreateDeck` and `CreateSupport`, creating a deck geometry with supports.

The attribute `images` specifies the `images` folder for all images that the extension is to use. You use the attribute `icon` for the extension, wizard, step, and so on to specify an image file in this folder to display as the icon.

Simdata Definition

The two `<simdata>` blocks provide data for the creation of the geometries **Deck** and **Support**.

Wizard Definition

The wizard **CreateBridge** and its steps are defined in the **<wizard>** block named **CreateBridge**. The first wizard **CreateBridge** is for DesignModeler.

- The attributes **name** and **version** specify the wizard name and version.
- The attribute **context** specifies the ANSYS product in which the wizard is executed. Because this wizard runs from DesignModeler, **context** is set to **DesignModeler**.
- The attribute **icon** specifies the filename of the image to display as the icon for the wizard: **wizard_icon**. The file **wizard_icon.png** is stored in the **images** folder specified for the attribute **images** in the **<interface>** block.
- The attribute **description** specifies the text to display in the **About** window for the extension when it is accessed from the **Extension Manager** and in the **About** window for the wizard when it is accessed from the **Wizards** page.

Step Definition

The **<step>** blocks define the steps of the wizard. This wizard has two steps: **Deck** and **Supports**.

For each step:

- The attributes **name**, **version**, and **caption** specify the name, version, and display text for the step.
- The attribute **context** specifies the ANSYS product in which the specific step is executed. You only need to set the context for the step if it differs from the context for the wizard.
- The attribute **HelpFile** specifies the HTML file to display as the custom help for the step.
- The **<callbacks>** block specifies callbacks to functions defined in the script **dm.py**.
 - For the step **Deck**, the callback **<onupdate>** executes the function **UpdateDeck**, creating the deck using the geometry **Deck**.
 - For the step **Supports**, the callback **<onupdate>** executes the function **UpdateSupports**, creating the bridge supports using the geometry **Support**.
- Any **<propertygroup>** blocks specify groupings of properties within the step. Any **<property>** blocks specify properties and property attributes.

An excerpt from the file **WizardDemos.xml** follows.

```
<extension version="2" minorversion="1" name="WizardDemos">
<guid shortid="WizardDemos">7fdb141e-3383-433a-a5af-32cb19971771</guid>
<author>ANSYS Inc.</author>
<description>Simple extension to test wizards in different contexts.</description>

<script src="main.py" />
<script src="ds.py" />
<script src="dm.py" />
<script src="sc.py" />

<interface context="Project|Mechanical|SpaceClaim">
<images>images</images>
</interface>

<interface context="DesignModeler">
<images>images</images>

<toolbar name="Deck" caption="Deck">
```

```

<entry name="Deck" icon="deck">
  <callbacks>
    <onclick>CreateDeck</onclick>
  </callbacks>
</entry>
<entry name="Support" icon="Support">
  <callbacks>
    <onclick>CreateSupport</onclick>
  </callbacks>
</entry>
</toolbar>

</interface>

<simdata context="DesignModeler">
  <geometry name="Deck" caption="Deck" icon="deck" version="1">
    <callbacks>
      <ongenerate>GenerateDeck</ongenerate>
    </callbacks>
    <property name="Length" caption="Length" control="float" unit="Length" default="300 [m]" />
    <property name="Width" caption="Width" control="float" unit="Length" default="20 [m]" />
    <property name="Beams" caption="Beams" control="integer" default="31" />
  </geometry>
</simdata>

<simdata context="DesignModeler">
  <geometry name="Support" caption="Support" icon="support" version="1">
    <callbacks>
      <ongenerate>GenerateSupport</ongenerate>
    </callbacks>
    <property name="Length" caption="Length" control="float" unit="Length" default="300 [m]" />
    <property name="Height" caption="Height" control="float" unit="Length" default="100 [m]" />
    <property name="Width" caption="Width" control="float" unit="Length" default="20 [m]" />
    <property name="Number" caption="Number" control="integer" default="3" />
  </geometry>
</simdata>

  ...
<wizard name="CreateBridge" version="1" context="DesignModeler" icon="wizard_icon">
  <description>Simple wizard for demonstration in DesignModeler.</description>

  <step name="Deck" caption="Deck" version="1" HelpFile="help/dm1.html">
    <description>Create the deck.</description>

    <callbacks>
      <onupdate>UpdateDeck</onupdate>
    </callbacks>

    <propertygroup display="caption" name="Deck" caption="Deck Definition" >
      <property name="Length" caption="Length" control="float" unit="Length" default="300 [m]" />
      <property name="Width" caption="Width" control="float" unit="Length" default="20 [m]" />
      <property name="Beams" caption="Beams" control="integer" default="31" />
    </propertygroup>
  </step>

  <step name="Supports" caption="Supports" enabled="true" version="1" HelpFile="help/dm2.html">
    <description>Create supports.</description>

    <callbacks>
      <onupdate>UpdateSupports</onupdate>
    </callbacks>

    <propertygroup display="caption" name="Supports" caption="Supports Definition" >
      <property name="Height" caption="Height" control="float" unit="Length" default="100 [m]" />
      <property name="Number" caption="Number" control="integer" default="3" />
    </propertygroup>
  </step>
</wizard>
```

...

```
</extension>
```

IronPython Script

The IronPython script dm.py follows. This script defines all functions executed by the callbacks in the XML extension definition file. Each step defined in the XML file can include multiple actions.

```
import units

def CreateDeck(ag):
    ExtAPI.CreateFeature( "Deck" )

def CreateSupport(ag):
    ExtAPI.CreateFeature( "Support" )

def GenerateDeck(feature,fct):
    length = feature.Properties["Length"].Value
    length = units.ConvertUnit(length, ExtAPI.DataModel.CurrentUnitFromQuantityName("Length"), "m")
    width = feature.Properties["Width"].Value
    width = units.ConvertUnit(width, ExtAPI.DataModel.CurrentUnitFromQuantityName("Length"), "m")
    num = feature.Properties["Beams"].Value

    builder = ExtAPI.DataModel.GeometryBuilder
    bodies = []

    boxGen = builder.Primitives.Solid.CreateBox([0.,-width/2.,-0.3],[length,width/2.,0.])
    bodies.Add(boxGen.Generate())

    w = (length-0.1*num)/(num-1.)+0.1
    for i in range(num-1):
        beamGen = builder.Primitives.Solid.CreateBox([i*w,-width/2.,-0.6],[i*w+0.1,width/2.,-0.3])
        bodies.Add(beamGen.Generate())

    beamGen = builder.Primitives.Solid.CreateBox([length-0.1,-width/2.,-0.6],[length,width/2.,-0.3])
    bodies.Add(beamGen.Generate())

    beamGen = builder.Primitives.Solid.CreateBox([0.,-width/2.,-1.],[length,-width/2.+0.2,-0.6])
    bodies.Add(beamGen.Generate())
    beamGen = builder.Primitives.Solid.CreateBox([0.,width/2.-0.2,-1.],[length,width/2.,-0.6])
    bodies.Add(beamGen.Generate())

    feature.Bodies = bodies
    feature.MaterialType = MaterialTypeEnum.Add

    return True

def GenerateSupport(feature,fct):
    length = feature.Properties["Length"].Value
    length = units.ConvertUnit(length, ExtAPI.DataModel.CurrentUnitFromQuantityName("Length"), "m")
    height = feature.Properties["Height"].Value
    height = units.ConvertUnit(height, ExtAPI.DataModel.CurrentUnitFromQuantityName("Length"), "m")
    width = feature.Properties["Width"].Value
    width = units.ConvertUnit(width, ExtAPI.DataModel.CurrentUnitFromQuantityName("Length"), "m")
    num = feature.Properties["Number"].Value

    builder = ExtAPI.DataModel.GeometryBuilder
    bodies = []

    w = (length-2.*num)/(num+1.)+2.
    for i in range(num):
        beamGen = builder.Primitives.Solid.CreateBox([(i+1)*w,-width/2.,-1.-height],
        [(i+1)*w+2.,width/2.,-1.])
        bodies.Add(beamGen.Generate())

    beamGen = builder.Primitives.Solid.CreateBox([0.,-width/2.,-5.],[2.,width/2.,-1.])
    bodies.Add(beamGen.Generate())
```

```

beamGen = builder.Primitives.Solid.CreateBox([length-2.,-width/2.,-5.],[length,width/2.,-1.])
bodies.Add(beamGen.Generate())

feature.Bodies = bodies
feature.MaterialType = MaterialTypeEnum.Freeze

return True

def UpdateDeck(step):
    deck = ExtAPI.CreateFeature("Deck")
    deck.Properties["Length"].Value = step.Properties["Deck/Length"].Value
    deck.Properties["Width"].Value = step.Properties["Deck/Width"].Value
    deck.Properties["Beams"].Value = step.Properties["Deck/Beams"].Value
    ExtAPI.DataModel.FeatureManager.Generate()

def UpdateSupports(step):
    supports = ExtAPI.CreateFeature("Support")
    supports.Properties["Length"].Value = step.PreviousStep.Properties["Deck/Length"].Value
    supports.Properties["Width"].Value = step.PreviousStep.Properties["Deck/Width"].Value+6
    supports.Properties["Height"].Value = step.Properties["Supports/Height"].Value
    supports.Properties["Number"].Value = step.Properties["Supports/Number"].Value
    ExtAPI.DataModel.FeatureManager.Generate()

```

SpaceClaim Wizard*

The extension **WizardDemos** contains a target product wizard for SpaceClaim named **CreateBridge**. This two-step wizard is for building a bridge

Note

The **WizardDemos** extension contains two wizards named **CreateBridge**. The first one is for DesignModeler, and the second one is for SpaceClaim. This topic describes the wizard for SpaceClaim. The previous topic describes the wizard for DesignModeler.

XML Extension Definition File

Definitions follow for codes blocks in the XML extension definition file `WizardDemos.xml`.

Extension Definition

The `<extension>` block is the top-most level in the hierarchy.

- The attributes `name` and `version` specify the extension name and version.
- The attribute `icon` specifies the image file to display as the icon for the extension. You specify the location of the `images` folder for all images used by the extension in the `<interface>` block.

Guid Definition

The `<guid>` block specifies a unique identifier for the extension.

Script Definition

The `<script>` blocks specify the IronPython scripts that are referenced by the extension. This extension references four scripts: `main.py`, `ds.py`, `dm.py`, and `sc.py`. The callbacks for steps in the wizard `CreateBridge` for SpaceClaim reference the script `sc.py`.

Wizard Interface Definition

The `<interface>` blocks define user interfaces for various wizards in the extension. The extension **WizardDemos** has two interface blocks.

- The first `<interface>` block has the attribute `context` set to `Project | Mechanical | SpaceClaim`, which indicates that wizards with their contexts set to any of these ANSYS products use the interface defined by this block. The wizard `CreateBridge` for SpaceClaim uses this `<interface>` block.
- The second `<interface>` block has the attribute `context` set to `DesignModeler`, which indicates that wizards with their contexts set to this ANSYS product use the interface defined by this block.

The attribute `images` specifies the `images` folder for all images that the extension is to use. You use the attribute `icon` for the extension, wizard, step, and so on to specify an image file in this folder to display as the icon.

Simdata Definition

The two `<simdata>` blocks provide data for the creation of the geometries `Deck` and `Support`.

Wizard Definition

The wizard `CreateBridge` and its steps are defined in the `<wizard>` block named `CreateBridge`. The second wizard `CreateBridge` is for SpaceClaim.

- The attributes `name` and `version` specify the wizard name and version.
- The attribute `context` specifies the ANSYS product in which the wizard is executed. Because this wizard runs from SpaceClaim, `context` is set to `SpaceClaim`.
- The attribute `icon` specifies the filename of the image to display as the icon for the wizard: `wizard_icon`. The file `wizard_icon.png` is stored in the `images` folder specified for the attribute `images` in the `<interface>` block.
- The attribute `description` specifies the text to display in the **About** window for the extension when it is accessed from the **Extension Manager** and in the **About** window for the wizard when it is accessed from the **Wizards** page.

Step Definition

The `<step>` blocks define the steps of the wizard. This wizard has two steps: `DeckSC` and `SupportSSC`.

For each step:

- The attributes `name`, `version`, and `caption` specify the name, version, and display text for the step.
- The attribute `context` specifies the ANSYS product in which the specific step is executed. You only need to set the context for the step if it differs from the context for the wizard.
- The attribute `HelpFile` specifies the HTML file to display as the custom help for the step.
- The `<callbacks>` block specifies callbacks to functions defined in the script `sc.py`.
 - For the step `DeckSC`, the callback `<onupdate>` executes the function `UpdateDeckSC`, creating the deck using the geometry `Deck`.
 - For the step `SupportSSC`, the callback `<onupdate>` executes the function `UpdateSupportSSC`, creating the bridge supports using the geometry `Support`.

- Any **<propertygroup>** blocks specify groupings of properties within the step. Any **<property>** blocks specify properties and property attributes.

Note

Beginning in release 18.2, the graphics window in SpaceClaim is not updated until the callbacks for a step have been executed. This change ensures graphics stability and better performance.

An excerpt from the file WizardDemos.xml follows.

```

<extension version="2" minorversion="1" name="WizardDemos">
  <guid shortid="WizardDemos">7fdb141e-3383-433a-a5af-32cb19971771</guid>
  <author>ANSYS Inc.</author>
  <description>Simple extension to test wizards in different contexts.</description>

  <script src="main.py" />
  <script src="ds.py" />
  <script src="dm.py" />
  <script src="sc.py" />

  <interface context="Project|Mechanical|SpaceClaim">
    <images>images</images>
  </interface>

  <interface context="DesignModeler">
    <images>images</images>

    <toolbar name="Deck" caption="Deck">
      <entry name="Deck" icon="deck">
        <callbacks>
          <onclick>CreateDeck</onclick>
        </callbacks>
      </entry>
      <entry name="Support" icon="Support">
        <callbacks>
          <onclick>CreateSupport</onclick>
        </callbacks>
      </entry>
    </toolbar>
  </interface>

  <simdata context="DesignModeler">
    <geometry name="Deck" caption="Deck" icon="deck" version="1">
      <callbacks>
        <ongenerate>GenerateDeck</ongenerate>
      </callbacks>
      <property name="Length" caption="Length" control="float" unit="Length" default="300 [m]" />
      <property name="Width" caption="Width" control="float" unit="Length" default="20 [m]" />
      <property name="Beams" caption="Beams" control="integer" default="31" />
    </geometry>
  </simdata>

  <simdata context="DesignModeler">
    <geometry name="Support" caption="Support" icon="support" version="1">
      <callbacks>
        <ongenerate>GenerateSupport</ongenerate>
      </callbacks>
      <property name="Length" caption="Length" control="float" unit="Length" default="300 [m]" />
      <property name="Height" caption="Height" control="float" unit="Length" default="100 [m]" />
      <property name="Width" caption="Width" control="float" unit="Length" default="20 [m]" />
      <property name="Number" caption="Number" control="integer" default="3" />
    </geometry>
  </simdata>
  ...

```

```

<wizard name="CreateBridge" version="1" context="SpaceClaim" icon="wizard_icon">
  <description>Simple wizard for demonstration in SpaceClaim.</description>

  <step name="DeckSC" caption="DeckSC" version="1" context="SpaceClaim">
    <description>Create the deck.</description>

    <callbacks>
      <onupdate>UpdateDeckSC</onupdate>
    </callbacks>

    <propertygroup display="caption" name="Deck" caption="Deck Definition" >
      <property name="Length" caption="Length" control="float" unit="Length" default="300 [m]" />
      <property name="Width" caption="Width" control="float" unit="Length" default="20 [m]" />
      <property name="Beams" caption="Beams" control="integer" default="31" />
    </propertygroup>
  </step>

  <step name="SupportsSC" caption="SupportsSC" context="SpaceClaim" enabled="true" version="1">
    <description>Create supports.</description>

    <callbacks>
      <onupdate>UpdateSupportsSC</onupdate>
    </callbacks>

    <propertygroup display="caption" name="Supports" caption="Supports Definition" >
      <property name="Height" caption="Height" control="float" unit="Length" default="100 [m]" />
      <property name="Number" caption="Number" control="integer" default="3" />
    </propertygroup>
  </step>
</wizard>

...
</extension>

```

IronPython Script

The IronPython script sc.py follows. This script defines all functions executed by the callbacks in the XML extension definition file. Each step defined in the XML file can include multiple actions.

```

import units

def createBox(xa, ya, za, xb, yb, zb):
  win = Window.ActiveWindow
  context = win.ActiveContext
  part = context.ActivePart

  lengthX = xb - xa
  lengthY = yb - ya
  lengthZ = zb - za
  xa = xa + lengthX * 0.5
  ya = ya + lengthY * 0.5

  p = Geometry.PointUV.Create(0, 0)
  body = Modeler.Body.ExtrudeProfile(Geometry.RectangleProfile(Geometry.Plane.PlaneXY, lengthX,
lengthY, p, 0), lengthZ)
  designBody = DesignBody.Create(part, "body", body)

  translation = Geometry.Matrix.CreateTranslation(Geometry.Vector.Create(xa, ya, za))
  designBody.Transform(translation)

def UpdateDeckSC(step):
  length = step.Properties["Deck/Length"].Value
  width = step.Properties["Deck/Width"].Value
  num = step.Properties["Deck/Beams"].Value

  createBox(0., -width/2., -0.3, length,width/2., 0.)

  w = (length-0.1*num)/(num-1.)+0.1
  for i in range(num-1):

```

```

createBox(i*w,-width/2.,-0.6, i*w+0.1,width/2.,-0.3)

createBox(length-0.1, -width/2., -0.6, length, width/2., -0.3)

createBox(0., -width/2., -1., length, -width/2.+0.2, -0.6)
createBox(0., width/2.-0.2, -1., length,width/2., -0.6)

return True

def UpdateSupportsSC(step):
    length = step.PreviousStep.Properties["Deck/Length"].Value
    width = step.PreviousStep.Properties["Deck/Width"].Value
    height = step.Properties["Supports/Height"].Value
    num = step.Properties["Supports/Number"].Value

    w = (length-2.*num)/(num+1.)+2.
    for i in range(num):
        createBox((i+1)*w, -width/2., -1.-height, (i+1)*w+2., width/2.,-1.)

    beamGen = createBox(0., -width/2., -5., 2., width/2., -1.)
    beamGen = createBox(length-2., -width/2., -5., length,width/2., -1.)

return True

```

Mechanical Wizard*

The extension **WizardDemos** contains a target product wizard for Mechanical named **SimpleAnalysis**. This wizard performs a simple analysis on the bridge built by either the DesignModeler or SpaceClaim wizard **CreateBridge**.

XML Extension Definition File

Definitions follow for codes blocks in the XML extension definition file **WizardDemos.xml**.

Extension Definition

The **<extension>** block is the top-most level in the hierarchy.

- The attributes **name** and **version** specify the extension name and version.
- The attribute **icon** specifies the image file to display as the icon for the extension. You specify the location of the **images** folder for all images used by the extension in the **<interface>** block.

Guid Definition

The **<guid>** block specifies a unique identifier for the extension.

Script Definition

The **<script>** blocks specify the IronPython scripts that are referenced by the extension. This extension references four scripts: **main.py**, **ds.py**, **dm.py**, and **sc.py**. The callbacks for steps in the wizard **SimpleAnalysis** reference the script **ds.py**.

Wizard Interface Definition

The **<interface>** blocks define user interfaces for various wizards in the extension. The extension **WizardDemos** has two interface blocks.

- The first **<interface>** block has the attribute **context** set to **Project | Mechanical | SpaceClaim**, which indicates that wizards with their contexts set to any of these ANSYS products use the interface defined by this block. The wizard **SimpleAnalysis** uses this first **<interface>** block because its attribute **context** is set to **Mechanical**.

- The second `<interface>` block has the attribute `context` set to `DesignModeler`, which indicates that wizards with their contexts set to this ANSYS product use the interface defined by this block.

The attribute `images` specifies the `images` folder for all images that the extension is to use. You use the attribute `icon` for the extension, wizard, step, and so on to specify an image file in this folder to display as the icon.

Wizard Definition

The wizard `SimpleAnalysis` and its steps are defined in the `<wizard>` block named `SimpleAnalysis`.

- The attributes `name` and `version` specify the wizard name and version.
- The attribute `context` specifies the ANSYS product in which the wizard is executed. Because this wizard runs from Mechanical, `context` is set to `Mechanical`.
- The attribute `icon` specifies the filename of the image to display as the icon for the wizard: `wizard_icon.png`. The file `wizard_icon.png` is stored in the `images` folder specified for the attribute `images` in the `<interface>` block.
- The attribute `description` specifies the text to display in the **About** window for the extension when it is accessed from the **Extension Manager** and in the **About** window for the wizard when it is accessed from the **Wizards** page.

Step Definition

The `<step>` blocks define the steps of the wizard. There are three steps: `Mesh`, `Solution`, and `Results`.

For each step:

- The attributes `name`, `version`, and `caption` specify the name, version, and display text for the step.
- The attribute `context` specifies the ANSYS product in which the specific step is executed. You only need to set the context for the step if it differs from the context for the wizard.
- The attribute `HelpFile` specifies the HTML file to display as the custom help for the step.
- The `<callbacks>` block specifies callbacks to functions defined in the script `ds.py`.
 - For the step `Mesh`, the callback `<onreset>` executes the function `RemoveControls` to clear existing mesh controls. The callback `<onupdate>` executes the function `CreateMeshControls` to create new mesh controls.
 - For the step `Solution`:
 - The callback `<onrefresh>` executes the function `RefreshLoads` to initialize various properties, including the number of nodes, number of elements computed in the previous step, and so on.
 - The callback `<onreset>` executes the function `RemoveLoads` to clear loads.
 - The callback `<onupdate>` executes the function `CreateLoads` to create new loads, create new results, and perform the solve.
 - For the step `Results`, the callback `<onrefresh>` executes the function `RefreshResults` to fill the property value associated with the result of the computation (Maximum of Total Deformation).

- Any **<propertygroup>** blocks specify groupings of properties within the step. Any **<property>** blocks specify properties and property attributes. For properties requiring location selection, the callback **<isvalid>** executes the function **IsLocationValid** to validate the selection. A custom message can be shown when the entered value fails validation.

An excerpt from the file `WizardDemos.xml` follows.

```

<extension version="2" minorversion="1" name="WizardDemos">
  <guid shortid="WizardDemos">7fdb141e-3383-433a-a5af-32cb19971771</guid>
  <author>ANSYS Inc.</author>
  <description>Simple extension to test wizards in different contexts.</description>

  <script src="main.py" />
  <script src="ds.py" />
  <script src="dm.py" />
  <script src="sc.py" />

  <interface context="Project|Mechanical|SpaceClaim">
    <images>images</images>
  </interface>

  <interface context="DesignModeler">
    <images>images</images>

    <toolbar name="Deck" caption="Deck">
      <entry name="Deck" icon="deck">
        <callbacks>
          <onclick>CreateDeck</onclick>
        </callbacks>
      </entry>
      <entry name="Support" icon="Support">
        <callbacks>
          <onclick>CreateSupport</onclick>
        </callbacks>
      </entry>
    </toolbar>
  </interface>

  ...
  ...

  <wizard name="SimpleAnalysis" version="1" context="Mechanical" icon="wizard_icon">
    <description>Simple wizard to illustrate how to setup, solve and analyse results of a simulation process.</description>

    <step name="Mesh" caption="Mesh" version="1" HelpFile="help/ds1.html">
      <description>Setup some mesh controls.</description>

      <callbacks>
        <onreset>RemoveControls</onreset>
        <onupdate>CreateMeshControls</onupdate>
      </callbacks>

      <propertygroup display="caption" name="Sizing" caption="Mesh Sizing" >
        <property name="Location" caption="Edge Location" control="geometry_selection">
          <attributes selection_filter="edge" />
          <callbacks>
            <isvalid>IsLocationValid</isvalid>
          </callbacks>
        </property>
        <property name="Ndiv" caption="Divisions" control="integer" />
      </propertygroup>

    </step>

    <step name="Solution" caption="Solution" version="1" HelpFile="help/ds2.html">
      <description>Setup loads.</description>

      <callbacks>
        <onrefresh>RefreshLoads</onrefresh>
      </callbacks>
    </step>
  </wizard>

```

```
<onreset>RemoveLoads</onreset>
<onupdate>CreateLoads</onupdate>
</callbacks>

<propertygroup display="caption" name="Mesh" caption="Mesh Statistics" >
  <property name="Nodes" caption="Nodes" control="text" readonly="true" />
  <property name="Elements" caption="Elements" control="text" readonly="true" />
</propertygroup>
<propertygroup display="caption" name="FixedSupport" caption="Fixed Support" >
  <property name="Location" caption="Face Location" control="geometry_selection">
    <attributes selection_filter="face" />
  <callbacks>
    <isvalid>IsLocationFSValid</isvalid>
  </callbacks>
  </property>
</propertygroup>

</step>

<step name="Results" caption="Results" version="1" HelpFile="help/ds3.html">
  <description>View Results.</description>

  <callbacks>
    <onrefresh>RefreshResults</onrefresh>
  </callbacks>

  <property name="Res" caption="Deformation" control="text" readonly="true" />
</step>

</wizard>

...
</extension>
```

IronPython Script

The IronPython script ds.py follows. This script defines all functions executed by the callbacks in the XML extension definition file. Each step defined in the XML file can include multiple actions.

```
ef IsLocationValid(step, prop):
    if prop.Value==None:
        return False
    if prop.Value.Ids.Count!=1:
        prop.StateMessage = "Select only one edge."
        return False
    return True

def CreateMeshControls(step):
    model = ExtAPI.DataModel.Project.Model
    mesh = model.Mesh
    sizing = mesh.AddSizing()
    sel = step.Properties["Sizing/Location"].Value
    entity = ExtAPI.DataModel.GeoEntityById(sel.Ids[0])
    len = entity.Length
    ids = []
    for part in ExtAPI.DataModel.GeoData.Assemblies[0].Parts:
        for body in part.Bodies:
            for edge in body.Edges:
                if abs(edge.Length-len)/len<1.e-6:
                    ids.Add(edge.Id)
    sel = ExtAPI.SelectionManager.CreateSelectionInfo(SelectionTypeEnum.GeometryEntities)
    sel.Ids = ids
    sizing.Location = sel
    sizing.Type = SizingType.NumberOfDivisions
    sizing.NumberOfDivisions = step.Properties["Sizing/Ndiv"].Value
    step.Attributes.SetValue("sizing", sizing)
    mesh.GenerateMesh()

def RemoveControls(step):
```

```

sizing = step.Attributes["sizing"]
sizing.Delete()

def IsLocationFSValid(step, prop):
    if prop.Value==None:
        return False
    if prop.Value.Ids.Count!=1:
        prop.StateMessage = "Select only one face."
        return False
    return True

def RefreshLoads(step):
    model = ExtAPI.DataModel.Project.Model
    step.Properties["Mesh/Nodes"].Value = model.Mesh.Nodes.ToString()
    step.Properties["Mesh/Elements"].Value = model.Mesh.Elements.ToString()
    panel = step.UserInterface.GetComponent("Properties")
    panel.UpdateData()
    panel.Refresh()

def CreateLoads(step):
    model = ExtAPI.DataModel.Project.Model
    analysis = model.Analyses[0]
    support = analysis.AddFixedSupport()
    sel = step.Properties["FixedSupport/Location"].Value
    entity = ExtAPI.DataModel.GeoData.GeoEntityById(sel.Ids[0])
    area = entity.Area
    ids = []
    for part in ExtAPI.DataModel.GeoData.Assemblies[0].Parts:
        for body in part.Bodies:
            for face in body.Faces:
                if abs(face.Area-area)/area<1.e-6:
                    ids.Add(face.Id)
    sel = ExtAPI.SelectionManager.CreateSelectionInfo(SelectionTypeEnum.GeometryEntities)
    sel.Ids = ids
    support.Location = sel
    loads = []
    loads.Add(support)
    step.Attributes.SetValue("loads", loads)

    loads.Add(analysis.AddEarthGravity())
    res = analysis.Solution.AddTotalDeformation()
    step.Attributes.SetValue("res", res)
    loads.Add(res)
    analysis.Solve(True)
    ExtAPI.Extension.SetAttributeValueWithSync("result", res.Maximum.ToString())

def RemoveLoads(step):
    loads = step.Attributes["loads"]
    for load in loads:
        load.Delete()

def RefreshResults(step):
    model = ExtAPI.DataModel.Project.Model
    res = step.PreviousStep.Attributes["res"]
    step.Properties["Res"].Value = res.Maximum.ToString()
    panel = step.UserInterface.GetComponent("Properties")
    panel.UpdateData()
    panel.Refresh()

```

Mixed Wizard*

The extension **WizardDemos** contains a mixed wizard named **BridgeSimulation**. It executes one step on the Workbench Project tab, reuses either the DesignModeler or SpaceClaim wizard **CreateBridge**, reruns the bridge analysis in the Mechanical wizard **SimpleAnalysis**, and then returns to the Project tab to execute the step **Results**.

XML Extension Definition File

Definitions follow for codes blocks in the XML extension definition file `WizardDemos.xml`. For the sake of this example, the `<uidefinition>` block has been omitted from the XML extension definition file that is shown later in this topic. For information on customizing layouts, see [Customizing the Layout of a Wizard \(p. 162\)](#) and the example [Wizard Custom Layout Example \(p. 391\)](#).

Extension Definition

The `<extension>` block is the top-most level in the hierarchy.

- The attributes `name` and `version` specify the extension name and version.
- The attribute `icon` specifies the image file to display as the icon for the extension. You specify the location of the `images` folder for all images used by the extension in the `<interface>` block.

Guid Definition

The `<guid>` block specifies a unique identifier for the extension.

Script Definition

The `<script>` blocks specify the IronPython scripts that are referenced by the extension. This extension references four scripts: `main.py`, `ds.py`, `dm.py`, and `sc.py`. The callbacks for steps in the wizard `BridgeSimulation` reference all four of these scripts.

Wizard Interface Definition

The `<interface>` blocks define user interfaces for various wizards in the extension. The extension `WizardDemos` has two interface blocks, both of which the wizard `BridgeSimulation` uses.

- The first `<interface>` block has the attribute `context` set to `Project | Mechanical | SpaceClaim`, which indicates that wizards with their contexts set to any of these ANSYS products use the interface defined by this block.
- The second `<interface>` block has the attribute `context` set to `DesignModeler`, which indicates that wizards with their contexts set to this ANSYS product use the interface defined by this block. This `<interface>` block has a `<toolbar>` block that defines two toolbar buttons for exposure in DesignModeler. When the buttons are clicked, the callback `<onclick>` executes the functions `CreateDeck` and `CreateSupport`, creating a deck geometry with supports.

The attribute `images` specifies the `images` folder for all images that the extension is to use. You use the attribute `icon` for the extension, wizard, step, and so on to specify an image file in this folder to display as the icon.

Simdata Definition

The two `<simdata>` blocks provide data for the creation of the geometries `Deck` and `Support` in DesignModeler.

Wizard Definition

The wizard `BridgeSimulation` and its steps are defined in the `<wizard>` block named `BridgeSimulation`.

- The attributes `name` and `version` specify the wizard name and version.
- The attribute `context` specifies the ANSYS product in which the wizard is executed. Because this wizard accesses the target product from the Project tab instead of executing in the target product directly, `context` is set to `Project`.

- The attribute **icon** specifies the filename of the image to display as the icon for the wizard: bridge. The file bridge.png is stored in the images folder specified for the attribute **images** in the **<interface>** block.
- The attribute **description** specifies the text to display in the **About** window for the extension when it is accessed from the **Extension Manager** and in the **About** window for the wizard when it is accessed from the **Wizards** page.

Step Definition

The **<step>** blocks define the steps of the wizard. There are eight steps: **Project**, **DeckDM**, **SupportsDM**, **DeckSC**, **SupportsSC**, **Mesh**, **Solution**, and **Results**. The steps with **DM** in their names are for execution in DesignModeler. The steps with **SC** in their names are for execution in SpaceClaim.

For each step:

- The attributes **name**, **version**, and **caption** specify the name, version, and display text for the step.
- The attribute **context** specifies the ANSYS product in which the specific step is executed. You only need to set the context for the step if it differs from the context for the wizard.
- The attribute **HelpFile** specifies the HTML file to display as the custom help for the step.
- The **<callbacks>** block specifies callbacks to functions defined in all four IronPython scripts.
 - For the step **Project**, the callback **<onupdate>** executes the function **CreateStaticStructural** in the script **main.py**.
 - For the step **DeckDM**, the callback **<onupdate>** executes the function **UpdateDeck** in the script **dm.py**.
 - For the step **SupportsDM**, the callback **<onupdate>** executes the function **UpdateSupports** in the script **dm.py**.
 - For the step **DeckSC**, the callback **<onupdate>** executes the function **UpdateDeckSC** in the script **sc.py**.
 - For the step **SupportsSC**, the callback **<onupdate>** executes the function **UpdateSupportsSC** in the script **sc.py**.
 - For the step **Mesh**, the callbacks **<onreset>** and **<onupdate>** execute the functions **RemoveControls** and **CreateMeshControls** in the script **ds.py**.
 - For the step **Solution**, the callback **<onrefresh>**, **<onreset>**, and **<onupdate>** execute the functions **RefreshLoads**, **RemoveLoads**, and **CreateLoads** in the script **ds.py**.
 - For the step **Results**, the callback **<onrefresh>** executes the function **RefreshResultsProject** in the script **main.py**.
- Any **<propertygroup>** blocks specify groupings of properties within the step. Any **<property>** blocks specify properties and property attributes.

An excerpt from the file **WizardDemos.xml** follows.

```
<extension version="2" minorversion="1" name="WizardDemos">
<guid shortid="WizardDemos">7fdb141e-3383-433a-a5af-32cb19971771</guid>
<author>ANSYS Inc.</author>
```

```
<description>Simple extension to test wizards in different contexts.</description>

<script src="main.py" />
<script src="ds.py" />
<script src="dm.py" />
<script src="sc.py" />

<interface context="Project|Mechanical|SpaceClaim">
  <images>images</images>
</interface>

<interface context="DesignModeler">
  <images>images</images>

  <toolbar name="Deck" caption="Deck">
    <entry name="Deck" icon="deck">
      <callbacks>
        <onclick>CreateDeck</onclick>
      </callbacks>
    </entry>
    <entry name="Support" icon="Support">
      <callbacks>
        <onclick>CreateSupport</onclick>
      </callbacks>
    </entry>
  </toolbar>
</interface>

<simdata context="DesignModeler">
  <geometry name="Deck" caption="Deck" icon="deck" version="1">
    <callbacks>
      <ongenerate>GenerateDeck</ongenerate>
    </callbacks>
    <property name="Length" caption="Length" control="float" unit="Length" default="300 [m]" />
    <property name="Width" caption="Width" control="float" unit="Length" default="20 [m]" />
    <property name="Beams" caption="Beams" control="integer" default="31" />
  </geometry>
</simdata>

<simdata context="DesignModeler">
  <geometry name="Support" caption="Support" icon="support" version="1">
    <callbacks>
      <ongenerate>GenerateSupport</ongenerate>
    </callbacks>
    <property name="Length" caption="Length" control="float" unit="Length" default="300 [m]" />
    <property name="Height" caption="Height" control="float" unit="Length" default="100 [m]" />
    <property name="Width" caption="Width" control="float" unit="Length" default="20 [m]" />
    <property name="Number" caption="Number" control="integer" default="3" />
  </geometry>
</simdata>

...

<wizard name="ProjectWizard" version="1" context="Project" icon="wizard_icon">
  <description>Simple wizard for demonstration in Project page.</description>

  <step name="Geometry" caption="Geometry" version="1" HelpFile="help/geometry.html"
layout="DefaultTabularDataLayout">
    <description>Create a geometry component.</description>

    <componentStyle component="Submit">
      <background-color>#b6d7a8</background-color>
    </componentStyle>

    <componentData component="TabularData">
      <CanAddDelete>false</CanAddDelete>
    </componentData>

    <callbacks>
      <onupdate>EmptyAction</onupdate>
      <onreset>DeleteGeometry</onreset>
    </callbacks>
  </step>
</wizard>
```

```

<oninit>InitTabularData</oninit>
</callbacks>

<propertygroup display="caption" name="definition" caption="Basic properties" >
  <property name="filename" caption="Geometry file name" control="fileopen" />
  <property name="myint" caption="Integer value" control="integer" />
  <property name="mytext" caption="Text value" control="text" />
  <property name="myquantity" caption="Quantity value" control="float" unit="Pressure" />
  <property name="myreadonly" caption="Readonly value" control="text" readonly="true" default="My value" />
  <propertygroup display="property" name="myselect" caption="List of choice" control="select"
default="Option1">
    <attributes options="Option1,Option2" />
    <property name="option1" caption="Option1 value" control="text" visibleon="Option1" />
    <property name="option2first" caption="Option2 first value" control="float" unit="Pressure"
visibleon="Option2" />
    <property name="option2second" caption="Option2 second value" control="float" unit="Length"
visibleon="Option2" />
  </propertygroup>
</propertygroup>

<propertytable name="Table" caption="Table" control="custom" display="worksheet"
class="Worksheet.TabularDataEditor.TabularDataEditor">
  <property name="Frequency" caption="Frequency" unit="Frequency" control="float"
isparameter="true"></property>
  <property name="Damping" caption="Damping" control="float" isparameter="true"></property>
  <property name="TestFileopen" caption="fileopen" control="fileopen"></property>
</propertytable>

<propertytable name="TableB" caption="Table B" control="tabulardata" display="worksheet">
  <property name="Integer" caption="Integer" control="integer"></property>
  <property name="FileOpen" caption="Fileopen" control="fileopen"></property>
  <property name="Number" caption="Number A" unit="Pressure" control="float"></property>
  <property name="ReadOnly" caption="ReadOnly" unit="Pressure" control="float" readonly="true"
default="4 [Pa]" />
  <propertygroup display="property" name="myselect" caption="List of choice" control="select"
default="Option1">
    <attributes options="Option1,Option2" />
    <property name="TestStr" caption="Text" control="text" visibleon="Option1"></property>
    <property name="Number B" caption="Number B" unit="Temperature" control="float"></property>
  </propertygroup>
</propertytable>

</step>

<step name="Mechanical" caption="Mechanical" enabled="true" version="1" HelpFile="help/mechanical.html">
  <description>Create a mechanical component.</description>

  <callbacks>
    <onupdate>EmptyAction</onupdate>
    <onreset>DeleteMechanical</onreset>
  </callbacks>

  <property name="name" caption="System name" control="text" />
</step>

<step name="Fluent" caption="Fluent" version="1" HelpFile="help/fluent.html">
  <description>Create a fluent component.</description>

  <callbacks>
    <onrefresh>CreateDialog</onrefresh>
    <onupdate>EmptyAction</onupdate>
    <onreset>EmptyReset</onreset>
  </callbacks>

  <property name="name" caption="System name" control="text" />
  <property name="dialog" caption="Dialog" control="text">
    <callbacks>
      <onvalidate>ValidateDialog</onvalidate>
    </callbacks>
  </property>

```

```
<property name="dialog2" caption="DialogProgress" control="text">
  <callbacks>
    <onvalidate>ValidateDialogProgress</onvalidate>
  </callbacks>
</property>
<property name="nextstep" caption="Next Step" control="select" >
  <callbacks>
    <onvalidate>ValidateNextStep</onvalidate>
  </callbacks>
</property>

</step>

<step name="ReportView" caption="ReportView" version="1" layout="ReportView@WizardDemos">
  <description>Simple example to demonstrate how report can be displayed.</description>

  <callbacks>
    <onrefresh>RefreshReport</onrefresh>
    <onreset>EmptyReset</onreset>
  </callbacks>

</step>

<step name="CustomStep" caption="CustomStep" enabled="true" version="1" layout="MyLayout@WizardDemos">
  <description>Create a mechanical component.</description>

  <callbacks>
    <onrefresh>RefreshMechanical</onrefresh>
    <onupdate>LogReport</onupdate>
    <onreset>EmptyReset</onreset>
  </callbacks>

  <property name="name" caption="System name" control="text" />

</step>

<step name="Charts" caption="Charts" enabled="true" version="1" layout="GraphLayout@WizardDemos">
  <description>Demonstrate all chart capabilities.</description>

  <callbacks>
    <onrefresh>RefreshCharts</onrefresh>
  </callbacks>

</step>

</wizard>

<wizard name="SCWizard" version="1" context="SpaceClaim" icon="wizard_icon">
  <description>Simple wizard for demonstration in SpaceClaim.</description>

  <step name="Geometry" caption="Geometry" version="1" HelpFile="help/geometry.html">
    <description>Create a geometry component.</description>

    <callbacks>
      <onupdate>EmptyAction</onupdate>
      <onreset>DeleteGeometry</onreset>
    </callbacks>

    <propertygroup display="caption" name="definition" caption="Basic properties" >
      <property name="filename" caption="Geometry file name" control="fileopen" />
      <property name="myint" caption="Integer value" control="integer" />
      <property name="mytext" caption="Text value" control="text" />
      <property name="myquantity" caption="Quantity value" control="float" unit="Pressure" />
      <property name="myreadonly" caption="Readonly value" control="text" readonly="true" default="My value" />
      <propertygroup display="property" name="myselect" caption="List of choice" control="select"
default="Option1">
        <attributes options="Option1,Option2" />
        <property name="option1" caption="Option1 value" control="text"
visibleon="Option1" />
        <property name="option2first" caption="Option2 first value" control="float" unit="Pressure"
visibleon="Option2" />
        <property name="option2second" caption="Option2 second value" control="float" unit="Length"
    
```

```

visibleon="Option2" />
</propertygroup>
</propertygroup>

<propertytable name="Table" caption="Table" control="custom" display="worksheet"
class="Worksheet.TabularDataEditor.TabularDataEditor">
<property name="Frequency" caption="Frequency" unit="Frequency" control="float"
isparameter="true"></property>
<property name="Damping" caption="Damping" control="float" isparameter="true"></property>
<property name="TestFileopen" caption="fileopen" control="fileopen"></property>
</propertytable>

</step>

<step name="Mechanical" caption="Mechanical" enabled="true" version="1" HelpFile="help/mechanical.html">
<description>Create a mechanical component.</description>

<callbacks>
<onupdate>EmptyAction</onupdate>
<onreset>DeleteMechanical</onreset>
</callbacks>

<property name="name" caption="System name" control="text" />

</step>

<step name="Fluent" caption="Fluent" version="1" HelpFile="help/fluent.html">
<description>Create a fluent component.</description>

<callbacks>
<onrefresh>CreateDialog</onrefresh>
<onupdate>EmptyAction</onupdate>
<onreset>EmptyReset</onreset>
</callbacks>

<property name="name" caption="System name" control="text" />
<property name="dialog" caption="Dialog" control="text">
<callbacks>
<onvalidate>ValidateDialog</onvalidate>
</callbacks>
</property>
<property name="dialog2" caption="DialogProgress" control="text">
<callbacks>
<onvalidate>ValidateDialogProgress</onvalidate>
</callbacks>
</property>
<property name="nextstep" caption="Next Step" control="select" >
<callbacks>
<onvalidate>ValidateNextStep</onvalidate>
</callbacks>
</property>

</step>

<step name="ReportView" caption="ReportView" version="1" layout="ReportView@WizardDemos">
<description>Simple example to demonstrate how report can be displayed.</description>

<callbacks>
<onrefresh>RefreshReport</onrefresh>
<onreset>EmptyReset</onreset>
</callbacks>

</step>

<step name="CustomStep" caption="CustomStep" enabled="true" version="1" layout="MyLayout@WizardDemos">
<description>Create a mechanical component.</description>

<callbacks>
<onrefresh>RefreshMechanical</onrefresh>
<onupdate>LogReport</onupdate>
<onreset>EmptyReset</onreset>

```

```
</callbacks>

<property name="name" caption="System name" control="text" />

</step>

<step name="Charts" caption="Charts" enabled="true" version="1" layout="GraphLayout@WizardDemos">
  <description>Demonstrate all chart capabilities.</description>

  <callbacks>
    <onrefresh>RefreshCharts</onrefresh>
  </callbacks>

</step>

</wizard>

<wizard name="FluentWizard" version="1" context="Fluent" icon="wizard_icon">
  <description>Simple wizard for demonstration in Fluent.</description>

  <step name="Geometry" caption="Geometry" version="1" HelpFile="help/geometry.html">
    <description>Create a geometry component.</description>

    <callbacks>
      <onupdate>EmptyAction</onupdate>
      <onreset>DeleteGeometry</onreset>
    </callbacks>

    <propertygroup display="caption" name="definition" caption="Basic properties" >
      <property name="filename" caption="Geometry file name" control="fileopen" />
      <property name="myint" caption="Integer value" control="integer" />
      <property name="mytext" caption="Text value" control="text" />
      <property name="myquantity" caption="Quantity value" control="float" unit="Pressure" />
      <property name="myreadonly" caption=" Readonly value" control="text" readonly="true" default="My value" />
      <propertygroup display="property" name="myselect" caption="List of choice" control="select" default="Option1">
        <attributes options="Option1,Option2" />
        <property name="option1" caption="Option1 value" control="text" visibleon="Option1" />
        <property name="option2first" caption="Option2 first value" control="float" unit="Pressure" visibleon="Option2" />
        <property name="option2second" caption="Option2 second value" control="float" unit="Length" visibleon="Option2" />
      </propertygroup>
    </propertygroup>

  </step>

  <step name="Mechanical" caption="Mechanical" enabled="true" version="1" HelpFile="help/mechanical.html">
    <description>Create a mechanical component.</description>

    <callbacks>
      <onupdate>EmptyAction</onupdate>
      <onreset>DeleteMechanical</onreset>
    </callbacks>

    <property name="name" caption="System name" control="text" />
    <!--<propertytable name="table" caption="TabularData" display="worksheet" control="applycancel" class="Worksheet.PropertyGroupEditor.PGEeditor">
      <property name="Temperature" caption="Temperature" unit="Temperature" control="float"></property>
      <property name="Pressure" caption="Pressure" unit="Pressure" control="float"></property>
    </propertytable>-->

  </step>

  <step name="Fluent" caption="Fluent" version="1" HelpFile="help/fluent.html">
    <description>Create a fluent component.</description>

    <callbacks>
      <onrefresh>CreateDialog</onrefresh>
      <onupdate>EmptyAction</onupdate>
      <onreset>EmptyReset</onreset>
    </callbacks>
  
```

```

</callbacks>

<property name="name" caption="System name" control="text" />
<property name="dialog" caption="Dialog" control="text">
  <callbacks>
    <onvalidate>ValidateDialog</onvalidate>
  </callbacks>
</property>
<property name="dialog2" caption="DialogProgress" control="text">
  <callbacks>
    <onvalidate>ValidateDialogProgress</onvalidate>
  </callbacks>
</property>
<property name="nextstep" caption="Next Step" control="select" >
  <callbacks>
    <onvalidate>ValidateNextStep</onvalidate>
  </callbacks>
</property>

</step>

<step name="ReportView" caption="ReportView" version="1" layout="ReportView@WizardDemos">
  <description>Simple example to demonstrate how a report can be displayed.</description>

  <callbacks>
    <onrefresh>RefreshReport</onrefresh>
    <onreset>EmptyReset</onreset>
  </callbacks>

</step>

<step name="CustomStep" caption="CustomStep" enabled="true" version="1" layout="MyLayout@WizardDemos">
  <description>Create a mechanical component.</description>

  <callbacks>
    <onrefresh>RefreshMechanical</onrefresh>
    <onupdate>LogReport</onupdate>
    <onreset>EmptyReset</onreset>
  </callbacks>

  <property name="name" caption="System name" control="text" />

</step>

<step name="Charts" caption="Charts" enabled="true" version="1" layout="GraphLayout@WizardDemos">
  <description>Demonstrate all chart capabilities.</description>

  <callbacks>
    <onrefresh>RefreshCharts</onrefresh>
  </callbacks>

</step>

</wizard>

<wizard name="EDWizard" version="1" context="ElectronicsDesktop" icon="wizard_icon">
  <description>Simple wizard for demonstration in ElectronicsDesktop.</description>

  <step name="Geometry" caption="Geometry" version="1" HelpFile="help/geometry.html">
    <description>Create a geometry component.</description>

    <callbacks>
      <onupdate>EmptyAction</onupdate>
      <onreset>DeleteGeometry</onreset>
    </callbacks>

    <propertygroup display="caption" name="definition" caption="Basic properties" >
      <property name="filename" caption="Geometry file name" control="fileopen" />
      <property name="myint" caption="Integer value" control="integer" />
      <property name="mytext" caption="Text value" control="text" />
      <property name="myquantity" caption="Quantity value" control="float" unit="Pressure" />
    </propertygroup>
  </step>
</wizard>
```

```
<property name="myreadonly" caption="Readonly value" control="text" readonly="true" default="My value" />
<propertygroup display="property" name="myselect" caption="List of choice" control="select"
default="Option1">
  <attributes options="Option1,Option2" />
  <property name="option1" caption="Option1 value" control="text" visibleon="Option1" />
  <property name="option2first" caption="Option2 first value" control="float" unit="Pressure"
visibleon="Option2" />
  <property name="option2second" caption="Option2 second value" control="float" unit="Length"
visibleon="Option2" />
</propertygroup>
</propertygroup>

</step>

<step name="Mechanical" caption="Mechanical" enabled="true" version="1" HelpFile="help/mechanical.html">
  <description>Create a mechanical component.</description>

  <callbacks>
    <onupdate>EmptyAction</onupdate>
    <onreset>DeleteMechanical</onreset>
  </callbacks>

  <property name="name" caption="System name" control="text" />
  <!--<propertytable name="table" caption="TabularData" display="worksheet" control="applycancel"
class="Worksheet.PropertyGroupEditor.PGEdition">
    <property name="Temperature" caption="Temperature" unit="Temperature" control="float"></property>
    <property name="Pressure" caption="Pressure" unit="Pressure" control="float"></property>
  </propertytable>-->

</step>

<step name="Fluent" caption="Fluent" version="1" HelpFile="help/fluent.html">
  <description>Create a fluent component.</description>

  <callbacks>
    <onrefresh>CreateDialog</onrefresh>
    <onupdate>EmptyAction</onupdate>
    <onreset>EmptyReset</onreset>
  </callbacks>

  <property name="name" caption="System name" control="text" />
  <property name="dialog" caption="Dialog" control="text">
    <callbacks>
      <onvalidate>ValidateDialog</onvalidate>
    </callbacks>
  </property>
  <property name="dialog2" caption="DialogProgress" control="text">
    <callbacks>
      <onvalidate>ValidateDialogProgress</onvalidate>
    </callbacks>
  </property>
  <property name="nextstep" caption="Next Step" control="select" >
    <callbacks>
      <onvalidate>ValidateNextStep</onvalidate>
    </callbacks>
  </property>
</step>

<step name="ReportView" caption="ReportView" version="1" layout="ReportView@WizardDemos">
  <description>Simple example to demonstrate how report can be displayed.</description>

  <callbacks>
    <onrefresh>RefreshReport</onrefresh>
    <onreset>EmptyReset</onreset>
  </callbacks>

</step>

<step name="CustomStep" caption="CustomStep" enabled="true" version="1" layout="MyLayout@WizardDemos">
  <description>Create a mechanical component.</description>
```

```

<callbacks>
<onrefresh>RefreshMechanical</onrefresh>
<onupdate>LogReport</onupdate>
<onreset>EmptyReset</onreset>
</callbacks>

<property name="name" caption="System name" control="text" />

</step>

<step name="Charts" caption="Charts" enabled="true" version="1" layout="GraphLayout@WizardDemos">
<description>Demonstrate all chart capabilities.</description>

<callbacks>
<onrefresh>RefreshCharts</onrefresh>
</callbacks>

</step>

</wizard>

<wizard name="CreateBridge" version="1" context="DesignModeler" icon="wizard_icon">
<description>Simple wizard for demonstration in DesignModeler.</description>

<step name="Deck" caption="Deck" version="1" HelpFile="help/dm1.html">
<description>Create the deck.</description>

<callbacks>
<onupdate>UpdateDeck</onupdate>
</callbacks>

<propertygroup display="caption" name="Deck" caption="Deck Definition" >
<property name="Length" caption="Length" control="float" unit="Length" default="300 [m]" />
<property name="Width" caption="Width" control="float" unit="Length" default="20 [m]" />
<property name="Beams" caption="Beams" control="integer" default="31" />
</propertygroup>

</step>

<step name="Supports" caption="Supports" enabled="true" version="1" HelpFile="help/dm2.html">
<description>Create supports.</description>

<callbacks>
<onupdate>UpdateSupports</onupdate>
</callbacks>

<propertygroup display="caption" name="Supports" caption="Supports Definition" >
<property name="Height" caption="Height" control="float" unit="Length" default="100 [m]" />
<property name="Number" caption="Number" control="integer" default="3" />
</propertygroup>

</step>

</wizard>

<wizard name="CreateBridge" version="1" context="SpaceClaim" icon="wizard_icon">
<description>Simple wizard for demonstration in SpaceClaim.</description>

<step name="DeckSC" caption="DeckSC" version="1" context="SpaceClaim">
<description>Create the deck.</description>

<callbacks>
<onupdate>UpdateDeckSC</onupdate>
</callbacks>

<propertygroup display="caption" name="Deck" caption="Deck Definition" >
<property name="Length" caption="Length" control="float" unit="Length" default="300 [m]" />
<property name="Width" caption="Width" control="float" unit="Length" default="20 [m]" />
<property name="Beams" caption="Beams" control="integer" default="31" />
</propertygroup>
</step>
```

```
<step name="SupportsSC" caption="SupportsSC" context="SpaceClaim" enabled="true" version="1">
  <description>Create supports.</description>

  <callbacks>
    <onupdate>UpdateSupportsSC</onupdate>
  </callbacks>

  <propertygroup display="caption" name="Supports" caption="Supports Definition" >
    <property name="Height" caption="Height" control="float" unit="Length" default="100 [m]" />
    <property name="Number" caption="Number" control="integer" default="3" />
  </propertygroup>
</step>
</wizard>

<wizard name="SimpleAnalysis" version="1" context="Mechanical" icon="wizard_icon">
  <description>Simple wizard to illustrate how to setup, solve and analyse results of a simulation process.</description>

  <step name="Mesh" caption="Mesh" version="1" HelpFile="help/ds1.html">
    <description>Setup some mesh controls.</description>

    <callbacks>
      <onreset>RemoveControls</onreset>
      <onupdate>CreateMeshControls</onupdate>
    </callbacks>

    <propertygroup display="caption" name="Sizing" caption="Mesh Sizing" >
      <property name="Location" caption="Edge Location" control="geometry_selection">
        <attributes selection_filter="edge" />
        <callbacks>
          <isvalid>IsLocationValid</isvalid>
        </callbacks>
      </property>
      <property name="Ndiv" caption="Divisions" control="integer" />
    </propertygroup>

  </step>

  <step name="Solution" caption="Solution" version="1" HelpFile="help/ds2.html">
    <description>Setup loads.</description>

    <callbacks>
      <onrefresh>RefreshLoads</onrefresh>
      <onreset>RemoveLoads</onreset>
      <onupdate>CreateLoads</onupdate>
    </callbacks>

    <propertygroup display="caption" name="Mesh" caption="Mesh Statistics" >
      <property name="Nodes" caption="Nodes" control="text" readonly="true" />
      <property name="Elements" caption="Elements" control="text" readonly="true" />
    </propertygroup>
    <propertygroup display="caption" name="FixedSupport" caption="Fixed Support" >
      <property name="Location" caption="Face Location" control="geometry_selection">
        <attributes selection_filter="face" />
        <callbacks>
          <isvalid>IsLocationFSValid</isvalid>
        </callbacks>
      </property>
    </propertygroup>

  </step>

  <step name="Results" caption="Results" version="1" HelpFile="help/ds3.html">
    <description>View Results.</description>

    <callbacks>
      <onrefresh>RefreshResults</onrefresh>
    </callbacks>

    <property name="Res" caption="Deformation" control="text" readonly="true" />
  </step>

```

```

</step>
</wizard>

<wizard name="BridgeSimulation" version="1" context="Project" icon="bridge">
  <description>Simple wizard for mixed wizard demonstration.</description>

  <step name="Project" caption="Create Project" version="1" context="Project" HelpFile="help/prj1.html">
    <description>Create a static structural analysis.</description>

    <callbacks>
      <onupdate>CreateStaticStructural</onupdate>
      <!--<onreset>DeleteStaticStructural</onreset>-->
    </callbacks>

    <property name="Name" caption="system name" control="text" />
    <property name="Context" caption="geometry context" control="select">
      <attributes options="DesignModeler,SpaceClaim" />
      <callbacks>
        <onvalidate>OnSelectContext</onvalidate>
      </callbacks>
    </property>
  </step>

  <step name="DeckDM" caption="DeckDM" version="1" context="DesignModeler" HelpFile="help/dm1.html">
    <description>Create the deck.</description>

    <callbacks>
      <onupdate>UpdateDeck</onupdate>
    </callbacks>

    <propertygroup display="caption" name="Deck" caption="Deck Definition" >
      <property name="Length" caption="Length" control="float" unit="Length" default="300 [m]" />
      <property name="Width" caption="Width" control="float" unit="Length" default="20 [m]" />
      <property name="Beams" caption="Beams" control="integer" default="31" />
    </propertygroup>
  </step>

  <step name="SupportsDM" caption="SupportsDM" context="DesignModeler" enabled="true" version="1"
HelpFile="help/dm2.html">
    <description>Create supports.</description>

    <callbacks>
      <onupdate>UpdateSupports</onupdate>
    </callbacks>

    <propertygroup display="caption" name="Supports" caption="Supports Definition" >
      <property name="Height" caption="Height" control="float" unit="Length" default="100 [m]" />
      <property name="Number" caption="Number" control="integer" default="3" />
    </propertygroup>
  </step>

  <step name="DeckSC" caption="DeckSC" version="1" context="SpaceClaim">
    <description>Create the deck.</description>

    <callbacks>
      <onupdate>UpdateDeckSC</onupdate>
    </callbacks>

    <propertygroup display="caption" name="Deck" caption="Deck Definition" >
      <property name="Length" caption="Length" control="float" unit="Length" default="300 [m]" />
      <property name="Width" caption="Width" control="float" unit="Length" default="20 [m]" />
      <property name="Beams" caption="Beams" control="integer" default="31" />
    </propertygroup>
  </step>

  <step name="SupportsSC" caption="SupportsSC" context="SpaceClaim" enabled="true" version="1">
    <description>Create supports.</description>

```

```
<callbacks>
<onupdate>UpdateSupportsSC</onupdate>
</callbacks>

<propertygroup display="caption" name="Supports" caption="Supports Definition" >
<property name="Height" caption="Height" control="float" unit="Length" default="100 [m]" />
<property name="Number" caption="Number" control="integer" default="3" />
</propertygroup>
</step>

<step name="Mesh" caption="Mesh" version="1" context="Mechanical" HelpFile="help/ds1.html">
<description>Setup some mesh controls.</description>

<callbacks>
<onreset>RemoveControls</onreset>
<onupdate>CreateMeshControls</onupdate>
</callbacks>

<propertygroup display="caption" name="Sizing" caption="Mesh Sizing" >
<property name="Location" caption="Edge Location" control="geometry_selection">
<attributes selection_filter="edge" />
<callbacks>
<isvalid>IsLocationValid</isvalid>
</callbacks>
</property>
<property name="Ndiv" caption="Divisions" control="integer" />
</propertygroup>

</step>

<step name="Solution" caption="Solution" version="1" context="Mechanical" HelpFile="help/ds2.html">
<description>Setup loads.</description>

<callbacks>
<onrefresh>RefreshLoads</onrefresh>
<onreset>RemoveLoads</onreset>
<onupdate>CreateLoads</onupdate>
</callbacks>

<propertygroup display="caption" name="Mesh" caption="Mesh Statistics" >
<property name="Nodes" caption="Nodes" control="text" readonly="true" />
<property name="Elements" caption="Elements" control="text" readonly="true" />
</propertygroup>
<propertygroup display="caption" name="FixedSupport" caption="Fixed Support" >
<property name="Location" caption="Face Location" control="geometry_selection">
<attributes selection_filter="face" />
<callbacks>
<isvalid>IsLocationFSValid</isvalid>
</callbacks>
</property>
</propertygroup>

</step>

<step name="Results" caption="Results" version="1" context="Project" HelpFile="help/prj2.html">
<description>View Results.</description>

<callbacks>
<onrefresh>RefreshResultsProject</onrefresh>
</callbacks>

<property name="Res" caption="Deformation" control="text" readonly="true" />

</step>

</wizard>

</extension>
```

IronPython Script

Because this mixed wizard incorporates steps on the Workbench **Project** tab with steps from the existing DesignModeler, Mechanical, and SpaceClaim wizards, the XML extension definition file references all four IronPython script described in previous sections:

- main.py
- dm.py
- ds.py
- sc.py

Fluent Wizard (MSH Input File)

The extension **FluentDemo1** contains a target product wizard for Fluent named **Simple Analysis (Fluent Demo 1)**. This Fluent wizard imports an MSH file, runs a simple analysis, generates results, and displays the results in a custom panel in the user interface. This example can be used in either of the following scenarios:

- Within a Workbench project that contains a **Fluent (with Fluent Meshing)** system
- In a Fluent standalone project with **Meshing Mode** selected

Note

The extension **FluentDemo1** and sample input file are included in the package **ACT Wizards Templates**, which you can download from the [ACT Resources](#) page on the [ANSYS Customer Portal](#). For more information, see [Extension Examples \(p. 273\)](#).

XML Extension Definition File

Definitions follow for codes blocks in the XML extension definition file `FluentDemo1.xml`.

Extension Definition

The `<extension>` block is the top-most level in the hierarchy.

- The attributes `name` and `version` specify the extension name and version.
- The attribute `icon` specifies the image file to display as the icon for the extension. You specify the location of the `images` folder for all images used by the extension in the `<interface>` block.

Guid Definition

The `<guid>` block specifies a unique identifier for the extension.

Script Definition

The `<script>` block specifies the IronPython script that is referenced by the extension. This extension references `main.py`.

Wizard Interface Definition

The `<interface>` block defines the user interface for the wizard.

- The attribute **context** is set to **Fluent** because the extension runs from ANSYS Fluent.
- The attribute **images** specifies the **images** folder for all images that the extension is to use. You use the attribute **icon** for the extension, wizard, step, and so on to specify an image file in this folder to use as an icon.

Custom Layout Definition

The **<uidefinition>** block defines a custom layout named **ResultsView** for the Fluent wizard. It is made up of five custom components: **Title**, **Properties**, **Report**, **Chart**, and **Submit**. The **Properties**, **Report**, and **Chart** components are views, and the **Submit** component is a button.

Wizard Definition

The wizard **Simple Analysis (Fluent Demo 1)** and its steps are defined in the **<wizard>** block named **Simple Analysis (Fluent Demo 1)**.

- The attributes **name** and **version** specify the wizard name and version.
- The attribute **context** specifies the ANSYS product in which the wizard is executed. Because the wizard runs from Fluent, whether a stand-alone instance or an instance within Workbench, **context** is set to **Fluent**.
- The attribute **icon** specifies the filename of the image to display as the icon for the wizard: **wizard_icon**. The file **wizard_icon.png** is stored in the **images** folder specified for the attribute **images** in the **<interface>** block.
- The attribute **description** specifies the text to display in the **About** window for the extension when it is accessed from the **Extension Manager** and in the **About** window for the wizard when it is accessed from the **Wizards** page.

Step Definition

The **<step>** blocks define the steps of the wizard. This wizard has two steps: **Analysis** and **Results**. For each step:

- The attributes **name**, **version**, and **caption** define the step name, version, and display text for the step.
- The attribute **context** specifies the ANSYS product in which the specific step is executed. You only need to set the context for the step if it differs from the context for the wizard.
- The attribute **HelpFile** specifies the HTML file to display as the custom help for the step.
- The attribute **layout** specifies the custom layout to apply to the step. For the step **Results**, **layout** is set to **ResultsView@FluentDemo1**. The custom layout **ResultsView** is defined in the **<uidefinition>** block.
- Any **<property>** blocks specify properties and property attributes to be used in the step. For the step **Analysis**, the property **velocity** is defined. The attribute **caption** is set to **velocity**. For the step **Results**, the property **pressure** is defined. The attribute **caption** is set to **Pressure**.
- The **<callbacks>** blocks specify callbacks to functions defined in the script **main.py**. These are discussed in more detail in [Reviewing the Analysis \(p. 355\)](#).

The XML extension definition file **FluentDemo1.xml** follows.

```
<extension name="FluentDemo1" version="1">
<guid>A0A3E28C-E094-4B2D-8A91-A0B0884D1AE2</guid>
<author>ANSYS Inc.</author>
<description>This extension demonstrates how to create a wizard in Fluent (.msh file import).</description>

<script src="main.py" />

<interface context="Fluent">
<images>images</images>
</interface>

<uidefinition>

<layout name="ResultsView">
<component
  name="Title"
  topAttachment=" "
  topOffset="10"
  leftAttachment=" "
  leftOffset="10"
  rightAttachment=" "
  rightOffset="10"
  bottomOffset="10"
  bottomAttachment="Properties"
  widthType="Percentage"
  width="100"
  heightType="FitToContent"
  height="200"
  componentType="startPageHeaderComponent" />
<component
  name="Properties"
  topAttachment="Title"
  leftAttachment=" "
  leftOffset="10"
  rightAttachment=" "
  rightOffset="10"
  bottomAttachment="Report"
  bottomOffset="10"
  widthType="Percentage"
  width="100"
  heightType="Percentage"
  height="20"
  componentType="propertiesComponent" />
<component
  name="Report"
  topAttachment="Properties"
  leftAttachment=" "
  leftOffset="10"
  rightAttachment=" "
  rightOffset="10"
  bottomAttachment="Chart"
  bottomOffset="10"
  widthType="Percentage"
  width="100"
  heightType="Percentage"
  height="40"
  componentType="helpContentComponent" />
<component
  name="Chart"
  topAttachment="Report"
  leftAttachment=" "
  leftOffset="10"
  rightAttachment=" "
  rightOffset="10"
  bottomAttachment="Submit"
  bottomOffset="10"
  widthType="Percentage"
  width="100"
  heightType="Percentage"
  height="40"
  componentType="chartComponent" />
<component
```

```
name="Submit"
topAttachment="Chart"
leftAttachment=""
leftOffset="10"
rightAttachment=""
rightOffset="10"
bottomAttachment=""
bottomOffset="10"
widthType="Percentage"
width="100"
heightType="FitToContent"
height="50"
componentType="buttonsComponent" />
</layout>

</uidefinition>

<wizard name="Simple Analysis (Fluent Demo 1)" version="1" context="Fluent" icon="wizard_icon">
<description>Generate the mesh and solve the analysis.</description>

<step name="Analysis" caption="Analysis" version="1">
<callbacks>
<onrefresh>ImportMesh</onrefresh>
<onupdate>CreateAnalysis</onupdate>
</callbacks>

<property name="velocity" caption="Velocity" control="float" default="3" />
</step>

<step name="Results" caption="Results" version="1" layout="ResultsView@FluentDemol">
<callbacks>
<onrefresh>CreateReport</onrefresh>
</callbacks>

<property name="pressure" caption="Pressure Drop" control="text" readonly="true" />
</step>

</wizard>

</extension>
```

IronPython Script

The IronPython script `main.py` follows. This script defines all functions executed by the callbacks in the XML extension definition file:

- The function `ImportMesh` is invoked by the callback `<onrefresh>` in the step `Analysis`.
- The function `CreateAnalysis` is invoked by the callback `<onupdate>` in the step `Analysis`.
- The function `CreateReport` is invoked by the callback `<onrefresh>` in the step `Results`.

```
def ImportMesh(step):
    tui = ExtAPI.Application.ScriptByName("TUI")

    tui.SendCommand("""switch-to-meshing-mode""")
    tui.SendCommand("""/file/read-mesh "{}""".format(System.IO.Path.Combine(
ExtAPI.Extension.InstallDir,"final.msh")))
    tui.SendCommand("""switch-to-solution-mode yes yes""")
    tui.SendCommand("""/display/mesh ok""")
    tui.SendCommand("""/display/views/restore-view right""")

def CreateAnalysis(step):
    tui = ExtAPI.Application.ScriptByName("TUI")

    res = System.IO.Path.GetTempFileName()
    img = System.IO.Path.GetTempFileName() + ".jpg"
    curve = System.IO.Path.GetTempFileName()
```

```

tui.SendCommand("""/define/boundary-conditions/velocity-inlet inlet no no yes yes no {0} no
0""".format(step.Properties["velocity"].Value), True)
tui.SendCommand("""/solve/initialize/hyb-initialization""", True)
tui.SendCommand("""/solve/iterate 50""", True)
tui.SendCommand("""/display/mesh ok""", True)
tui.SendCommand("""/display/views	restore-view right""", True)
tui.SendCommand("""/display/set/contours/filled-contours? yes""", True)
tui.SendCommand("""/display/set/contours/surfaces wall inlet outlet ()""", True)
tui.SendCommand("""/display/contour pressure 0 2""", True)
tui.SendCommand("""/display/save-picture "{0}" """.format(img), True)
tui.SendCommand("""/surface/rake-surface rake-5 0 0 0 0 1 50""", True)
tui.SendCommand("""/plot/plot-direction 0 0 1""", True)
tui.SendCommand("""/plot/plot yes "{0}" yes yes no no pressure yes 0 0 1 rake-5 ()""".format(curve),
True)
tui.SendCommand("""/report/surface-integrals/area-weighted-avg inlet () pressure yes "{0}"
""".format(res), True)

val = 0.
n = 0;
with System.IO.StreamReader(res) as reader:
    while n!=5:
        line = reader.ReadLine()
        if line!=None:
            n+=1
    str = line.Substring(32)
    val = System.Double.Parse(str, System.Globalization.CultureInfo.InvariantCulture)

x=[]
y=[]
n = 0;
with System.IO.StreamReader(curve) as reader:
    while n!=5:
        line = reader.ReadLine()
        if line!=None:
            n+=1
    while not line.StartsWith(" "):
        tab = line.Split("\t")
        x.Add(System.Double.Parse(tab[0], System.Globalization.CultureInfo.InvariantCulture))
        y.Add(System.Double.Parse(tab[1], System.Globalization.CultureInfo.InvariantCulture))
        line = reader.ReadLine()
System.IO.File.Delete(res)
System.IO.File.Delete(curve)

step.NextStep.Properties["pressure"].Value = val
step.NextStep.Attributes["img"] = img
step.NextStep.Attributes["x"] = x
step.NextStep.Attributes["y"] = y

def CreateReport(step):
    graph = step.UserInterface.GetComponent("Chart")
    graph.Title("Static Pressure")
    graph.ShowLegend(False)
    graph.Plot(step.Attributes["x"], step.Attributes["y"], key="Static Pressure", color='g')

    help = step.UserInterface.GetComponent("Report")
    help.SetHtmlContent(ExtAPI.Extension.InstallDir,"""<center></center>""".format(step.Attributes["img"]))

```

Reviewing the Analysis

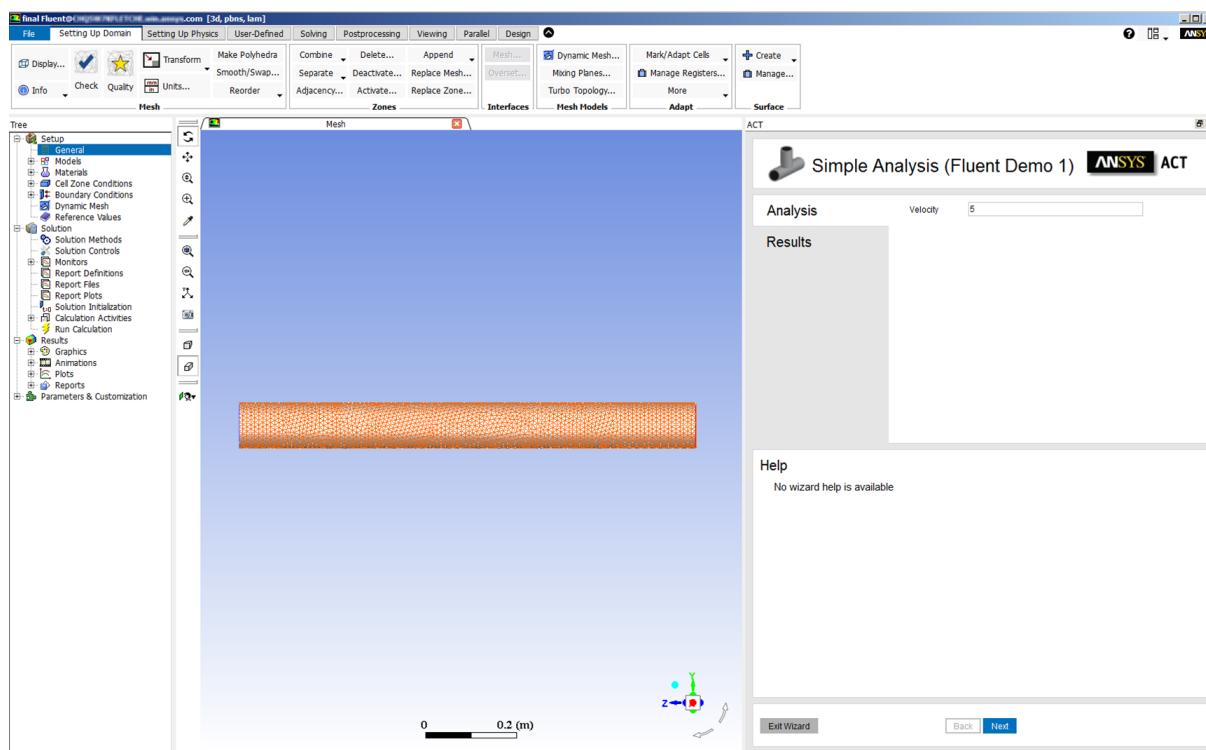
This section reviews the user interface and the processes performed in each step of the wizard **Simple Analysis (Fluent Demo 1)**.

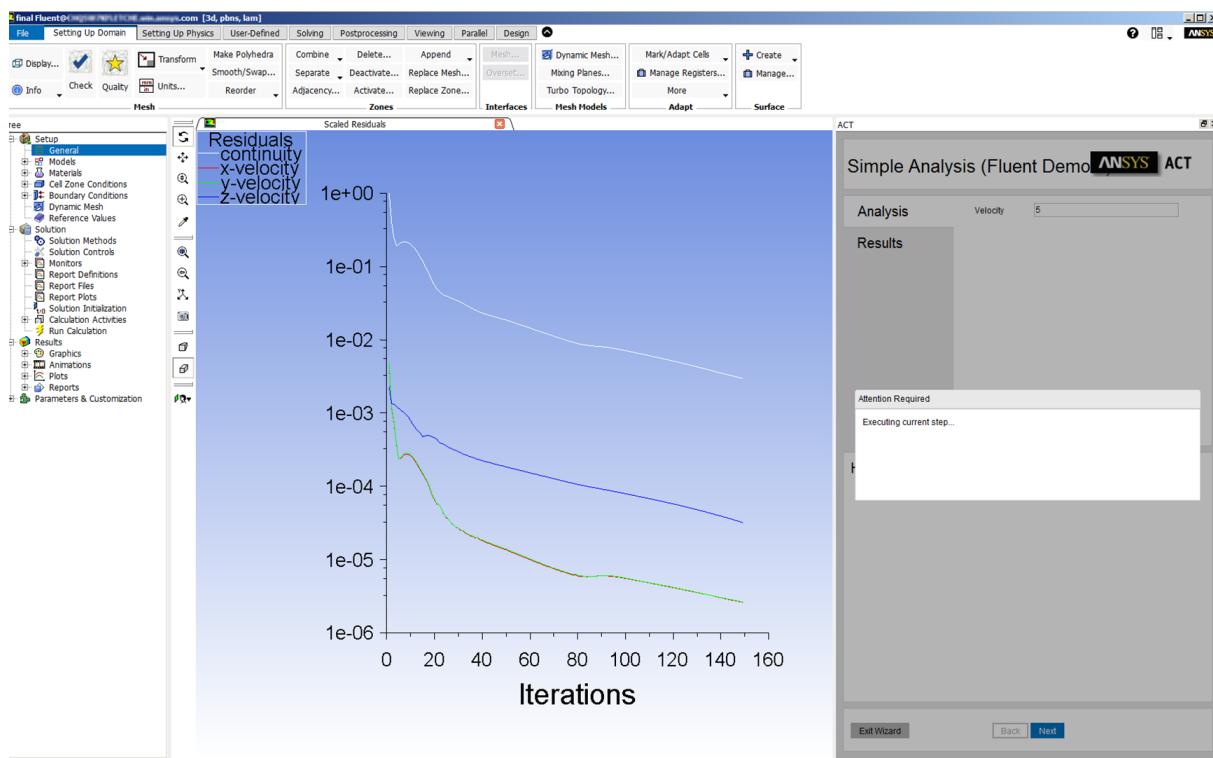
Reviewing the Analysis Step

The first step is **Analysis**, which imports the mesh file and creates the analysis. It requires the user to specify a value for the property **Velocity**.

In this step:

- The function **ImportMesh** is invoked by the callback **<onrefresh>**. It imports the file `final.msh` and displays the mesh in the **Mesh** view.
- The function **CreateAnalysis** is invoked by the callback **<onupdate>** when the user clicks the **Next** button. This function sets up the analysis.
 - The property **Velocity** defined in the **<property>** block is shown in the GUI as a float field with a default value of **3**. The user can enter the desired value. In this example, a value of **5** is entered.
 - The additional actions to be performed as part of the analysis are defined.
 - Intermediate results are shown dynamically in the **Scaled Residuals** window as the analysis is performed.



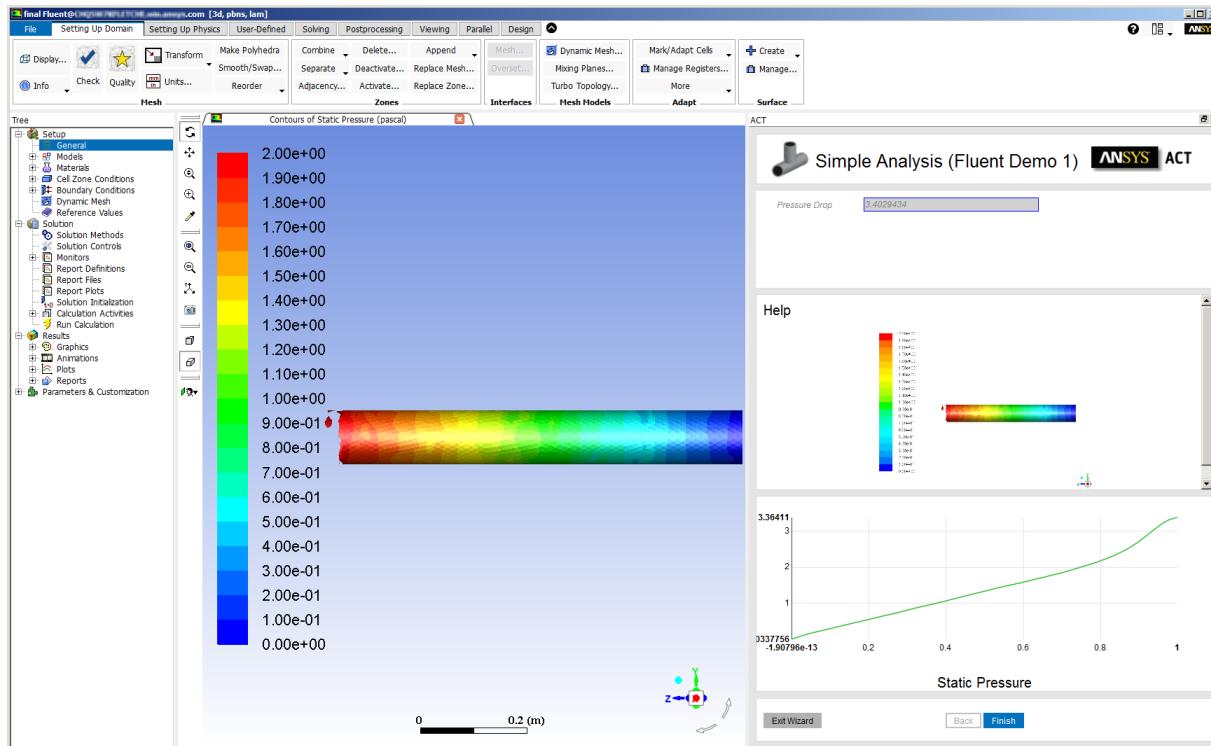


Reviewing the Results Step

The second step is **Results**, which creates a report of the results.

In this step:

- The function `CreateReport` is invoked by the callback `<onrefresh>`.
- The `<layout>` block specifies the custom layout to apply to the results. This is a child block within the `<uidefinition>` block.
- The wizard takes the results obtained and displays them:
 - The calculated value for the property `PressureDrop` defined in the `<property>` block is displayed in the GUI as a ready-only field.
 - The static pressure results are shown in the **Graph** and **Contours** views.



Fluent Wizard (CAS Input File)

The extension **FluentDemo2** contains a target product wizard for Fluent named **Simple Analysis (Fluent Demo 2)**. This Fluent wizard imports a CAS file, runs a simple analysis, generates results, and displays the results in a customized panel in the user interface. This example can be used in either of the following scenarios:

- Within a Workbench project that contains a **Fluent** or **Fluent (with CFD-Post)** system
- In a Fluent standalone project with **Meshing Mode** not selected

Note

The extension **FluentDemo2** and sample input file are included in the package **ACT Wizards Templates**. You can download this package from the [ACT Resources](#) page on the [ANSYS Customer Portal](#). For more information, see [Extension Examples \(p. 273\)](#).

XML Extension Definition File

Definitions follow for codes blocks in the XML extension definition file `FluentDemo2.xml`.

Extension Definition

The `<extension>` block is the top-most level in the hierarchy.

- The attributes `name` and `version` specify the extension name and version.
- The attribute `icon` specifies the image file to display as the icon for the extension. You specify the location of the `images` folder for all images used by the extension in the `<interface>` block.

Guid Definition

The `<guid>` block specifies a unique identifier for the extension.

Script Definition

The `<script>` block specifies the IronPython script that is referenced by the extension. This extension references `main.py`.

Wizard Interface Definition

The `<interface>` block defines the user interface for the wizard.

- The attribute `context` is set to **Fluent** because the extension runs from ANSYS Fluent.
- The attribute `images` specifies the `images` folder for all images that the extension is to use. You use the attribute `icon` for the extension, wizard, step, and so on to specify an image file in this folder to use as an icon.

Custom Layout Definition

The `<uidefinition>` block defines a custom layout called **ResultsView** for the Fluent wizard. It is made up of five custom components: **Title**, **Properties**, **Report**, **Chart**, and **Submit**. The **Properties**, **Report**, and **Chart** components are views, and the **Submit** component is a button.

Wizard Definition

The wizard **Simple Analysis (Fluent Demo 2)** and its steps are defined in the `<wizard>` block named **Simple Analysis (Fluent Demo 2)**.

- The attributes `name` and `version` specify the wizard name and version.
- The attribute `context` specifies the ANSYS product in which the wizard is executed. Because the wizard runs from Fluent, whether a stand-alone instance or an instance within Workbench, `context` is set to **Fluent**.
- The attribute `icon` specifies the filename of the image to be used as the icon for the wizard: `wizard_icon`. The file `wizard_icon.png` is stored in the `images` folder defined in the `<images>` block of the interface definition.
- The attribute `description` specifies the text to display in the **About** window for the extension when it is accessed from the **Extension Manager** and in the **About** window for the wizard when it is accessed from the **Wizards** page.

Step Definition

The `<step>` blocks define the steps of the wizard. This wizard has two steps: **Analysis** and **Results**. For each step:

- The attributes `name`, `version`, and `caption` define the step name, version, and display text for the step.
- The attribute `context` specifies the ANSYS product in which the specific step is executed. You only need to set the context for the step if it differs from the context for the wizard.
- The attribute `HelpFile` specifies the HTML file to display as the custom help for the step.
- The attribute `layout` specifies the custom layout to apply to the step. For the step **Results**, `layout` is set to `ResultsView@FluentDemo2`. The custom layout `ResultsView` is defined in the `<uidefinition>` block.

- Any **<property>** blocks specify properties and property attributes to be used in the step. For the step **Analysis**, the property **velocity** is defined. The attribute **caption** is set to **Velocity**. For the step **Results**, the property **pressure** is defined. The attribute **caption** is set to **Pressure**.
- The **<callbacks>** blocks specify callbacks to functions defined in the script `main.py`. For more information, see [Reviewing the Analysis \(p. 355\)](#).

The XML extension definition file `FluentDemo2.xml` follows.

```
<extension name="FluentDemo2" version="1">
    <guid>49B358FB-EAFB-4678-BBBD-82E949B46F70</guid>
    <author>ANSYS Inc.</author>
    <description>This extension demonstrates how to create a wizard in Fluent (.cas file import).</description>
    <script src="main.py" />

    <interface context="Fluent">
        <images>images</images>
    </interface>

    <uidefinition>

        <layout name="ResultsView">
            <component
                name="Title"
                topAttachment=""
                topOffset="10"
                leftAttachment=""
                leftOffset="10"
                rightAttachment=""
                rightOffset="10"
                bottomOffset="10"
                bottomAttachment="Properties"
                widthType="Percentage"
                width="100"
                heightType="FitToContent"
                height="200"
                componentType="startPageHeaderComponent" />
            <component
                name="Properties"
                topAttachment="Title"
                leftAttachment=""
                leftOffset="10"
                rightAttachment=""
                rightOffset="10"
                bottomAttachment="Report"
                bottomOffset="10"
                widthType="Percentage"
                width="100"
                heightType="Percentage"
                height="20"
                componentType="propertiesComponent" />
            <component
                name="Report"
                topAttachment="Properties"
                leftAttachment=""
                leftOffset="10"
                rightAttachment=""
                rightOffset="10"
                bottomAttachment="Chart"
                bottomOffset="10"
                widthType="Percentage"
                width="100"
                heightType="Percentage"
                height="40"
                componentType="helpContentComponent" />
            <component
                name="Chart"
                topAttachment="Report"
                leftAttachment=""
```

```

        leftOffset="10"
        rightAttachment=" "
        rightOffset="10"
        bottomAttachment="Submit"
        bottomOffset="10"
        widthType="Percentage"
        width="100"
        heightType="Percentage"
        height="40"
        componentType="chartComponent" />
    <component
        name="Submit"
        topAttachment="Chart"
        leftAttachment=" "
        leftOffset="10"
        rightAttachment=" "
        rightOffset="10"
        bottomAttachment=" "
        bottomOffset="10"
        widthType="Percentage"
        width="100"
        heightType="FitToContent"
        height="50"
        componentType="buttonsComponent" />
</layout>

</uidefinition>

<wizard name="Simple Analysis (Fluent Demo 2)" version="1" context="Fluent" icon="wizard_icon">
    <description>Generate the mesh and solve the analysis.</description>

    <step name="Analysis" caption="Analysis" version="1">
        <callbacks>
            <onrefresh>ImportModel</onrefresh>
            <onupdate>CreateAnalysis</onupdate>
        </callbacks>

        <property name="velocity" caption="Velocity" control="float" default="3" />
    </step>

    <step name="Results" caption="Results" version="1" layout="ResultsView@FluentDemo2">
        <callbacks>
            <onrefresh>CreateReport</onrefresh>
        </callbacks>

        <property name="pressure" caption="Pressure Drop" control="text" readonly="true" />
    </step>
</wizard>
</extension>
```

IronPython Script

The IronPython script `main.py` follows. This script defines all functions executed by the callbacks in the XML extension definition file:

- The function `ImportModel` is invoked by the callback `<onrefresh>` in the step `Analysis`.
- The function `CreateAnalysis` is invoked by the callback `<onupdate>` in the step `Analysis`.
- The function `CreateReport` is invoked by the callback `<onrefresh>` in the step `Results`.

```

def ImportModel(step):
    tui = ExtAPI.Application.ScriptByName("TUI")

    tui.SendCommand("""/file/read-case "{0}" """.format(System.IO.Path.Combine(
ExtAPI.Extension.InstallDir, "final.cas")))
```

```

tui.SendCommand(" ///display/mesh ok")
tui.SendCommand(" ///display/views/restore-view right")

def CreateAnalysis(step):
    tui = ExtAPI.Application.ScriptByName("TUI")

    res = System.IO.Path.GetTempFileName()
    img = System.IO.Path.GetTempFileName() + ".jpg"
    curve = System.IO.Path.GetTempFileName()
    tui.SendCommand(" ///define/boundary-conditions/velocity-inlet inlet no no yes yes no {0} no
0""".format(step.Properties["velocity"].Value), True)
    tui.SendCommand(" ///solve/initialize/hyb-initialization", True)
    tui.SendCommand(" ///solve/iterate 50", True)
    tui.SendCommand(" ///display/mesh ok", True)
    tui.SendCommand(" ///display/views/restore-view right", True)
    tui.SendCommand(" ///display/set/contours/filled-contours? yes", True)
    tui.SendCommand(" ///display/set/contours/surfaces wall inlet outlet ()", True)
    tui.SendCommand(" ///display/contour pressure 0 2", True)
    tui.SendCommand(" ///display/save-picture "{0}" """.format(img), True)
    tui.SendCommand(" ///surface/rake-surface rake-5 0 0 0 0 1 50", True)
    tui.SendCommand(" ///plot/plot-direction 0 0 1", True)
    tui.SendCommand(" ///plot/plot yes "{0}" yes yes no no pressure yes 0 0 1 rake-5 ()""".format(curve),
True)
    tui.SendCommand(" ///report/surface-integrals/area-weighted-avg inlet () pressure yes "{0}"
""".format(res), True)

    val = 0.
    n = 0;
    with System.IO.StreamReader(res) as reader:
        while n!=5:
            line = reader.ReadLine()
            if line!=None:
                n+=1
    str = line.Substring(32)
    val = System.Double.Parse(str, System.Globalization.CultureInfo.InvariantCulture)

    x=[]
    y=[]
    n = 0;
    with System.IO.StreamReader(curve) as reader:
        while n!=5:
            line = reader.ReadLine()
            if line!=None:
                n+=1
        while not line.StartsWith(" "):
            tab = line.Split("\t")
            x.Add(System.Double.Parse(tab[0], System.Globalization.CultureInfo.InvariantCulture))
            y.Add(System.Double.Parse(tab[1], System.Globalization.CultureInfo.InvariantCulture))
            line = reader.ReadLine()
    System.IO.File.Delete(res)
    System.IO.File.Delete(curve)

    step.NextStep.Properties["pressure"].Value = val
    step.NextStep.Attributes["img"] = img
    step.NextStep.Attributes["x"] = x
    step.NextStep.Attributes["y"] = y

def CreateReport(step):
    graph = step.UserInterface.GetComponent("Chart")
    graph.Title("Static Pressure")
    graph.ShowLegend(False)
    graph.Plot(step.Attributes["x"], step.Attributes["y"], key="Static Pressure", color='g')

    help = step.UserInterface.GetComponent("Report")
    help.SetHtmlContent(ExtAPI.Extension.InstallDir, """<center></center>""".format(step.Attributes["img"]))

```

Reviewing the Analysis

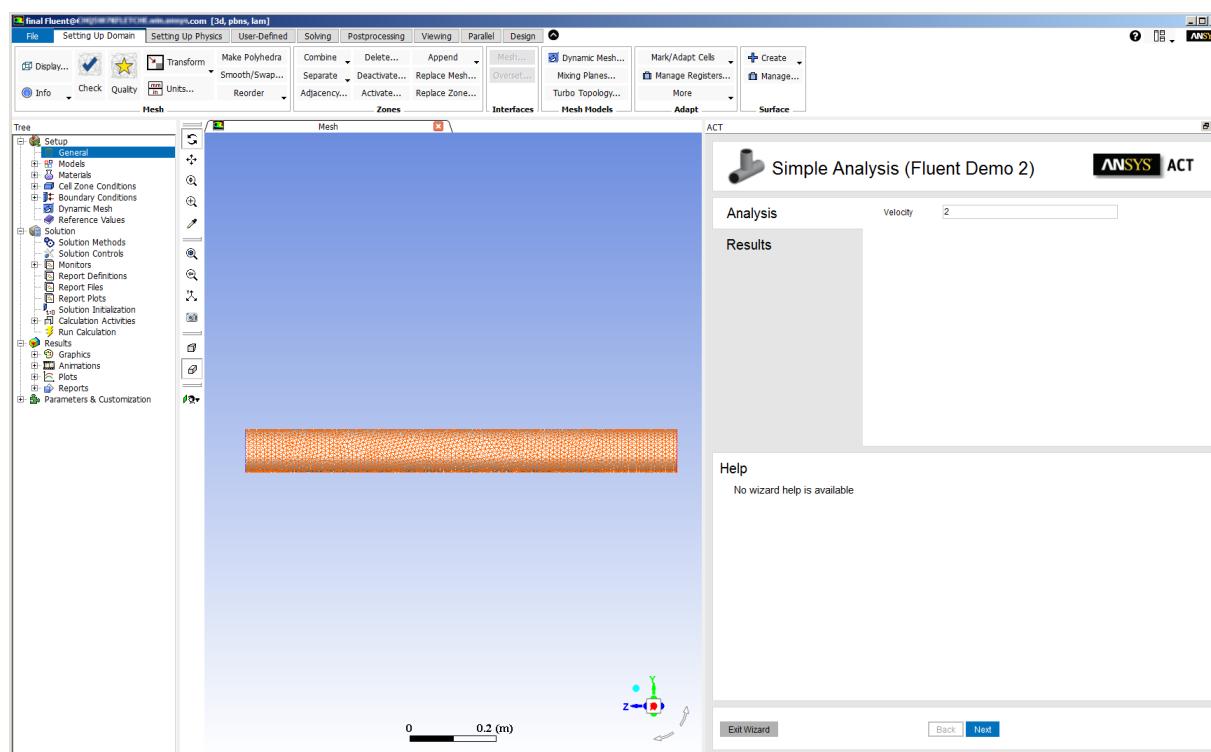
This section reviews the wizard interface and the processes performed in each step of the wizard **Simple Analysis (Fluent Demo 2)**.

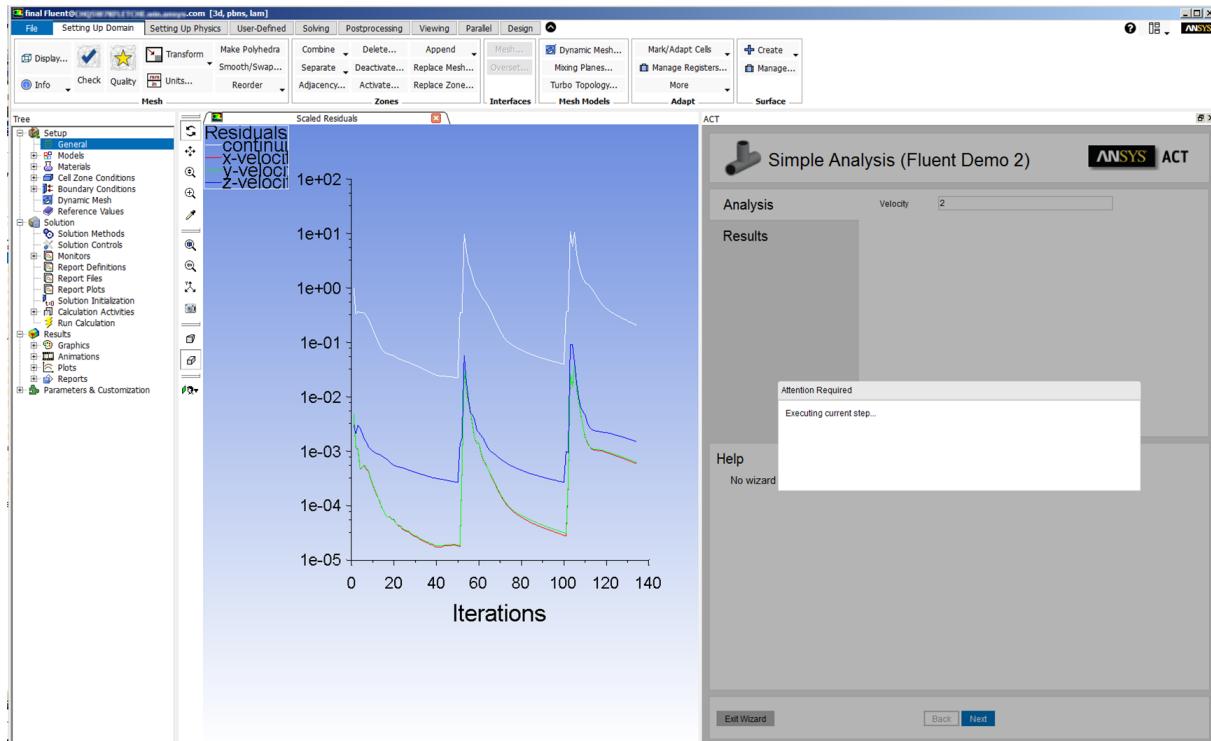
Reviewing the Analysis Step

The first step is **Analysis**, which imports the CAS file and creates the analysis. It requires the user to specify a value for the property **Velocity**.

In this step:

- The function **ImportModel** is invoked by the callback **<onrefresh>**. It imports the file `final.cas` and displays the model in the **Mesh** view.
- The function **CreateAnalysis** is invoked by the callback **<onupdate>** when the user clicks the step's **Next** button. This function sets up the analysis.
 - The property **Velocity** defined in the **<property>** block is shown in the GUI as a float field with a default value of **3**. The user can enter the desired value. In this example, a value of **2** is entered.
 - The additional actions to be performed as part of the analysis are defined.
 - Intermediate results are shown dynamically in the **Scaled Residuals** window as the analysis is performed.



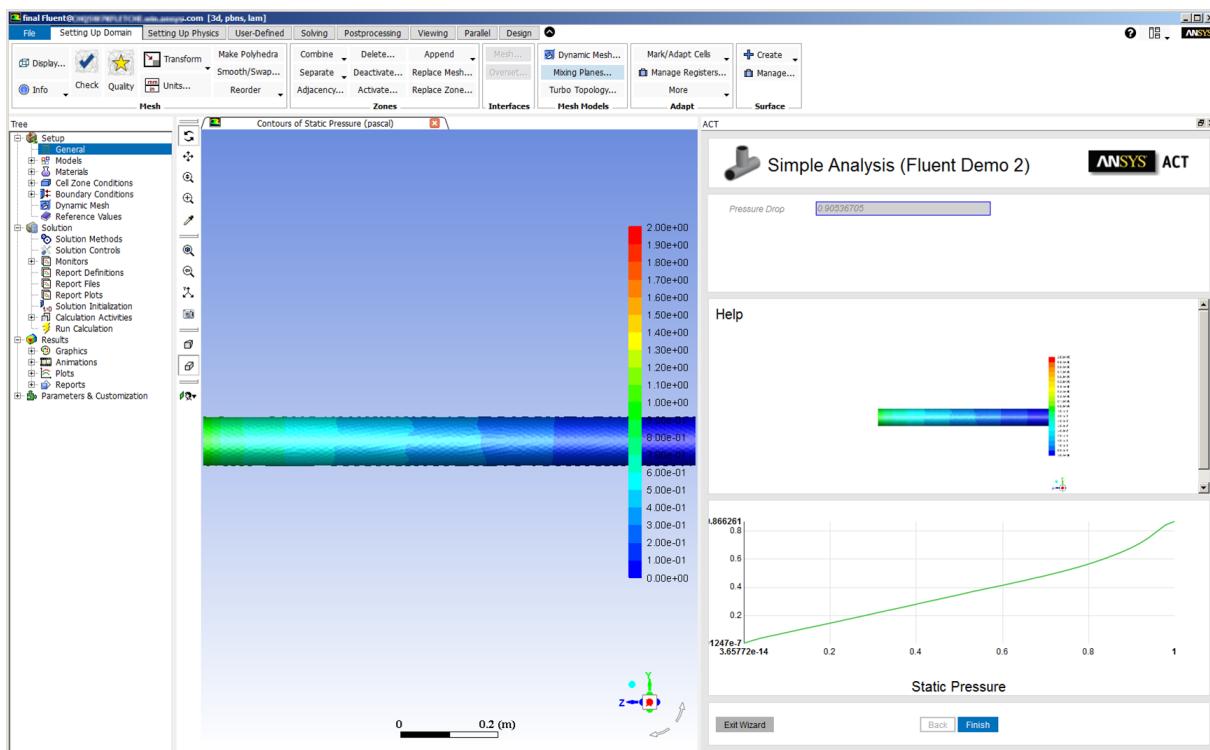


Reviewing the Results Step

The second step is **Results**, which creates a report of the results.

In this step:

- The function `CreateReport` is invoked by the callback `<onrefresh>`.
- The attribute `<layout>` specifies the custom layout to apply to the results. Custom layouts are defined in the `<uidefinition>` block.
- The wizard takes the results obtained and displays them:
 - The calculated value for the property `PressureDrop` defined in the `<property>` block is displayed in the GUI as a ready-only field.
 - The static pressure results are shown in the **Graph** and **Contours** views.



Electronics Desktop Wizard

The extension **Wizard Demo** contains a target product wizard for Electronics Desktop that has the same name as the extension. This simple two-step wizard enables the creation of two geometries from three dimension properties and the addition of two boundary conditions.

- In the first step, the first two dimensions define and create a rectangle. The first rectangle is then automatically duplicated and set at an offset along the Y axis, which is defined by the third dimension.
- In the second step, two boundary conditions are automatically created.

XML Extension Definition File

Definitions follow for codes blocks in the XML extension definition file **wizardDemo.xml**.

Extension Definition

The **<extension>** block is the top-most level in the hierarchy.

- The attributes **name** and **version** specify the extension name and version.
- The attribute **icon** specifies the image file to display as the icon for the extension. You specify the location of the **images** folder for all images used by the extension in the **<interface>** block.

Guid Definition

The **<guid>** block specifies a unique identifier for the extension.

Script Definition

The **<script>** block specifies the IronPython script that is referenced by the extension. This extension references **callbacks.py**.

Wizard Interface Definition

The `<interface>` block defines the user interface for the wizard.

- The attribute `context` is set to `ElectronicsDesktop` because the extension runs from ANSYS Electronics Desktop.
- The attribute `images` specifies the `images` folder for all images that the extension is to use. You use the attribute `icon` for the extension, wizard, step, and so on to specify an image file in this folder to use as an icon.

Wizard Definition

The wizard `Wizard Demo` and its steps are defined in the `<wizard>` block named `Wizard Demo`.

- The attributes `name` and `version` specify the wizard name and version.
- The attribute `context` specifies the ANSYS product in which the wizard is executed. Because the wizard runs from Electronic Desktop, `context` is set to `ElectronicsDesktop`.
- The attribute `icon` specifies the filename of the image to display as the icon for the wizard: `antimage`. The file `antimage.png` is stored in the folder specified for the attribute `images` in the `<interface>` block.
- The attribute `description` specifies the text to display in the **About** window for the extension when it is accessed from the **Extension Manager** and in the **About** window for the wizard when it is accessed from the **Wizards** page.

Step Definition

The `<step>` blocks define the steps of the wizard. This wizard has two steps: `Step1` and `Step2`. For each step:

- The attributes `name`, `version`, and `caption` define the step name, version, and display text for the step. These are shown in the **About** window for the wizard when it is accessed from the **Wizards** page.
- The attribute `context` specifies the ANSYS product in which the specific step is executed. You only need to set the context for the step if it differs from the context for the wizard.
- The attribute `HelpFile` specifies the HTML file to display as the custom help for the step.
- The `<propertygroup>` blocks specify groupings of properties within the step. Any `<property>` blocks specify properties and property attributes.
- The `<callbacks>` blocks specify callbacks to functions defined in the script `callbacks.py`. These are discussed in more detail in [Reviewing the Analysis \(p. 369\)](#).

The XML extension definition file `WizardDemo.xml` follows.

```
<extension name="Wizard Demo" version="1">
  <author>ANSYS Inc.</author>
  <guid shortid="ElectronicsWizard">819d1fd2-17c2-493a-ad6a-ee608356229a</guid>
  <script src="callbacks.py" />

  <interface context="ElectronicsDesktop">
    <images>images</images>
  </interface>

  <wizard name="Wizard Demo" caption="Wizard Demo" version="1" context="ElectronicsDesktop" icon="antImage">
    <description>Simple example of a wizard in Electronics Desktop</description>
```

```

<step name="Step1" version="1" caption="Design">
<description>Create geometry</description>

    <propertygroup name="dim" caption="Dimensions (mm)" persistent="False" parameterizable="False">
        <property control="float" name="width" caption="Width" persistent="False" parameterizable="False" default="0.4" />
        <property control="float" name="height" caption="Height" persistent="False" parameterizable="False" default="1.7" />
    </propertygroup>

    <propertygroup name="offset" caption="Offset for duplication (mm)" persistent="False" parameterizable="False">
        <property control="float" name="offset" caption="Offset" persistent="False" parameterizable="False" default="0.3" />
    </propertygroup>

    <callbacks>
        <onupdate>OnUpdate1</onupdate>
        <onreset>OnReset1</onreset>
    </callbacks>

</step>

<step name="Step2" version="1" caption="Setup">
<description>Define boundary conditions</description>

    <propertygroup name="bc" caption="Boundary Conditions (Ohm)" persistent="False" parameterizable="False">
        <property control="float" name="resistance" caption="Full Resistance" persistent="False" parameterizable="False" default="50." />
        <property control="float" name="reactance" caption="Full Reactance" persistent="False" parameterizable="False" default="0." />
    </propertygroup>

    <callbacks>
        <onupdate>OnUpdate2</onupdate>
    </callbacks>

</step>

</wizard>
</extension>

```

IronPython Script

The IronPython script `callbacks.py` follows. This script defines all functions executed by the callbacks in the XML extension definition file.

```

# -----
# Callbacks
# -----
oDesign = None
oProject = None

def OnUpdate1(step):
    global oDesign, oProject
    oProject = oDesktop.NewProject()
    oProject.InsertDesign("HFSS", "HFSSDesign1", "DrivenModal", "")
    oDesign = oProject.SetActiveDesign("HFSSDesign1")
    oEditor = oDesign.SetActiveEditor("3D Modeler")

    width = step.Properties["dim/width"].Value
    height = step.Properties["dim/height"].Value
    offset = step.Properties["offset/offset"].Value

    oEditor.CreateRectangle(
        [
            "NAME:RectangleParameters",
            "IsCovered:=" , True,
            "XStart:=" , "0mm",
            "YStart:=" , "0mm",
            "Width:=" , width,
            "Height:=" , height
        ]
    )

```

```

        "YStart:=" , "0mm",
        "ZStart:=" , "0mm",
        "Width:=" , str(width)+"mm",
        "Height:=" , str(height)+"mm",
        "WhichAxis:=" , "Z"
    ],
[
    "NAME:Attributes",
    "Name:=" , "Rectangle1",
    "Flags:=" , "",
    "Color:=" , "(128 255 255)",
    "Transparency:=" , 0.800000011920929,
    "PartCoordinateSystem:=" , "Global",
    "UDMID:=" , "",
    "MaterialValue:=" , "\"vacuum\"",
    "SolveInside:=" , True
])
oEditor.DuplicateAlongLine(
[
    "NAME:Selections",
    "Selections:=" , "Rectangle1",
    "NewPartsModelFlag:=" , "Model"
],
[
    "NAME:DuplicateToAlongLineParameters",
    "CreateNewObjects:=" , True,
    "XComponent:=" , "0mm",
    "YComponent:=" , str(height+offset)+"mm",
    "ZComponent:=" , "0mm",
    "NumClones:=" , "2"
],
[
    "NAME:Options",
    "DuplicateAssignments:=" , False
])

def OnReset1(step):
    global oProject
    oDesktop.CloseProject(oProject.GetName())

def OnUpdate2(step):
    global oDesign
    oModule = oDesign.GetModule("BoundarySetup")

    width = step.PreviousStep.Properties["dim/width"].Value
    height = step.PreviousStep.Properties["dim/height"].Value
    offset = step.PreviousStep.Properties["offset/offset"].Value

    resistance = step.Properties["bc/resistance"].Value
    reactance = step.Properties["bc/reactance"].Value

    oModule.AssignPerfectE(
    [
        "NAME:PerfE1",
        "Objects:=" , ["Rectangle1_1","Rectangle1"],
        "InfGroundPlane:=" , False
    ])
    oEditor = oDesign.SetActiveEditor("3D Modeler")
    oEditor.CreateRectangle(
    [
        "NAME:RectangleParameters",
        "IsCovered:=" , True,
        "XStart:=" , "0mm",
        "YStart:=" , str(height)+"mm",
        "ZStart:=" , "0mm",
        "Width:=" , str(width)+"mm",
        "Height:=" , str(offset)+"mm",
        "WhichAxis:=" , "Z"
    ],
    [
        "NAME:Attributes",
        "Name:=" , "Rectangle2",
        "Flags:=" ,
        "Color:=" , "(128 255 255)",
        "Transparency:=" , 0.800000011920929,
        "PartCoordinateSystem:=" , "Global",
        "UDMID:=" , "",
        "MaterialValue:=" , "\"vacuum\"",
        "SolveInside:=" , True
    ])
)

```

```

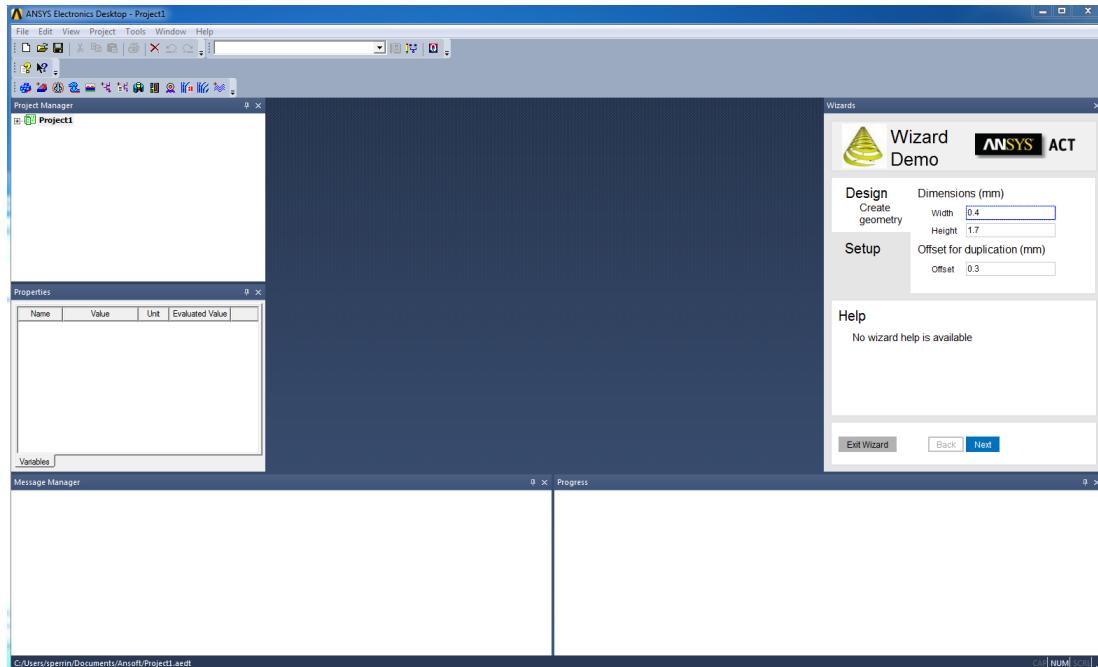
    "Flags:="           ,   "",
    "Color:="          , "(128 255 255)",
    "Transparency:="   , 0.800000011920929,
    "PartCoordinateSystem:=" , "Global",
    "UDMID:="          ,   "",
    "MaterialValue:="   , "\"vacuum\"",
    "SolveInside:="     , True
  ])
oModule.AssignLumpedPort(
[
  "NAME:1",
  "Objects:="         , [ "Rectangle2" ],
  "RenormalizeAllTerminals:=" , True,
  "DoDeembed:="        , False,
  [
    "NAME:Modes",
    [
      "NAME:Model",
      "ModeNum:="       , 1,
      "UseIntLine:="    , True,
      [
        "NAME:IntLine",
        "Start:="        , [ str(width/2.)+"mm",str(height)+"mm", "0mm" ],
        "End:="          , [ str(width/2.)+"mm",str(height+offset)+"mm", "0mm" ]
      ],
      "AlignmentGroup:=" , 0,
      "CharImp:="        , "Zpi",
      "RenormImp:="      , "50ohm"
    ]
  ],
  "ShowReporterFilter:=" , False,
  "ReporterFilter:="    , [True],
  "FullResistance:="    , str(resistance)+"ohm",
  "FullReactance:="     , str(reactance)+"ohm"
])
)

```

Reviewing the Analysis

This section reviews the wizard interface and the processes performed in each step of the wizard **Wizard Demo**.

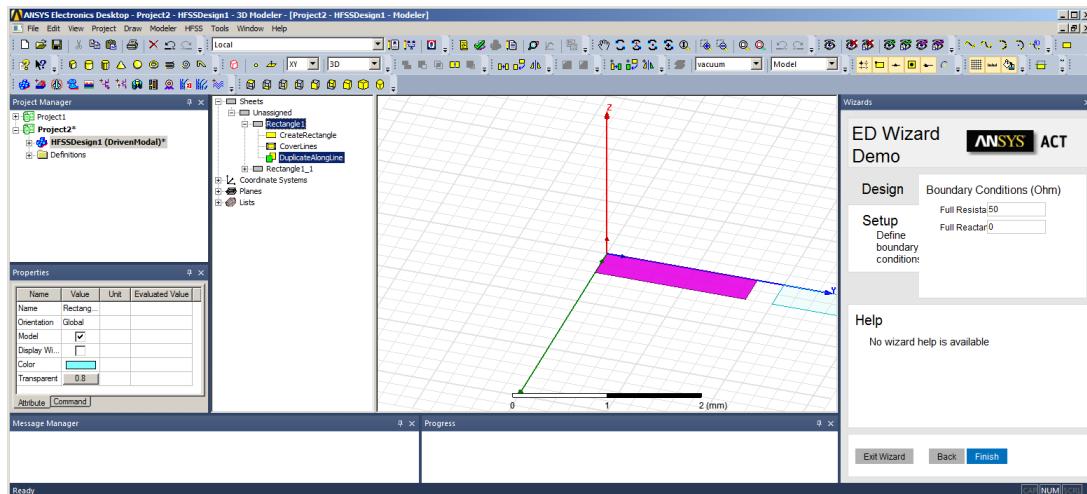
The first step is **Design**, which defines the geometries.



In this step:

- The two `<propertygroup>` blocks define the three dimension properties.
 - The first block defines the property group **Dimensions** to include the properties **Width** and **Height** for creating the first rectangle.
 - The second block defines the property **Offset for duplication** to use for duplicating the first rectangle.
- The `<callbacks>` block defines the following callbacks:
 - The callback `<onupdate>` invokes the function `OnUpdate1` when the **Next** button is clicked. This function creates the new project, creates the first rectangle based on the entered dimensions, and then creates the second rectangle via duplication and offset according to the entered offset value.
 - The callback `<onreset>` is invoked when the **Back** button on the next step is clicked. The **Back** button is enabled only if the callback `<onreset>` has been defined.

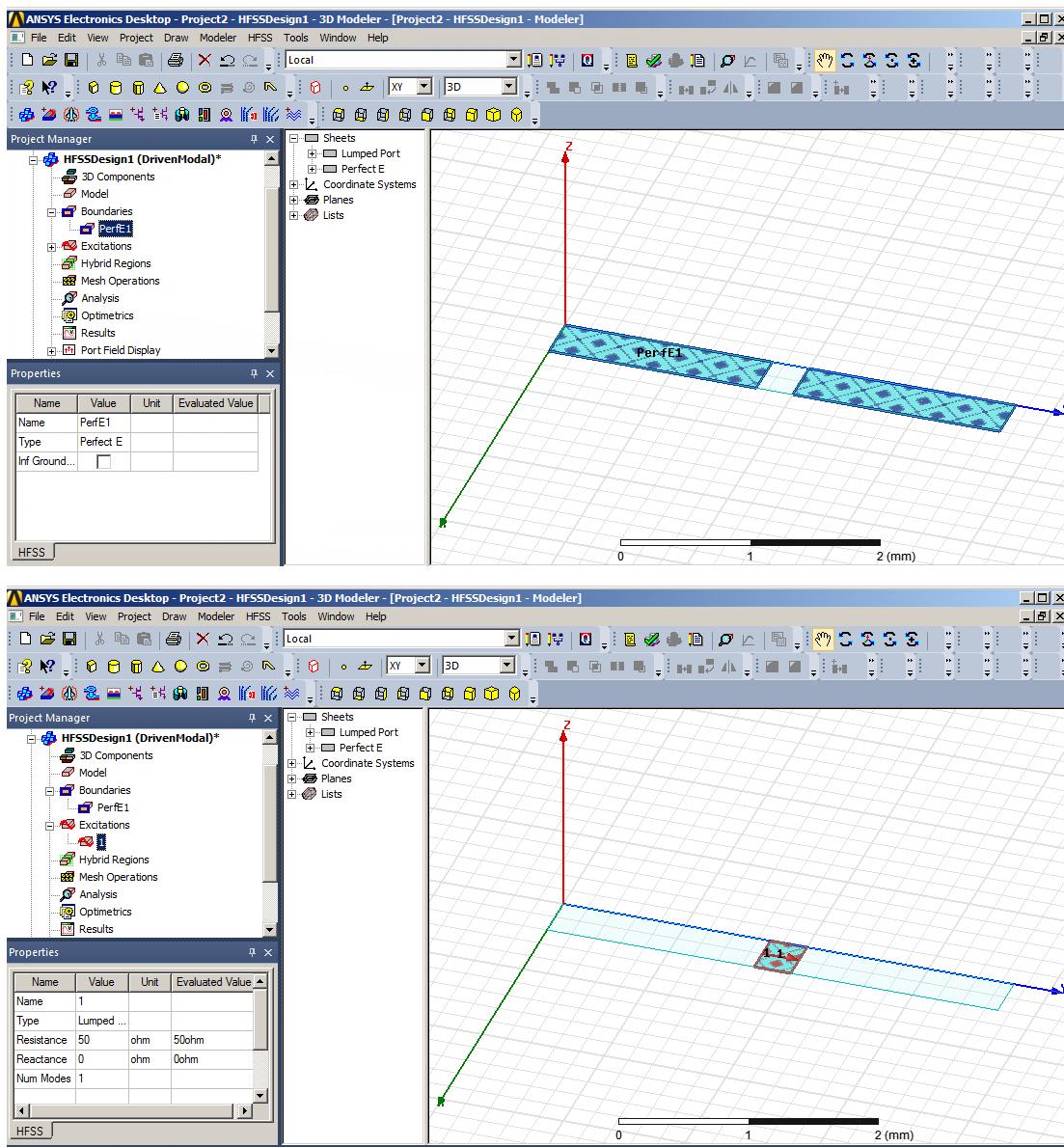
The next step is **Setup**, which defines and applies the boundary conditions.



In this step:

- The `<propertygroup>` block defines two boundary condition properties: **resistance** and **reactance**. The attribute **caption** for these two properties are **Full Resistance** and **Full Reactance**.
- The callback `<onupdate>` invokes the function `OnUpdate2` when the **Next** button is clicked. This function applies the boundary conditions to both rectangles.

Once the wizard has completed, the boundary conditions can be viewed by selecting them in the Electronics Desktop **Project Manager** tree.



SpaceClaim Wizard

The extension **SC_BGA_Extension** contains a product target wizard for SpaceClaim named **BGAWizard.ard**.

XML Extension Definition File

Definitions follow for codes blocks in the XML extension definition file **SC_BGAExtension.xml**.

Extension Definition

The **<extension>** block is the top-most level in the hierarchy.

- The attributes **name** and **version** specify the extension name and version.
- The attribute **icon** specifies the image file to display as the icon for the extension. You specify the location of the **images** folder for all images used by the extension in the **<interface>** block.

Guid Definition

The `<guid>` block specifies a unique identifier for the extension.

Script Definition

The `<script>` block specifies the IronPython script that is referenced by the extension. This extension references `main.py`.

Wizard Interface Definition

The `<interface>` block defines the user interface for the wizard.

- The attribute `context` is set to `SpaceClaim` because the extension runs from ANSYS SpaceClaim.
- The attribute `images` specifies the `images` folder for all images that the extension is to use. You use the attribute `icon` for the extension, wizard, step, and so on to specify an image file in this folder to use as an icon.
- The `<toolbar>` block defines a toolbar and toolbar button for exposure in SpaceClaim.

Wizard Definition

The wizard `BGAWizard` and its steps are defined in the `<wizard>` block.

- The attributes `name` and `version` specify the wizard name and version.
- The attribute `context` specifies the ANSYS product in which the wizard is executed. Because the wizard runs from SpaceClaim, `context` is set to `SpaceClaim`.
- The attribute `description` specifies the text to display in the **About** window for the extension when it is accessed from the **Extension Manager** and in the **About** window for the wizard when it is accessed from the **Wizards** page.

Step Definition

The `<step>` block defines the steps of the wizard. This wizard has three steps: `Die`, `SubstrateAndSolderMask`, and `SolderBall`.

For each step:

- The attributes `name`, `version`, and `caption` define the step name, version, and display text for the step.
- The attribute `context` specifies the ANSYS product in which the specific step is executed. You only need to set the context for the step if it differs from the context for the wizard.
- The attribute `HelpFile` specifies the HTML file to display as the custom help for the step.
- The `<propertygroup>` blocks specify properties and property attributes for the step.
- The `<callbacks>` blocks specify callbacks to functions defined in the script `main.py`.
 - For the step `Die`, the callback `<onupdate>` invokes the function `GenerateDie`.
 - For the step `SubstrateAndSolderMask`, the callback `<onupdate>` invokes the function `GenerateSubstrateAndSolderMask`.

- For the step **SolderBall**, the callback **<onupdate>** invokes the function **GenerateBalls**.

Note

Beginning in release 18.2, the graphics window in SpaceClaim is not updated until the callbacks for a step have been executed. This change ensures graphics stability and better performance.

The XML extension definition file `SC_BGA_Extension.xml` follows.

```

<extension version="1" name="SC_BGA_Extension">
  <script src="main.py" />
  <guid shortid="SC_BGA_Extension">5107C33A-E123-4F55-8166-2ED2AA59B3B2</guid>
  <interface context="SpaceClaim">
    <images>images</images>

    <callbacks>
      <oninit>oninit</oninit>
    </callbacks>

    <toolbar name="SC_BGA_Extension" caption="SC_BGA Extension">
      <entry name="SC_BGA_Package" icon="icepak_package">
        <callbacks>
          <onclick>createMyFeature</onclick>
        </callbacks>
      </entry>
    </toolbar>
  </interface>

  <wizard name="BGAWizard" version="1" context="SpaceClaim">
    <description>BGA Wizard</description>

    <step name="Die" caption="Die" version="1">
      <callbacks><onupdate>GenerateDie</onupdate></callbacks>
      <property name="Thickness" caption="Height" unit="Length" control="float" default="0.3[mm]" />
      <property name="Width" caption="Width" unit="Length" control="float" default="5 [mm]" />
    </step>

    <step name="SubstrateAndSolderMask" caption="Substrate and SolderMask" version="1">
      <callbacks><onupdate>GenerateSubstrateAndSolderMask</onupdate></callbacks>
      <propertygroup name="SubstrateDetails" caption="SubstrateDetails" display="caption">
        <property name="Thickness" caption="Thickness" unit="Length" control="float" default="0.4 [mm]" />
      </propertygroup>
      <propertygroup name="SolderMaskDetails" caption="SolderMaskDetails" display="caption">
        <property name="Height" caption="Solder Mask Height" unit="Length" control="float" default="0.05 [mm]" />
      </propertygroup>
    </step>

    <step name="SolderBall" caption="Solder ball" version="1">
      <callbacks><onupdate>GenerateBalls</onupdate></callbacks>
      <property name="Face" caption="Face" control="scoping">
        <attributes selection_filter="face"/>
      </property>

      <propertygroup name="SolderBallDetails" caption="Solder Ball Details" display="caption">
        <propertygroup display="property" name="BallsPrimitive" caption="Balls primitive" control="select" default="sphere">
          <attributes options="sphere,cylinder,cone,cube,gear"/>
        </propertygroup>
        <property name="Pitch" caption="Pitch" unit="Length" control="float" default="0.8 [mm]" />
        <property name="Radius" caption="Radius" unit="Length" control="float" default="0.35 [mm]" />
        <property name="Number of Solder Ball Columns" caption="No of Solder Ball Columns" control="integer" default="16" />
        <property name="Number of Solder Ball Rows" caption="No of Solder Ball Rows" control="integer" default="16" />
      </propertygroup>
    </step>
  </wizard>

```

```
</propertygroup>

<propertygroup name="Central Balls" caption="Central Thermal Balls">
<propertygroup display="property" name="Central Thermal Balls" caption="Want to Supress Central
Balls" control="select" default="No">
<attributes options="Yes,No"/>
<property name="Number of Solder Ball Columns" caption="No of Solder Ball Columns" control="integer"
default="4" visible = "false" visibleon="Yes"/>
<property name="Number of Solder Ball Rows" caption="No of Solder Ball Rows" control="integer"
default="4" visible = "false" visibleon="Yes"/>
</propertygroup>
</propertygroup>
</step>

</wizard>

</extension>
```

IronPython Script

The IronPython script main.py follows. This script defines all the functions executed by the callbacks in the XML extension definition file. Each step defined in the XML file can include multiple actions.

```
import System
import clr
import sys
import os
import math

part = None

def oninit(context):
    return

def createMyFeature(ag):
    ExtAPI.CreateFeature("MyFeature1")

def createSphere(x, y, z, radius):
    global part
    from System.Collections.Generic import List

    # get selected part
    if part==None:
        win = Window.ActiveWindow
        context = win.ActiveContext
        part = context.ActivePart.Master

    center = Geometry.Point.Create(x, y, z)
    profileFrame = Geometry.Frame.Create(center, Geometry.Direction.DirX, Geometry.Direction.DirY)
    sphereCircle = Geometry.Circle.Create(profileFrame, radius)
    sphereRevolveLine = Geometry.Line.Create(center, Geometry.Direction.DirX)
    profile = List[Geometry.ITrimmedCurve]()
    profile.Add(Geometry.CurveSegment.Create(sphereCircle, Geometry.Interval.Create(0, math.pi)));
    profile.Add(Geometry.CurveSegment.Create(sphereRevolveLine, Geometry.Interval.Create(-radius, radius)))
    path = List[Geometry.ITrimmedCurve]()
    sweepCircle = Geometry.Circle.Create(Geometry.Frame.Create(center, Geometry.Direction.DirY,
    Geometry.Direction.DirZ), radius)
    path.Add(Geometry.CurveSegment.Create(sweepCircle))
    body = Modeler.Body.SweepProfile(Geometry.Profile(Geometry.Plane.Create(profileFrame), profile), path)
    DesignBody.Create(part, "sphere", body)

def createCylinder(x, y, z, radius, h):
    global part
    from System.Collections.Generic import List

    # get selected part
    if part==None:
        win = Window.ActiveWindow
        context = win.ActiveContext
        part = context.ActivePart.Master
```

```

defaultPointUV = Geometry.PointUV.Create(0, 0)
profile = Geometry.CircleProfile(Geometry.Plane.PlaneXY, radius, defaultPointUV, 0)

points = List[Geometry.Point]()
points.Add(Geometry.Point.Create(0, 0, 0))
points.Add(Geometry.Point.Create(0, 0, h))
path = Geometry.PolygonProfile(Geometry.Plane.PlaneXY, points)

body = Modeler.Body.SweepProfile(profile, path.Boundary)
designBody = DesignBody.Create(part, "Cylinder", body)

translation = Geometry.Matrix.CreateTranslation(Geometry.Vector.Create(x, y, z))
designBody.Transform(translation)

def createCone(x, y, z, radius, h):
    global part
    from System.Collections.Generic import List

    # get selected part
    if part==None:
        win = Window.ActiveWindow
        context = win.ActiveContext
        part = context.ActivePart.Master

    defaultPointUV = Geometry.PointUV.Create(0, 0)
    path = Geometry.CircleProfile(Geometry.Plane.PlaneXY, radius, defaultPointUV, 0)

    points = List[Geometry.Point]()
    points.Add(Geometry.Point.Create(0, 0, 0))
    points.Add(Geometry.Point.Create(radius, 0, h))
    points.Add(Geometry.Point.Create(0, 0, h))
    triangle = Geometry.PolygonProfile(Geometry.Plane.PlaneZX, points)

    body = Modeler.Body.SweepProfile(triangle, path.Boundary)
    designBody = DesignBody.Create(part, "Cone", body)

    translation = Geometry.Matrix.CreateTranslation(Geometry.Vector.Create(x, y, z))
    designBody.Transform(translation)

def createBox(xa, ya, za, xb, yb, zb):
    global part
    # get selected part
    if part==None:
        win = Window.ActiveWindow
        context = win.ActiveContext
        part = context.ActivePart.Master

    lengthX = xb - xa
    lengthY = yb - ya
    lengthZ = zb - za
    xa = xa + lengthX * 0.5
    ya = ya + lengthY * 0.5

    p = Geometry.PointUV.Create(0, 0)
    body = Modeler.Body.ExtrudeProfile(Geometry.RectangleProfile(Geometry.Plane.PlaneXY, lengthX,
lengthY, p, 0), lengthZ)
    designBody = DesignBody.Create(part, "body", body)

    translation = Geometry.Matrix.CreateTranslation(Geometry.Vector.Create(xa, ya, za))
    designBody.Transform(translation)

def createGear(x, y, z, innerRadius, outerRadius, width, count, holeRadius):
    global part
    from System.Collections.Generic import List

    # get selected part
    if part==None:
        win = Window.ActiveWindow
        context = win.ActiveContext
        part = context.ActivePart.Master
    frame = Geometry.Frame.World

```

```

# create gear
outsideCircle = Geometry.Circle.Create(frame, outerRadius);
insideCircle = Geometry.Circle.Create(frame, innerRadius);

boundary = List[Geometry.ITrimmedCurve]()
inwardLine = Geometry.Line.Create(frame.Origin, -frame.DirX);
outwardLine = Geometry.Line.Create(frame.Origin, frame.DirX);
axis = outsideCircle.Axis;

nTeeth = count;
repeatAngle = 2 * math.pi / nTeeth;
toothAngle = 0.6 * repeatAngle;
gapAngle = repeatAngle - toothAngle;

for i in range(0, nTeeth):
    # an arc is just a parameter interval of a circle
    startTooth = i * repeatAngle;
    endTooth = startTooth + toothAngle;
    boundary.Add(Geometry.CurveSegment.Create(outsideCircle, Geometry.Interval.Create(startTooth,
endTooth)));

    # rotate 'inwardLine' about the circle axis
    rotatedInwardLine = Geometry.Matrix.CreateRotation(axis, endTooth) * inwardLine;
    # a line segment is just a parameter interval of an unbounded line
    boundary.Add(Geometry.CurveSegment.Create(rotatedInwardLine, Geometry.Interval.Create(-outerRadius,
-innerRadius)));

    startGap = endTooth;
    endGap = startGap + gapAngle;
    boundary.Add(Geometry.CurveSegment.Create(insideCircle, Geometry.Interval.Create(startGap,
endGap)));

    rotatedOutwardLine = Geometry.Matrix.CreateRotation(axis, endGap) * outwardLine;
    boundary.Add(Geometry.CurveSegment.Create(rotatedOutwardLine, Geometry.Interval.Create(
innerRadius, outerRadius)));

hole = Geometry.Circle.Create(frame.Create(frame.Origin, frame.DirX, frame.DirY), holeRadius);
boundary.Add(Geometry.CurveSegment.Create(hole));

body = Modeler.Body.ExtrudeProfile(Geometry.Profile(Geometry.Plane.Create(frame), boundary), width);
pieces = body.SeparatePieces().GetEnumerator()
while pieces.MoveNext():
    designBody = DesignBody.Create(part, "GearBody", pieces.Current);

    translation = Geometry.Matrix.CreateTranslation(Geometry.Vector.Create(x, y, z))
    designBody.Transform(translation)

class Vector:
    def __init__(self, x = 0, y = 0, z = 0):
        self.x = x
        self.y = y
        self.z = z

    def Clone(self):
        return Vector(self.x, self.y, self.z)

    def NormSQ(self):
        return self.x*self.x + self.y*self.y + self.z*self.z

    def Norm(self):
        return math.sqrt(self.x*self.x + self.y*self.y + self.z*self.z)

    def Normalize(self):
        norm = self.Norm()
        self.x = self.x / norm
        self.y = self.y / norm
        self.z = self.z / norm

    def GetNormalize(self):
        norm = self.Norm(self)
        return Vector(self.x / norm, self.y / norm, self.z / norm)

```

```

def __add__(va, vb):
    return Vector(va.x + vb.x, va.y + vb.y, va.z + vb.z)

def __sub__(va, vb):
    return Vector(va.x - vb.x, va.y - vb.y, va.z - vb.z)

def __mul__(v, x):
    return Vector(v.x*x, v.y*x, v.z*x)

def Cross(va, vb):
    return Vector(va.y*vb.z - va.z*vb.y, -va.z*vb.x + va.x*vb.z, va.x*vb.y - va.y*vb.x)

def Dot(va, vb):
    return va.x*vb.x + va.y*vb.y + va.z*vb.z

def ToString(self):
    return "( " + str(self.x) + ", " + str(self.y) + ", " + str(self.z) + " )"

def CreateBalls(primitive, pitch, radius, column, row, supr, columnSupr, rowSupr, center, dirColumn, dirRow):
    dirColumn.Normalize()
    dirRow.Normalize()
    startVector = center - dirColumn*column*pitch*0.5 - dirRow*row*pitch*0.5
    startVector = startVector + dirColumn*radius + dirRow*radius
    startVector = startVector + dirRow.Cross(dirColumn)*radius
    stepVectorColumn = dirColumn * pitch
    stepVectorRow = dirRow * pitch

    if(supr == "Yes"):
        column_index_to_start_supress = int( column * 0.5 - columnSupr * 0.5 )
        row_index_to_start_supress = int( row * 0.5 - rowSupr * 0.5 )

    v = startVector.Clone()
    for i in range(column):
        for j in range(row):
            createBall = False
            if (supr == "Yes" and (i < column_index_to_start_supress or
                                   i >= column_index_to_start_supress + columnSupr or
                                   j < row_index_to_start_supress or
                                   j >= row_index_to_start_supress+ rowSupr)
                or supr == "No"):
                if primitive == "sphere":
                    createSphere(v.x, v.y, v.z, radius)
                elif primitive == "cylinder":
                    createCylinder(v.x, v.y, v.z, radius, radius * 2.)
                elif primitive == "cone":
                    createCone(v.x, v.y, v.z, radius, radius * 2.)
                elif primitive == "cube":
                    createBox(v.x - radius, v.y - radius, v.z - radius,
                              v.x + radius, v.y + radius, v.z + radius)
                elif primitive == "gear":
                    createGear(v.x, v.y, v.z,
                               radius*0.5, radius, radius*2, 10, radius*0.2)
            v = v + stepVectorRow

    v = startVector.Clone()
    startVector = startVector + stepVectorColumn
    v = v + stepVectorColumn

def CreateDie(width, thickness, zStart):
    createBox(-0.5 * width, -0.5 * width, zStart,
              0.5 * width, 0.5 * width, zStart + thickness)

def CreateSubstrate(width, thickness, zStart):
    createBox(-0.5 * width, -0.5 * width, zStart,
              0.5 * width, 0.5 * width, zStart + thickness)

def CreateSolderMask(width, thickness, zStart):
    createBox(-0.5 * width, -0.5 * width, zStart,
              0.5 * width, 0.5 * width, zStart + thickness)

def generateBGAGeometry(feature,fct):
    ps = feature.Properties

```

```

Pitch = ps["Solder Ball Details/Pitch"].Value
Solder_Ball_Radius = ps["Solder Ball Details/Solder Ball Radius"].Value
No_Of_Solder_Ball_Column = ps["Solder Ball Details/Number of Solder Ball
Columns"].Value
No_Of_Solder_Ball_Row = ps["Solder Ball Details/Number of Solder Ball Rows"].Value
No_Of_Solder_Ball_Column_Supress = ps["Central Balls/Central Thermal Balls/Number of Solder Ball
Columns"].Value
No_Of_Solder_Ball_Row_Supress = ps["Central Balls/Central Thermal Balls/Number of Solder Ball
Rows"].Value
Substrate_Thickness = ps["Substrate Details/Substrate Thickness"].Value
Substrate_Width = ps["Substrate Details/Substrate Length"].Value
Die_Thickness = ps["Die Details/Die Thickness"].Value
Die_Width = ps["Die Details/Die Width"].Value
Solder_Mask_Height = ps["Solder Ball Details/Solder Mask Height"].Value
supress_balls = ps["Central Balls/Central Thermal Balls"].Value
ballsPrimitive = ps["BallsPrimitive"].Value

bodies = []

CreateBalls(ballsPrimitive, Pitch, Solder_Ball_Radius, No_Of_Solder_Ball_Column,
No_Of_Solder_Ball_Row, supress_balls,
           No_Of_Solder_Ball_Column_Supress, No_Of_Solder_Ball_Row_Supress,
           Vector(0, 0, 0), Vector(1, 0, 0), Vector(0, 1, 0))

#Creating Substrate and soldermask
CreateSubstrate(Substrate_Width, Substrate_Thickness, 0)
CreateSolderMask(Substrate_Width, Solder_Mask_Height, 0)

#Creating Die
Die_Start = Substrate_Thickness
CreateDie(Die_Width, Die_Thickness, Die_Start)

return True

def GenerateDie(step):
    global part
    win = Window.ActiveWindow
    context = win.ActiveContext
    part = context.ActivePart

    ps = step.Properties
    Die_Thickness = ps["Thickness"].Value
    Die_Width = ps["Width"].Value
    CreateDie(Die_Width, Die_Thickness, 0)

    part = None

def GenerateSubstrateAndSolderMask(step):
    global part
    win = Window.ActiveWindow
    context = win.ActiveContext
    part = context.ActivePart

    Die_Thickness = step.PreviousStep.Properties["Thickness"].Value

    ps = step.Properties
    Substrate_Thickness = ps["SubstrateDetails/Thickness"].Value
    Substrate_Width = ps["SubstrateDetails/Length"].Value
    Solder_Mask_Height = ps["SolderMaskDetails/Height"].Value

    CreateSubstrate(Substrate_Width, Substrate_Thickness, Die_Thickness)
    CreateSolderMask(Substrate_Width, Solder_Mask_Height, Die_Thickness + Substrate_Thickness)

    part = None

def GenerateBalls(step):
    global part
    win = Window.ActiveWindow
    context = win.ActiveContext
    part = context.ActivePart

```

```

zStart = 0
zStart += step.PreviousStep.PreviousStep.Properties["Thickness"].Value
zStart += step.PreviousStep.Properties["SubstrateDetails/Thickness"].Value
zStart += step.PreviousStep.Properties["SolderMaskDetails/Height"].Value

ps = step.Properties
faces      = ps["Face"].Value.Faces
pitch       = ps["SolderBallDetails/Pitch"].Value
radius      = ps["SolderBallDetails/Radius"].Value
column     = ps["SolderBallDetails/Number of Solder Ball Columns"].Value
row        = ps["SolderBallDetails/Number of Solder Ball Rows"].Value
primitive   = ps["SolderBallDetails/BallsPrimitive"].Value
columnSupr = ps["Central Balls/Central Thermal Balls/Number of Solder Ball Columns"].Value
rowSupr    = ps["Central Balls/Central Thermal Balls/Number of Solder Ball Rows"].Value
supr       = ps["Central Balls/Central Thermal Balls"].Value

for i in range(0, faces.Count):
    face = faces[i]
    edges = face.Edges
    if edges.Count == 0:
        continue

    # find two edges with a common point
    edgeA = edges[0]
    startPointA = edgeA.Shape.StartPoint
    endPointA   = edgeA.Shape.EndPoint
    for j in range(1, edges.Count):
        edgeB = edges[j]
        startPointB = edgeB.Shape.StartPoint
        endPointB   = edgeB.Shape.EndPoint

        if startPointB == startPointA:
            basePoint   = startPointB
            pointRow   = endPointA
            pointColumn = endPointB
        elif endPointB == startPointA:
            basePoint   = endPointB
            pointRow   = endPointA
            pointColumn = startPointB
        elif startPointB == endPointA:
            basePoint   = startPointB
            pointRow   = startPointA
            pointColumn = endPointB
        elif endPointB == endPointA:
            basePoint   = endPointB
            pointRow   = startPointA
            pointColumn = startPointB

        if not basePoint is None:
            dirColumn = Vector(pointRow.X - basePoint.X, pointRow.Y - basePoint.Y, pointRow.Z -
basePoint.Z)
            dirRow    = Vector(pointColumn.X - basePoint.X, pointColumn.Y - basePoint.Y, pointColumn.Z -
basePoint.Z)
            center    = Vector(basePoint.X, basePoint.Y, basePoint.Z) + (dirRow + dirColumn)*0.5
            CreateBalls(primitive, pitch, radius, column, row, supr, columnSupr, rowSupr, center,
dirColumn, dirRow)
            break

    part = None

```

AIM Custom Template (Single-Step)

The extension **StudyDemo** contains a product target wizard for AIM named **StudyDemo1**. This wizard has only one step. In AIM, wizards for simulations are called custom templates.

XML Extension Definition File

Definitions follow for codes blocks in the XML extension definition file **StudyDemo.xml**.

Extension Definition

The `<extension>` block is the top-most level in the hierarchy.

- The attributes `name` and `version` specify the extension name and version.
- The attribute `icon` specifies the image file to display as the icon for the extension. You specify the location of the `images` folder for all images used by the extension in the `<interface>` block.
- The attribute `description` specifies the text to display in the extension details accessed via the right-click **About** option in the **Extension Manager** when it is accessed from the **ACT Start Page**.

Guid Definition

The `<guid>` block specifies a unique identifier for the extension.

Script Definition

The `<script>` block specifies the IronPython script that is referenced by the extension. This extension references `main.py`.

Wizard Interface Definition

The `<interface>` block defines the user interface for the wizard.

- The attribute `context` is set to `Study` because the extension runs from ANSYS AIM. In AIM, a *study* is the workspace in which you create and define one or more simulation processes.
- The attribute `images` specifies the `images` folder for all images that the extension is to use. You use the attribute `icon` for the extension, wizard, step, and so on to specify an image file in this folder to use as an icon.

Wizard Definition

The custom template `StudyDemo1` is defined in the `<wizard>` block named `StudyDemo1`. This custom template has only a single step.

- The attributes `name` and `version` specify the wizard name and version.
- The attribute `context` specifies the ANSYS product in which the wizard is executed. Because the wizard runs from AIM, `context` is set to `Study`.
- The attribute `icon` specifies the filename for the image to display for the custom template on the AIM start page and when adding a new simulation from the **Study** panel. The file `icon.png` is stored in the folder specified for the attribute `images` in the `<interface>` block.
- The attribute `description` specifies the text to display in the **About** window for the extension when it is accessed from the **Extension Manager** and in the **About** window for the wizard when it is accessed from the **Wizards** page.

Step Definition

The `<step>` block defines the single step of the custom template: `Step1`.

- The attributes `name`, `version`, and `caption` specify the name, version, and display text for the step.
- The attribute `context` specifies the ANSYS product in which the specific step is executed. You only need to set the context for the step if it differs from the context for the wizard.
- The attribute `HelpFile` specifies the HTML file to display as the custom help for the step.

- For general information on custom help for wizards and custom templates, see [Custom Help for Wizards \(p. 159\)](#).
- For information on the custom help for this specific example, see [Defining Custom Help for the Single-Step Custom Template \(p. 383\)](#).
- The **<property>** blocks specify properties and property attributes for the step.
- The **<callbacks>** block defines callbacks to functions defined in the IronPython script. For a custom template, only the callbacks **<onupdate>**, **<onrefresh>**, and **<onreset>** are supported.

In this example:

- The callback **<onupdate>** executes the function **action**, which creates the AIM study.
- The callback **<onreset>** executes the function **reset**, which allows you to modify the template properties.
- The **<properties>** block defines a property used in a step.
 - For the property **Geometry**, the attribute **control** is set to **fileopen**, which displays a file selection property initialized to the default attribute value. The child **<help>** block indicates the text message to display as the custom help for this property: **The geometry file to import**.
 - For the property **NAnalysis**:
 - The attribute **control** is set to **integer**, which allows you to edit the integer value for the default attribute. The child **<help>** block indicates the text message to display as the custom help for this property: **The number of static structural analyses to create. You must enter an integer greater than 0**.
 - The callback **<isvalid>** invokes the function **isValid**, which validates that the entered integer is not less than 1. A custom message can be displayed when the entered value fails validation.

The XML extension definition file `StudyDemo.xml` follows.

```

<extension version="1" name="StudyDemo">
  <author>ANSYS Inc.</author>

  <guid shortid="StudyDemo">150c0680-292b-4291-b46d-bac59949cd56</guid>

  <script src="main.py" />

  <interface context="Study">
    <images>images</images>
  </interface>

  <wizard name="StudyDemo1" version="1" context="Study" icon="icon">
    <description>Wizard to demonstrate the concept inside AIM.</description>

    <step name="Step1" caption="Load geometry file" version="1" context="Study" helpFile="help/StudyDemol.html">
      <description>Import a geometry file.</description>

      <callbacks>
        <onupdate>action</onupdate>
        <onreset>reset</onreset>
      </callbacks>

      <property name="Geometry" caption="Geometry filename" control="fileopen" default="ANSYSInputs">
        <help>The geometry file to import.</help>
      <property name="NAnalysis" caption="Number of analyses" control="integer" default="1">
    
```

```
<help>The number of static structural analyses to create. You must enter an integer greater than 0.</help>
<callbacks>
  <isValid>isValid</isValid>
</callbacks>
</property>

</step>
</wizard>

</extension>
```

IronPython Script

The IronPython script main.py follows. This script defines all the functions executed by the callbacks in the XML extension definition file for the custom template.

- The function **action** is the single step in the custom template. When the **Create Simulation** button is clicked, the callback **<onupdate>** is invoked. It creates the **Study** workflow by creating and updating the **Geometry**, **Mesh**, and **Physics** tasks.
- The callback **<onreset>** invokes the function **reset**, which resets the user interface and allows the user to modify properties defined for the custom template. This occurs when encountering an error during the execution of the callback **<onupdate>**.
- The callback **<isValid>** in the **<property>** block invokes the function **isValid**, which validates that the property value entered is not less than 1. A custom message can be displayed when the entered value fails validation.

```
tasksToDelete = []
groupsToDelete = []

def action(step):
    global tasksToDelete, groupsToDelete
    tasksToDelete = []
    groupsToDelete = []
    system1 = GetSystem(Name="Study")
    importComponent1 = Study.CreateTask(
        Type="Import",
        System=system1)
    tasksToDelete.Add(importComponent1)
    study1 = system1.GetContainer(ComponentName="Study")
    import1 = importComponent1.GetTaskObject()
    geometryImportSource1 = study1.CreateEntity(
        Type="GeometryImportSource",
        Association=import1)
    geometryImportSource1.FilePath = step.Properties["Geometry"].Value
    geometryImportSource1.GenerateImportSourceOperation()

    step.UpdateProgressInformation(10.)

    pct = 10.
    for i in range(step.Properties["NAnalysis"].Value):
        meshingComponent1 = Study.CreateTask(
            Type="Meshing",
            System=system1,
            Input=importComponent1)
        tasksToDelete.Add(meshingComponent1)
        meshingComponent1.Refresh()
        physicsSolutionGroup1 = Study.CreateGroup(Name="Physics Solution")
        groupsToDelete.Add(physicsSolutionGroup1)
        physicsDefinitionComponent1 = Study.CreateTask(
            Type="Physics Definition",
            System=system1)
        tasksToDelete.Add(physicsDefinitionComponent1)
        solvePhysicsComponent1 = Study.CreateTask(
```

```

        Type="Solve Physics",
        System=system1)
tasksToDelete.Add(solvePhysicsComponent1)
physicsSolutionGroup1.Add(Component=physicsDefinitionComponent1)
physicsSolutionGroup1.Add(Component=solvePhysicsComponent1)
AddSourceToComponentInSystem(
    SourceComponent=physicsDefinitionComponent1,
    TargetComponent=solvePhysicsComponent1)
AddSourceToComponentInSystem(
    SourceComponent=meshingComponent1,
    TargetComponent=physicsDefinitionComponent1)
physicsDefinitionComponent1.Refresh()
solvePhysicsComponent1.Refresh()
physicsDefinition1 = physicsDefinitionComponent1.GetTaskObject()
physicsRegion1 = study1.CreateEntity(
    Type="PhysicsRegion",
    Association=physicsDefinition1)
solverSettings1 = study1.CreateEntity(
    Type="SolverSettings",
    Association=physicsDefinition1)
transcript1 = study1.CreateEntity(
    Type="Transcript",
    Association=physicsDefinition1)
physicsDefinitionComponent1.Refresh()
solvePhysicsComponent1.Refresh()
meshing1 = meshingComponent1.GetTaskObject()
meshing1.EngineeringIntent = "StructuralOrThermalOrElectricConduction"
physicsRegion1.Location = ["BODY1"]
physicsRegion1.PhysicsType = "Structural"
materialAssignment1 = study1.CreateEntity(
    Type="MaterialAssignment",
    Association=physicsDefinition1)
materialAssignment1.Location = ["BODY1"]
material1 = study1.CreateEntity(
    Type="Material",
    Association=physicsDefinition1)
material1.ImportEngineeringData(Name="Structural Steel")
materialAssignment1.Material = material1
pct += 10.
step.UpdateProgressInformation(pct)

if i==9:
    raise UserErrorMessageException("My own error message.")

def reset(step):
    global tasksToDelete,groupsToDelete
    system1 = GetSystem(Name="Study")
    for group in groupsToDelete:
        Study.DeleteTaskGroup(Group=group)
    for task in tasksToDelete:
        task.DeleteTask(System=system1)

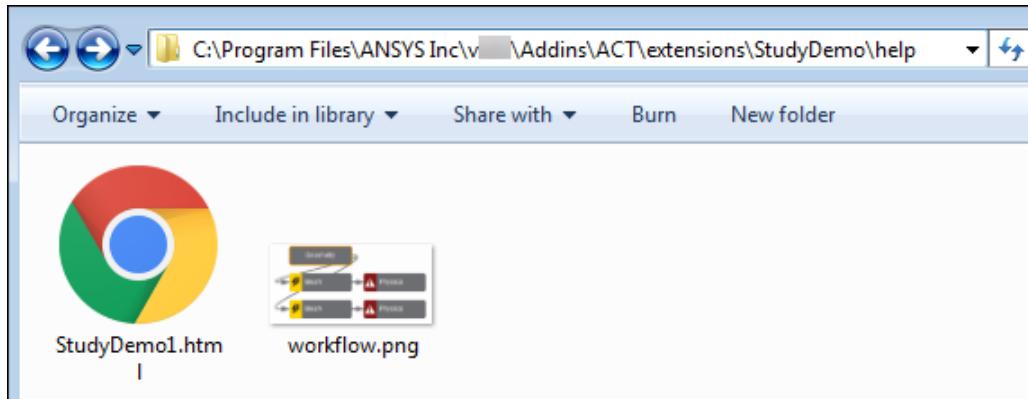
def isvalid(step, prop):
    if prop.Value<1:
        prop.StateMessage = "Must be greater than 0."
        return False
    return True

```

Defining Custom Help for the Single-Step Custom Template

One custom help file is defined for this custom template: `StudyDemo1.html`. Because this wizard is for AIM, the filename for the custom help file and the name of the step (`Step1`) in the XML file do not have to be the same. The single custom help file references the image file `workflow.png`.

In the following figure, the extension `StudyDemo` is in the default folder `extensions` for the current installation. Within this extension, the HTML and PNG files for the custom help are in a folder named `help`.



The **<step>** block in the XML extension definition file references the help file for the step:

```
<step name="Step1" caption="Load geometry file" version="1" context="Study" helpFile="help/StudyDemo1.html">
```

Additionally, for the two properties in this step, the **<properties>** blocks define the custom help to display:

```
<property name="Geometry" caption="Geometry filename" control="fileopen" default="ANSYSInputs">
  <help>The geometry file to import.</help>
</property>
<property name="nAnalysis" caption="Number of analyses" control="fileopen" default="">
  <help>The number of static structural analyses to create. You must enter an integer greater than 0.</help>
  <callbacks>
    <isValid>isValid</isValid>
  </callbacks>
</property>
```

AIM Custom Template (Multiple-Step)

The extension **PressureLossMultiple** contains a product target wizard for AIM. This multi-step wizard has the same name as the extension. In AIM, wizards for simulations are called custom templates.

XML Extension Definition File

Definitions follow for codes blocks in the XML extension definition file **PressureLossMultiple.xml**.

Extension Definition

The **<extension>** block is the top-most level in the hierarchy.

- The attributes **name** and **version** specify the extension name and version.
- The attribute **icon** specifies the image file to display as the icon for the extension. You specify the location of the **images** folder for all images used by the extension in the **<interface>** block.
- The attribute **description** specifies the text to display in the extension details accessed via the right-click **About** option in the **Extension Manager** when it is accessed from the **ACT Start Page**.

Guid Definition

The **<guid>** block specifies a unique identifier for the extension.

Script Definition

The **<script>** block specifies the IronPython script that is referenced by the extension. This extension references **pressureLoss.py**.

Wizard Interface Definition

The `<interface>` block defines the user interface for the wizard.

- The attribute `context` is set to `Project` because the extension is executed on the AIM Project tab.
- The attribute `images` specifies the `images` folder for all images that the extension is to use. You use the attribute `icon` for the extension, wizard, step, and so on to specify an image file in this folder to use as an icon.

Wizard Definition

The custom template `PressureLossMultiple` and its steps are defined in the `<wizard>` block named `PressureLossMultiple`.

- The attributes `name` and `version` specify the custom template name and version.
- The attribute `context` specifies the ANSYS product in which the wizard is executed. Because the wizard runs from AIM, `context` is set to `Study`. In AIM, a `study` is the workspace in which you create and define one or more simulation processes.
- The attribute `icon` specifies the filename for the image to display for the custom template on the start page or when adding a new simulation from the `Study` panel. This image must be stored in the `images` folder for the extension.
- The attribute `icon` specifies the filename for the image to display for the custom template. The file `loss.png` is stored in the `images` folder specified for the attribute `images` in the `<interface>` block.
- The attribute `description` specifies the text to display in the extension details accessed via the right-click `About` option in the **Extension Manager** when it is accessed from the **ACT Start Page**.

Step Definition

The `<step>` blocks define the steps of the custom template. This custom template has four steps: `Step1`, `Step2`, `Step3`, and `Step4`.

For each step:

- The attributes `name` and `version` specify the step name and version for the step.
- The attribute `context` specifies the ANSYS product in which the specific step is executed. You only need to set the context for the step if it differs from the context for the wizard.
- The attribute `HelpFile` specifies the HTML file to display as the custom help for the step.
 - For general information on custom help for wizards and custom templates, see [Custom Help for Wizards \(p. 159\)](#).
 - For information on the custom help for this specific example, see [Defining Custom Help for the Multiple-Step Custom Template \(p. 390\)](#).
- The attribute `caption` specifies the label to display for the step in AIM's data panel.
- The attribute `description` specifies the text to display for the step in the user interface for the custom template.

- The `<property>` blocks specify properties and property attributes for the step.
- The `<callbacks>` block defines callbacks to functions defined in the IronPython script. For a custom template, only the callbacks `<onupdate>`, `<onrefresh>`, and `<onreset>` are supported.

During the execution of the callback `<onupdate>` for a step, AIM displays a progress message, indicating that the step is being executed.
- The `<properties>` block defines a property used in a step. A property definition can include a callback, as in **Step3** of this example. A property definition can also include a child `<help>` block with the text message to display as the custom help for the property.

For more information about each step, see [Breakdown of Steps \(p. 386\)](#).

Breakdown of Steps

Step1

Imports the geometry and creates the workflow in AIM.

- The property `geometryfile` enables you to specify a geometry. You can either keep the geometry specified by the attribute `default` or browse to select a different one (`control="fileopen"`). For this example, a geometry file named `TubeSelectionSet.agdb` is selected.
- The callback `<onupdate>` is invoked when the **Next** button is clicked. It executes the function `importGeometry`, which imports the selected geometry and then creates the workflow in AIM. Specifically, it creates the workflow from journal content included in the function.
- The attribute `HelpFile` is set to `Step1.html`.

Step2

Refines the mesh for the geometry from low to high.

- The property `MeshResolution` enables you to set the resolution for the mesh. This property has a default of 1 (`default="1"`) and accepts only integer values (`control= "integer"`).
- The callback `<onupdate>` is invoked when the **Next** button is clicked. It executes the function `refineMesh`, which creates a mesh according to the specification.
- The attribute `HelpFile` is set to `step2.html`.

Step3

Defines the loads applied on the geometry.

- The properties `Velocity` and `GaugePressure` enable users to enter values (`control="float"`).
- The properties `InletLocation` and `OutletLocation` require users to define the inlet and outlet locations to which to apply the load (`control="scoping"`).
- When a value is entered for the outlet location, the callback `<isvalid>` is invoked. It executes the function `isValid`, which verifies that the inlet and outlet locations are not the same.
- The callback `<onupdate>` invokes the function `setup` when the **Next** button is clicked. This function applies the loads to the selected locations and performs the calculations.
- The attribute `HelpFile` is set to `step3.html`.

Step4

Exposes the maximum velocity and pressure loss results.

- The properties **MaximumVelocity** and **PressureLoss** show the calculated values in read-only format (**readonly="true"**).
- The attribute **HelpFile** is set to `step3.html`.
- When the **Finish** button is clicked, the custom template closes, returning the user to the analysis.

The file `PressureLossMultiple.xml` follows.

```
<extension version="1" name="PressureLossMultiple" icon="images\loss.png">
<guid shortid="AimMultiStepWizard">C3F86496-2F13-49E3-B7D0-353542207EAE</guid>
<author>ANSYS Inc.</author>
<description>Demonstration of a pressure loss in AIM.</description>

<script src="pressureLoss.py" />

<interface context="Project">
  <images>images</images>
</interface>

<wizard name="PressureLossMultiple" version="1" context="Study" icon="loss">
  <description>This wizard is for demonstration of ACT wizard capability in AIM.</description>

  <step name="Step1" caption="Import the geometry" version="1" context="Study" HelpFile="help/Step1.html">
    <description>Import the geometry file and create the workflow.</description>

    <callbacks>
      <onupdate>importGeometry</onupdate>
    </callbacks>

    <property name="geometryfile" caption="geometry file" control="fileopen"
    default="E:\Geometry\TubeSelectionSet.agdb"/>
  </step>

  <step name="Step2" caption="Refine the mesh" version="1" HelpFile="help/Step2.html">
    <description>Refine the mesh from Low to High.</description>

    <callbacks>
      <onupdate>refineMesh</onupdate>
    </callbacks>

    <property name="MeshResolution" caption="mesh resolution" control="integer" default="1"/>
  </step>

  <step name="Step3" caption="Define the loads" version="1" HelpFile="help/Step3.html">
    <description>Specify the loads to applied on the geometry.</description>

    <callbacks>
      <onupdate>setup</onupdate>
    </callbacks>

    <property name="Velocity" caption="Velocity" control="float" unit="Velocity" default="0.1 [m s^-1]"/>
    <property name="InletLocation" caption="InletLocation" control="scoping" />

    <property name="GaugePressure" caption="Gauge Pressure" control="float" unit="Pressure" default="0
    [Pa]"/>
    <property name="OutletLocation" caption="OutletLocation" control="scoping" >
      <callbacks>
        <isValid>isValid</isValid>
      </callbacks>
    </property>
  </step>

  <step name="Step4" caption="Export the maximum velocity" version="1" HelpFile="help/Step4.html">
    <description>Here we are just exposing the value of the maximum velocity and the pressure
  
```

```

    loss.</description>

        <property name="MaximumVelocity" caption="Maximum Velocity" control="float" unit="Velocity"
readonly = "True"/>
        <property name="PressureLoss" caption="Pressure Loss" control="float" unit="Pressure" readonly = "True"/>
    </step>
</wizard>

</extension>

```

IronPython Script

The IronPython script `pressureLoss.py` follows. This script defines all functions executed by the callbacks in the XML extension definition file for the custom template.

```

meshingComponent1 = None
study1 = None
import1 = None
physicsDefinitionComponent1 = None
resultsEvaluationComponent1 = None
solvePhysicsComponent1 = None
physicsRegion1 = None
vectorResult1 = None
singleValueResult1 = None
results1 = None
materialAssignment1 = None
currentStep = None
clr.AddReference("Ans.UI")

def getSelectionSetsForProject():
    context = __scriptingEngine__.CommandContext
    project = context.Project
    containers = project.GetContainers()
    dataEntity = "SelectionSet"

    for container in containers:
        if container.Name == "Study":
            try:
                lockObject = context.ContainerReadLock(container)
                dataReferences = project.GetDataReferencesByType(container, dataEntity)
            finally:
                lockObject.Dispose()
            break
    return dataReferences

def isValid(step, property):
    if property.Value == step.Properties["InletLocation"].Value:
        ExtAPI.Log.WriteWarning("Inlet and Outlet locations must be different.")
        return False
    return True

def importGeometry(step):
    global meshingComponent1, import1, study1, results1, vectorResult1, singleValueResult1,
    physicsDefinitionComponent1, resultsEvaluationComponent1, solvePhysicsComponent1, physicsRegion1,
    materialAssignment1
    with Transaction():
        system1 = GetSystem(Name="Study")
        physicsDefinitionComponent1 = Study.CreateTask( Type="Physics Definition", System=system1)
        study1 = system1.GetContainer(ComponentName="Study")
        physicsDefinition1 = physicsDefinitionComponent1.GetTaskObject()
        physicsRegion1 = study1.CreateEntity( Type="PhysicsRegion", Association=physicsDefinition1)
        solverSettings1 = study1.CreateEntity( Type="SolverSettings", Association=physicsDefinition1)
        solvePhysicsComponent1 = Study.CreateTask( Type="Solve Physics", System=system1,
Input=physicsDefinitionComponent1)
        solvePhysicsComponent1.Refresh()
        resultsEvaluationComponent1 = Study.CreateTask( Type="Results Evaluation", System=system1,
Input=solvePhysicsComponent1)
        resultsEvaluationComponent1.Refresh()
        physicsDefinition1.CalculationType = "Static"
        physicsRegion1.PhysicsType = "Fluid"
        physicsRegion1.Location = "AllBodies()"

```

```

materialAssignment1 = study1.CreateEntity( Type="MaterialAssignment", Association=physicsDefinition1)
material1 = study1.CreateEntity( Type="Material", Association=physicsDefinition1)
material1.ImportEngineeringData(Name="Air")
materialAssignment1.Material = material1

materialAssignment1.Location = [physicsRegion1]
results1 = resultsEvaluationComponent1.GetTaskObject()
vectorResult1 = study1.CreateEntity( Type="VectorResult", Association=results1)
vectorResult1.Variable = "Velocity"
vectorResult1.DisplayText = "Velocity"
transcript1 = study1.CreateEntity( Type="Transcript", Association=physicsDefinition1)
transcript1.DisplayText = "Fluid Flow Output 1"
physicsSolutionGroup1 = Study.CreateGroup(Name="Physics Solution")
physicsSolutionGroup1.Add(Component=physicsDefinitionComponent1)
physicsSolutionGroup1.Add(Component=solvePhysicsComponent1)
meshingComponent1 = Study.CreateTask( Type="Meshing", System=system1, Output=physicsDefinitionComponent1)
physicsDefinitionComponent1.Refresh()
solvePhysicsComponent1.Refresh()
resultsEvaluationComponent1.Refresh()
importComponent1 = Study.CreateTask( Type="Import", System=system1, Output=meshingComponent1)
meshingComponent1.Refresh()
physicsDefinitionComponent1.Refresh()
solvePhysicsComponent1.Refresh()
resultsEvaluationComponent1.Refresh()
import1 = importComponent1.GetTaskObject()
geometryImportSource1 = import1.AddGeometryImportSourceOperation()
geometryImportSource1.FilePath = step.Properties["geometryfile"].Value
importComponent1.Update(AllDependencies=True)
meshingComponent1.Refresh()
physicsDefinitionComponent1.Refresh()
solvePhysicsComponent1.Refresh()
resultsEvaluationComponent1.Refresh()

Study.ChangeModelSelectionContextTo(import1)

def refineMesh(step):
    global meshingComponent1, import1, study1, results1, vectorResult1, physicsDefinitionComponent1,
singleValueResult1,
resultsEvaluationComponent1, solvePhysicsComponent1, physicsRegion1
    meshing1 = meshingComponent1.GetTaskObject()
    meshing1.MeshResolution = step.Properties["MeshResolution"].Value

    Study.ChangeModelSelectionContextTo(import1)

def setup(step):
    global meshingComponent1, study1, results1, vectorResult1, physicsDefinitionComponent1, singleValueResult1,
resultsEvaluationComponent1, solvePhysicsComponent1, physicsRegion1
    with Transaction():
        meshing1 = meshingComponent1.GetTaskObject()
        meshControlLocalInflation1 = study1.CreateEntity( Type="MeshControlLocalInflation", Association=meshing1)
        meshing1.EngineeringIntent = "FluidFlow"
        AddSourceToComponentInSystem( SourceComponent=physicsDefinitionComponent1,
TargetComponent=resultsEvaluationComponent1)
        resultsEvaluationComponent1.Refresh()

    Study.Delete(Items=[meshControlLocalInflation1])

    with Transaction():
        meshingComponent1.Update(AllDependencies=True)
        physicsDefinitionComponent1.Refresh()
        solvePhysicsComponent1.Refresh()
        resultsEvaluationComponent1.Refresh()

    with Transaction():
        inletBoundary1 = study1.CreateEntity( Type="InletBoundary", Association=physicsRegion1)
        inlet_location = step.Properties["InletLocation"].Value.LocationSet

        if inlet_location == None :
            ExtAPI.Log.WriteMessage("inlet selection set does not exist")

        inletBoundary1.Location = inlet_location
        inletBoundary1.Flow.Velocity.Magnitude = step.Properties["Velocity"].DisplayString

```

```
outletBoundary1 = study1.CreateEntity( Type="OutletBoundary", Association=physicsRegion1)
outlet_location = step.Properties["OutletLocation"].Value.LocationSet

if outlet_location == None :
    ExtAPI.Log.WriteMessage("outlets selection set does not exist")

outletBoundary1.Location = outlet_location
outletBoundary1.Flow.Pressure.GaugeStaticPressure = step.Properties["GaugePressure"].DisplayString

wallBoundary1 = study1.CreateEntity( Type="WallBoundary", Association=physicsRegion1)

# Creation of the pressure loss expression.
singleValueResult1 = study1.CreateEntity( Type="SingleValueResult", Association=results1)
singleValueResult1.Method = "UserDefinedExpressionMethod"
singleValueResult1.Expression = "Average(Pressure, GetBoundary('@Inlet 1'), Weight='Area') - Average(Pressure, GetBoundary('@Outlet 1'), Weight='Area')"

with Transaction():
    physicsDefinitionComponent1.Update(AllDependencies=True)
    solvePhysicsComponent1.Update(AllDependencies=True)
    resultsEvaluationComponent1.Refresh()
    resultsEvaluationComponent1.Refresh()
resultsEvaluationComponent1.Update(AllDependencies=True)

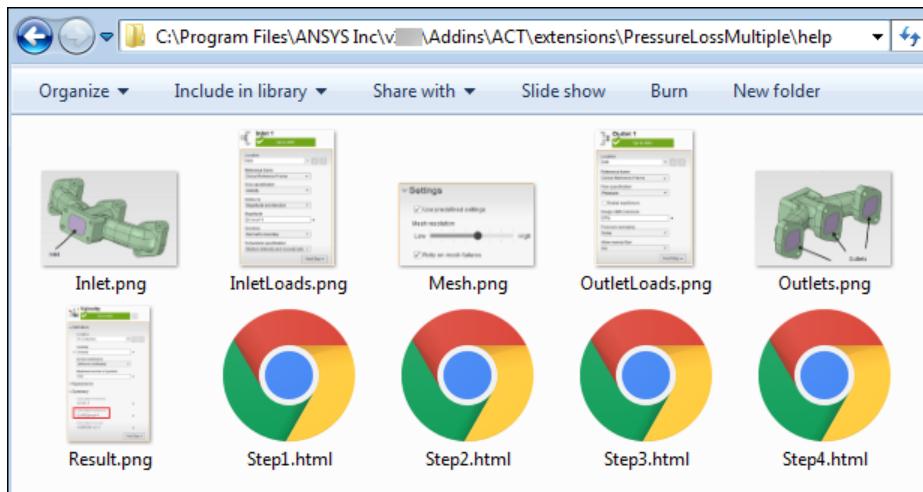
vectorResult1.Legend.Coloring = "Banded"
vectorResult1.Legend.NumberOfColors = "10"
vectorResult1.Distribution = "Mesh"
vectorResult1.Evaluate()
maximum_velocity = vectorResult1.Summary.Max
step.NextStep.Properties["MaximumVelocity"].Value = maximum_velocity.ToString()
step.NextStep.Properties["PressureLoss"].Value = singleValueResult1.Value.ToString()

def solve(step):
    global meshingComponent1, study1, results1, vectorResult1, physicsDefinitionComponent1,
    singleValueResult1, resultsEvaluationComponent1, solvePhysicsComponent1, physicsRegion1
    with Transaction():
        physicsDefinitionComponent1.Update(AllDependencies=True)
        solvePhysicsComponent1.Update(AllDependencies=True)
        resultsEvaluationComponent1.Refresh()
        resultsEvaluationComponent1.Refresh()
    resultsEvaluationComponent1.Update(AllDependencies=True)

    vectorResult1.Legend.Coloring = "Banded"
    vectorResult1.Legend.NumberOfColors = "10"
    vectorResult1.Distribution = "Mesh"
    vectorResult1.Evaluate()
    maximum_velocity = vectorResult1.Summary.Max
    step.NextStep.Properties["MaximumVelocity"].Value = maximum_velocity.ToString()
    step.NextStep.Properties["PressureLoss"].Value = singleValueResult1.Value.ToString()
```

Defining Custom Help for the Multiple-Step Custom Template

For the custom template PressureLossMultiple, help files are defined for each of its four steps: Step1.html, Step2.html, Step3.html, and Step4.html. For an AIM wizard, filenames for custom help files do not have to be the same as the names for the steps. However, in this example, they are the same. The HTML files and the image files that they reference are placed in a folder within the extension. In this example, the folder is named help.



In the XML file, each `<step>` block references the custom help file for this step. For example, the `Step1` definition begins as follows:

```
<step name="Step1" caption="Import the geometry" version="1" context="Study" HelpFile="help/Step1.html">
```

If you wanted to define custom help for a property, you would do so directly in the `<properties>` block in the XML file.

Wizard Custom Layout Example

You can use the optional `<uidefinition>` block to create custom layouts for your wizard interface. To demonstrate, a project wizard named `CustomLayoutWizard` is supplied as an example. It is defined in the extension `CustomLayout`.

This extension defines two custom layouts—one for each step in the `CustomLayoutWizard` wizard.

Note

Because the basics of constructing a wizard extension are described elsewhere, this section focuses solely on aspects of layout customization.

XML Extension Definition File

Definitions follow for codes blocks in the XML extension definition file `CustomLayout.xml`.

In the `<uidefinition>` block, two layouts are defined:

- **TabularDataLayout:**
 - This layout has five components: `Title`, `Steps`, `TabularData`, `Help`, and `Submit`.
 - This layout is referenced by the step `TabularData` `Sample` with `TabularDataLayout@CustomLayout`.
- **ChartLayout:**
 - This layout has four components: `Title`, `Steps`, `Chart`, and `Submit`.

- This layout is referenced by the step **Chart Sample** with **ChartLayout@CustomLayout**.

The file **CustomLayout.xml** follows.

```
<extension name="CustomLayout" version="0">
  <guid shortid="CustomWizardLayout">e7fa4157-90b7-4156-ac31-4000d1d687e1</guid>
  <script src="main.py" />
  <wizard name="CustomLayoutWizard" caption="CustomLayoutWizard" version="1" context="Project">
    <step layout="TabularDataLayout@CustomLayout" name="TabularData Sample" version="0"
caption="TabularData Sample">
      <propertytable display="Worksheet" control="tabulardata" name="Table" caption="Table"
persistent="False" parameterizable="False">
        <property control="quantity" name="Time" caption="Time" persistent="False"
parameterizable="False" unit="Time" />
        <property control="quantity" name="Pressure" caption="Pressure" persistent="False"
parameterizable="False" unit="Pressure" />
      </propertytable>
      <callbacks>
        <onrefresh>onrefreshTabularDataSample</onrefresh>
        <onreset>onresetTabularDataSample</onreset>
      </callbacks>
    </step>
    <step layout="ChartLayout@CustomLayout" name="Chart Sample" version="0" caption="Chart Sample">
      <callbacks>
        <onrefresh>onrefreshChartSample</onrefresh>
      </callbacks>
    </step>
  </wizard>
  <uidefinition>
    <layout name="TabularDataLayout">
      <component name="Title" leftoffset="10" topoffset="10" rightoffset="10" bottomattachment="Steps"
bottomoffset="10" heighttype="FitToContent" height="0" widthtype="Percentage" width="100"
componenttype="startPageHeaderComponent" />
      <component name="Steps" leftoffset="10" topattachment="Title" topoffset="0"
rightattachment="TabularData" rightoffset="0" bottomattachment="Help" bottomoffset="10"
heighttype="Percentage" height="67" widthtype="Percentage" width="30" componenttype="stepsListComponent" />
      <component name="TabularData" leftattachment="Steps" leftoffset="0" topattachment="Title"
topoffset="0" rightoffset="10" bottomattachment="Help" bottomoffset="10" heighttype="Percentage"
height="67" widthtype="Percentage" width="70" componenttype="tabularDataComponent" />
      <component name="Help" leftoffset="10" topattachment="TabularData" topoffset="0" rightoffset="10"
bottomattachment="Submit" bottomoffset="10" heighttype="Percentage" height="33" widthtype="Percentage"
width="100" componenttype="helpContentComponent" />
      <component name="Submit" leftoffset="10" topattachment="Help" topoffset="0" rightoffset="10"
bottomoffset="10" heighttype="FitToContent" height="0" widthtype="Percentage" width="100"
componenttype="buttonsComponent" />
    </layout>
    <layout name="ChartLayout">
      <component name="Title" leftoffset="10" topoffset="10" rightoffset="10" bottomattachment="Steps"
bottomoffset="10" heighttype="FitToContent" height="0" widthtype="Percentage" width="100"
componenttype="startPageHeaderComponent" />
      <component name="Steps" leftoffset="10" topattachment="Title" topoffset="0"
rightattachment="Chart" rightoffset="0" bottomattachment="Submit" bottomoffset="10"
heighttype="Percentage" height="100" widthtype="Percentage" width="30" componenttype="stepsListComponent" />
      <component name="Chart" leftattachment="Steps" leftoffset="0" topattachment="Title" topoffset="0"
rightoffset="10" bottomattachment="Submit" bottomoffset="10" heighttype="Percentage" height="100"
widthtype="Percentage" width="70" componenttype="chartComponent" />
      <component name="Submit" leftoffset="10" topattachment="Chart" topoffset="0" rightoffset="10"
bottomoffset="10" heighttype="FitToContent" height="0" widthtype="Percentage" width="100"
componenttype="buttonsComponent" />
    </layout>
  </uidefinition>
  <description>Demo layout</description>
</extension>
```

IronPython Script

The IronPython script `main.py` follows. This script defines all functions executed by the callbacks in the XML extension definition file. Note that the functions reference the defined layouts using `GetComponent()`.

```

def onrefreshTabularDataSample(step):
    comp = step.UserInterface.GetComponent( "TabularData" )
    table = step.Properties[ "Table" ]
    comp SetPropertyTable(table)

def onrefreshChartSample(step):
    table = step.PreviousStep.Properties[ "Table" ]
    tableValue = table.Value
    rowCount = table.RowCount

    x = [ ]
    y = [ ]
    for rowIndex in range(0, rowCount):
        x.append(tableValue[ "Table/Time" ][rowIndex].Value.Value)
        y.append(tableValue[ "Table/Pressure" ][rowIndex].Value.Value)

    comp = step.UserInterface.GetComponent( "Chart" )
    comp.Plot(x, y)

def onresetTabularDataSample(step):
    #nothing to do
    pass

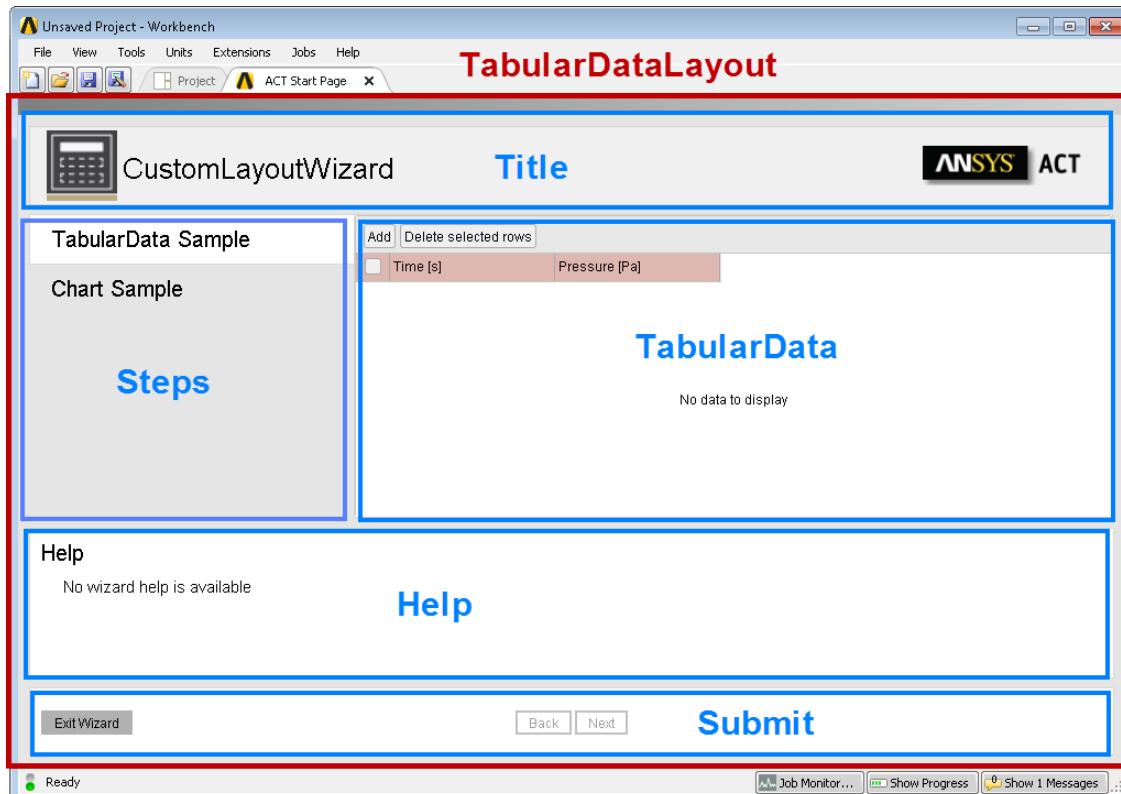
```

Exposure of the Wizard Layouts

The following images show the exposure of the layouts defined in the extension **CustomLayoutWizard**. In both images, the layout is noted in red and individual components are noted in blue.

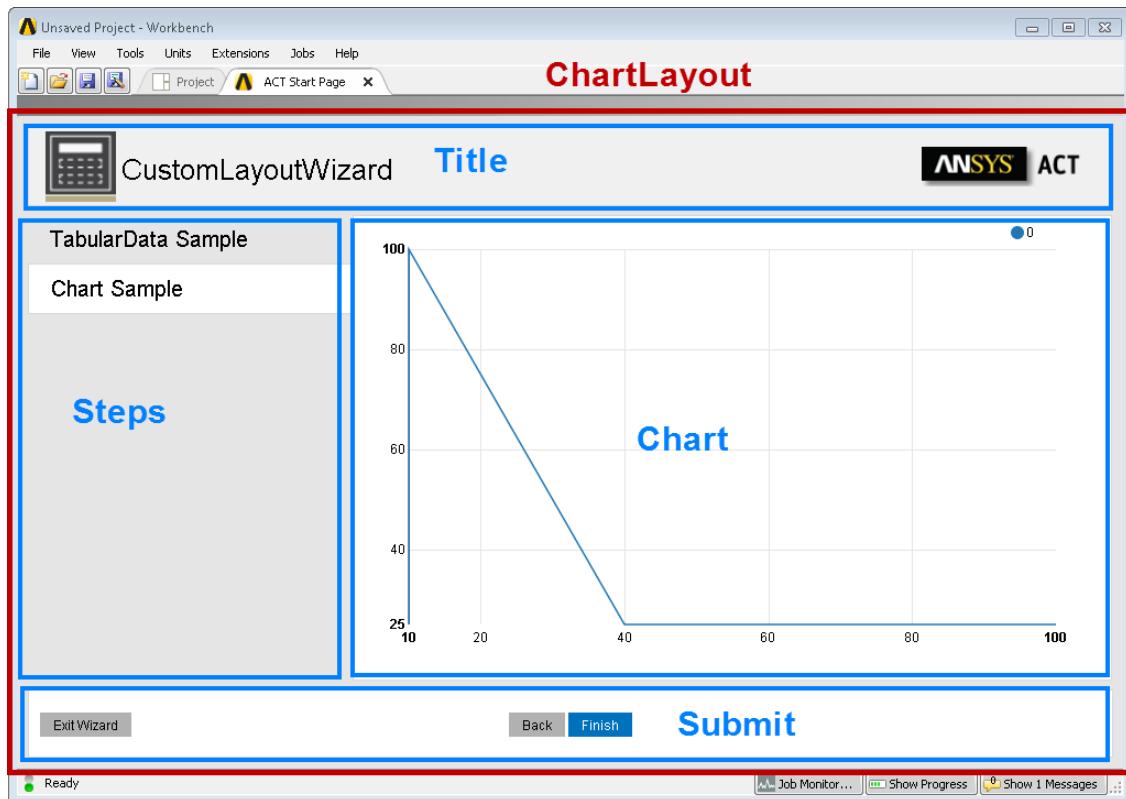
The layout **TabularDataLayout** is used for the step **TabularData Sample**.

Data must be entered into the component **TabularData** for the **Next** button in the component **Submit** to become enabled.



The layout **ChartLayout** is used for the step **Chart Sample**.

The component **Chart** contains the chart generated for the tabular data entered in the previous step.



XML Extension Definition

ACT extensions have two basic components: an XML extension definition file that describes the content of the extension and an IronPython script that defines the functions that the XML file references.

Note

The term *element* used in this section is synonymous with the term *block* used elsewhere in this document.

This section provides summary information on the following subset of the XML tags used in ACT:

- Top-level `<extension>` (p. 396) element
 - Attributes of the `<extension>` element
 - Secondary elements that can go under the `<extension>` element
 - Sub-elements, attributes, and callbacks that can go under the secondary element

For the most comprehensive and up-to-date list of available XML tags—as well as more detailed information on tags, attributes, and callbacks—see the following reference documents:

Table 8: ANSYS ACT XML Reference Documentation

Document	Description	Location
ACT Online Reference Guide	<p>Easy-to-navigate HTML document providing both API and XML reference materials.</p> <p>Both online (.html) and downloadable (.zip) versions are available.</p> <p>*Recommend version of developer's reference documentation.</p>	<p>ACT Resources page on the ANSYS Customer Portal (Downloads > ACT Resources)</p>
ACT XML Reference Guide	<p>Reference document providing XML definition and tagging information.</p>	<p>ANSYS Product Help (Help > ANSYS Documentation > Customization)</p> <p>ANSYS Customer Portal (Knowledge Resources ></p>

Document	Description	Location
		Online Documentation > Customization)

Note

The content of the XML file is case-sensitive.

<extension>

The main element for an XML extension definition file is the **<extension>** element, which provides initialization and configuration information for the extension. This is the base tag or “root” element under which all other elements fall.

```
<extension version="[version number]" minorversion="[minor version number]"
name="[extension name]">
    <author>author name</author>
    <description>description of the extension</description>
    <assembly src="[source file name (string)]"
        namespace="[namespace (string)]"/>
</extension>
```

Beneath the **<extension>** element are the secondary elements that make up the extension.

The secondary elements and the attributes for the **<extension>** element follow. Click the links to access the corresponding sections in this document.

Secondary XML elements under the <extension> element

<Application> (p. 397)

Defines a new ACT app.

<appStoreId> (p. 398)

Defines a unique identifier for the ACT app to be used in the [ANSYS App Store](#).

<assembly> (p. 398)

Defines the assembly to be loaded.

<author> (p. 399)

Defines the author of the extension.

<description> (p. 399)

Defines the description of the extension.

<Guid> (p. 399)

Defines a unique identifier for the extension.

<Interface> (p. 400)

Specifies the customizations to be done at the GUI level.

<Licenses> (p. 402)

Defines a licenses collection for the extension.

<script> (p. 402)

Specifies the IronPython script that defines the functions called by the extension.

<simdata> (p. 403)

Defines a general section that stores all user object definitions; specifies the custom features to be integrated.

<Templates> (p. 405)

Defines a collection of control templates.

<UIDefinition> (p. 405)

Defines the user interface (customized panels and layouts) for the extension.

<Wizard> (p. 405)

Defines one or more wizards within the extension.

<Workflow> (p. 406)

Defines custom workflows composed of process integration items (tasks and task groups).

Attributes for the <extension> element

version

Major version of the extension.

Mandatory attribute.

```
version="[version number (integer)]"
```

name

Name of the extension.

Mandatory attribute.

```
name="[extension name (string)]"
```

minorversion

Minor version of the extension.

Optional attribute.

```
minorversion="[minor version number (integer)]"
```

debug

Specifies if the scripted version of the extension should be opened in the debug mode.

Optional attribute.

icon

Icon for the extension.

Optional attribute.

<Application>

Defines a new application.

```
<application>
  <callbacks> ... </callbacks>
  <description>
  <description>
  <panel>
</application>
```

Sub-element tags for the <application> element

<Callbacks>

Specifies the callbacks that invoke functions from the IronPython extension script.

<description>

Description of the application.

<Panel>

Defines a panel to display.

Attributes for the <application> element

name

Name of the application.

Mandatory.

class

Class name of the controller of the application.

Optional.

context

Defines a context or combination of contexts (separated using '|') in which the application can be launched.

Optional.

MainPanel

Name of the first panel to display.

Optional.

Callbacks for the <application> element

<OnApplicationFinished>

Invoked when the application finishes.

<OnApplicationInitialized>

Invoked when the application initializes.

<OnApplicationStarted>

Invoked when the application starts.

<appStoreId>

Defines the unique identifier to use in the ANSYS App Store.

No sub-element tags, attributes, or callbacks.

```
<appstoreid>appStoreId</appstoreid>
```

<assembly>

Defines the assembly to be loaded.

```
<assembly src="[file name]" namespace="[namespace]" />
```

Attributes for the <assembly> element

context

Context or combination of contexts (separated using '|') for the import.

Mandatory.

namespace

Namespace to import.

Mandatory.

```
namespace=" [namespace] "
```

src

Name of the DLL file to import.

Mandatory.

```
src=" [file name] "
```

<author>

Defines the author of the extension.

No sub-element tags, attributes, or callbacks.

```
<author>[Name of the author or organisation (string)]</author>
```

<description>

Defines the description of the extension.

No sub-element tags, attributes, or callbacks.

```
<description>[Description (string)]</description>
```

<Guid>

Defines a unique identifier for the extension.

The guid is the unique identifier of the extension. ACT considers two extensions with the same guid as the same extension. This is very important when the extension is deployed to ensure that two different extensions are never in conflict (have the same name, for example). This identifier must be added at least before the first build of the extension and must never changed after that. When the extension is updated, a new version created, or features added, the GUID must be kept unchanged.

```
<guid shortid=" [name (string)] ">GUID</guid>
```

Attributes for the <guid> element

shortid

Short identifier for backward compatibility. Must be the same as the extension name for all extensions created before R15.

Optional.

```
shortid="[extension name (string)]"
```

<Interface>

Defines the user interface for the extension.

```
<extension version="[version id (integer)]"  
          name="[extension name (string)]"  
          <interface context="[Project | Mechanical]">  
          ...  
</interface>
```

Sub-element tags for the <Interface> element

<Callbacks>

Specifies the callbacks that invoke functions from the IronPython extension script.

<filter>

Defines a filter.

<images>

Defines the default folder where images to be used by the extension are stored.

```
<images>[folder]</images>
```

<toolbar>

Defines a toolbar.

```
<toolbar name="[toolbar internal name (string)]" caption="[toolbar display name (string)]">  
  <entry>...</entry>  
</toolbar>
```

Attributes for the <Interface> element

context

Context or combination of contexts (separated using '|') for the interface.

Mandatory.

```
context="[context name]"
```

Callbacks for the <Interface> element

<GetPostCommands>

Called to collect all "post" commands to add to the solver input file.

```
<getpostcommands>[function(analysis,stream)]</getpostcommands>
```

<GetPreCommands>

Called to collect all "pre" commands to add to the solver input file.

```
<getprecommands>[function(analysis,stream)]</getprecommands>
```

<GetSolveCommands>

Called to collect all "solve" commands to add to the solver input file.

```
<getsolvecommands timedependent="[true | false(default)]">
```

<IsAnalysisValid>

Called to check if an analysis is valid.

```
<isanalysisvalid>[function(solver)]</isanalysisvalid>
```

<OnActiveObjectChange>

Called when the active object is changed.

<OnAfterGeometryUpdate>

Called after the geometry has been updated.

<OnAfterRemove>

Called after the object has been removed.

<OnAfterSolve>

Called after an analysis has been solved.

```
<onaftersolve>[function(analysis)]</onaftersolve>
```

<OnBeforeGeometryUpdate>

Called before the geometry is starts to update.

<OnBeforeSolve>

Called before an analysis starts to solve.

```
<onbeforesolve>[function(analysis)]</onbeforesolve>
```

<OnBodySuppressStateChange>

Called when the body suppress state has been changed.

<OnDraw>

Called when the application is drawn.

<OnDraw2D>

Called when the application is drawn.

```
<ondraw>[function()]</ondraw>
```

<OnInit>

Called when the given context is initialized.

```
<oninit>[function name(application context)] </oninit>
```

<OnLoad>

Called when a project is loaded.

```
<onload>[function(currentFolder)]</onload>
```

<OnMeshCleaned>

Called when the mesh is cleaned.

<OnMeshGenerated>

Called when the mesh is generated.

<OnPostFinished>

Called when the postprocessing ends for a given analysis.

```
<onpostfinished>[function(analysis)]</onpostfinished>
```

<OnPostStarted>

Called when the postprocessing starts for a given analysis.

```
<onpoststarted>[function(analysis)]</onpoststarted>
```

<OnReady>

Called when the application is fully loaded and in a "ready" state.

<OnSave>

Called when the project is saved.

```
<onsave>[function(currentFolder)]</onsave>
```

<OnTerminate>

Called when the given context is terminated.

```
<onterminate>[function(context)]</onterminate>
```

<Resume>

Called when a project is loaded.

```
<resume>[function(binary reader)]</resume>
```

<Save>

Called when a project is saved.

```
<save>[function(binary writer)]</save>
```

<Licenses>

Defines a licenses collection for the extension.

No sub-element tags, attributes, or callbacks.

<script>

Specifies the IronPython script referenced by the extension.

```
<extension version="[version id (integer)]"
           name="[extension name (string)]"
           <script src="[python file name (string)]"></script></extension>
```

You can either insert the IronPython script directly into the XML extension definition file, or use the **src** attribute to specify the path to the script.

Additional paths can be specified by adding new **<script>** elements. For example:

```
<script src="[Path]\filename.py" />
```

If the **src** attribute is defined, then the tag content is ignored.

By default, ACT looks for IronPython scripts in the same directory as the extension. If the scripts are not located in that directory, you can specify the path the scripts in addition to the file name. For example:

```
<script src="my_path\main.py" />
```

Attributes for the <script> element

compiled

Specifies whether the script is to be compiled as a binary file.

Optional.

```
compiled="[false(default) | true]"
```

src

Specifies the IronPython script referenced by the extension.

Optional.

```
src="[python file name (string)]"
```

<simdata>

Defines a general section that stores all user object definitions.

The <simdata> element information pertains specifically to the simulation environment. Child elements are used for integrating custom simulation features into the application. These main features are nested as child elements within the <simdata> element.

```
<simdata>
  <load>
  <object>
  <optimizer>
  <solver>
  <geometry>
  <result>
  <step>
  <ExtensionObject>
  <Sampling>
</simdata>
```

Sub-element tags for the <simdata> element

<load>

Defines a simulation load or boundary.

```
<load name="[load internal name]"
      version="[version identifier of the load]"
      caption="[load display name]"
      icon="[name of an icon file]"
      issupport="[true | false]"
      isload="[true | false]"
      color="#xxxxxxxx"
      contextual="[true(default) | false]"
      class="[class name]"
      unit="[Default unit name]"
      ...
</load>
```

<object>

Defines a simulation object.

```
<object>
  <callbacks> ...
  <property>
  <propertygroup>
  <propertytable>
```

```
<target>
</object>
```

<optimizer>

Defines an optimizer.

```
<optimizer>
  <callbacks> ... </callbacks>
  <property>
  <propertygroup>
  <propertytable>
</optimizer>
```

<solver>

Specifies a third-party solver to be used in the simulation.

```
<solver>
  <callbacks> ... </callbacks>
  <property>
  <propertygroup>
  <propertytable>
</solver>
```

<geometry>

Defines a geometry feature.

```
<geometry>
  <callbacks> ... </callbacks>
  <property>
  <propertygroup>
  <propertytable>
</geometry>
```

<result>

Defines a custom result.

```
<result>
  <callbacks> ... </callbacks>
  <property>
  <propertygroup>
  <propertytable>
</result>
```

<step>

Defines a step in a wizard.

```
<step>
  <callbacks> ... </callbacks>
  <description>
  <property>
  <propertygroup>
</step>
```

<ExtensionObject>

Extends the extension object definition. (Inherited from DesignXplorer SimEntity)

```
<extensionobject>
  <callbacks> ... </callbacks>
  <property>
  <propertygroup>
  <propertytable>
  <target>
</extensionobject>
```

<Sampling>

Defines a custom sampling.

```
<sampling>
  <callbacks> ... </callbacks>
  <property>
  <propertygroup>
  <propertytable>
  <target>
</sampling>
```

Attributes for the <simdata> element**context**

Context or combination of contexts (separated using '|').

Mandatory.

```
context="[Project | targetproduct | targetproduct]">
```

<Templates>

Defines a collection of control templates.

```
<templates>
  <controltemplate name="[ template name (string) ]" version="[version id (integer)]">
    <propertygroup>
      <property> ... </property>
      <propertygroup> ... </propertygroup>
    </propertygroup>
  </controltemplate>
</templates>
```

No sub-element tags, attributes, or callbacks.

<UIDefintion>

Defines one or more layouts that can be used for wizards.

```
<uidefinition>
  <layout>
</uidefinition>
```

Sub-element tags for the <UIDefintion> element**<ControlTemplate>**

Defines a control template for the creation of groups of properties.

<Wizard>

Defines one or more wizards within the extension.

```
<wizard>
  <author>
  <description>
  <step></step>
</wizard>
```

Sub-element tags for the <Wizard> element

<Callbacks>

Specifies the callbacks that invoke functions from the IronPython extension script.

Attributes for the <Wizard> element

<context>

Context or combination of contexts (separated using '|') for the wizard.

Mandatory.

<name>

Name of the wizard.

Mandatory.

<version>

Version of the wizard.

Mandatory.

<caption>

Caption for the wizard.

Optional.

<icon>

Icon for the wizard.

Optional.

<layout>

Layout of the wizard.

Optional.

<description>

Description of the wizard.

Optional.

Callbacks for the <Wizard> element

<canstart>

Invoked to determine whether the wizard can be started.

<Workflow>

Defines custom workflows composed of process integration items (tasks and task groups). Defines the top-level workflow tag within an ACT app.

```
<workflow>
  <callbacks> ... </callbacks>
  <taskgroups>
```

```
<tasks>
</workflow>
```

Sub-element tags for the <Workflow> element

<Callbacks>

Specifies the callbacks that invoke functions from the IronPython extension script. (Inherited from SimEntity)

<TaskGroups>

Task groupings to be exposed as organized blocks within the workflow.

<Tasks>

Tasks exposed by this workflow.

Attributes for the <Workflow> element

caption

Caption for the object. (Inherited from SimEntity)

Optional.

class

Class name of the controller of the object. (Inherited from SimEntity)

Optional.

context

Context (application) to which this workflow applies.

Mandatory.

contextual

Indicates whether the object must be displayed in the contextual menu. (Inherited from SimEntity)

Optional.

icon

Icon for the object. (Inherited from SimEntity)

Optional.

name

Name of the object. (Inherited from SimEntity)

Mandatory.

version

Version of the object. (Inherited from SimEntity)

Mandatory.

Callbacks for the <Workflow> element

<onbeforedesignpointchanged>

Invoked before the design point changes.

<onafterdesignpointchanged>

Invoked after the design point changes.

<onbeforetaskreset>

Invoked before the task is reset back to its pristine, new state.

<onaftertaskreset>

Invoked after the task has been reset back to its pristine, new state.

<onbeforetaskrefresh>

Invoked before the task consumes all upstream data and prepares any local data for an ensuing update.

<onaftertaskrefresh>

Invoked after the task has consumed all upstream data and has prepared any local data for an ensuing update.

<onbeforetaskupdate>

Invoked before the task generates all broadcast output types that render the component fully solved.

<onaftertaskupdate>

Invoked after the task has generated all broadcast output types that render the component fully solved.

<onbeforetaskcreate>

Invoked before the task is created based on an underlying template.

<onaftertaskcreate>

Invoked after the task has been created based on an underlying template.

<onbeforetaskdelete>

Invoked before the task is removed from a task group.

<onaftertaskdelete>

Invoked after the task has been removed from a task group.

<onbeforetaskduplicate>

Invoked before an identical, yet independent, clone of the task is created.

<onaftertaskduplicate>

Invoked after an identical, yet independent, clone of the task has been created.

<onbeforetasksourceschanged>

Invoked before the task processes a change in upstream sources.

<onaftertasksourceschanged>

Invoked after the task has processed a change in upstream sources.

<onbeforetaskcanusertransfer>

Invoked before the task checks whether it can consume data from a specific upstream task.

<onaftertaskcanusertransfer>

Invoked after the task has checked whether it can consume data from a specific upstream task.

<onbeforetaskcanduplicate>

Invoked before the task checks whether it permits duplication.

<onaftertaskanduplicate>

Invoked after the task has checked whether it permits duplication.

<onbeforetaskstatus>

Invoked before the task calculates its current state.

<onaftertaskstatus>

Invoked after the task has calculated its current state.

<onbeforetaskpropertyretrieval>

Invoked before the task determines the visibility of its property-containing objects.

<onaftertaskpropertyretrieval>

Invoked after the task has determined the visibility of its property-containing objects.

Appendix A. Component Input and Output Tables

The following tables list component inputs and outputs supported by ACT.

Table 9: Autodyn

Taskgroup	Task	Input	Output
Autodyn	Setup	AUTODYN_Remap	AutodynSetup
		MechanicalSetup	
		SimulationGeneratedMesh	
	Analysis	None	None

Table 10: BladeGen

Taskgroup	Task	Input	Output
BladeGen	Blade Design	None	TurboGeometry
			VistaGeometry
BladeGen (Beta)	Blade Design	None	TurboGeometry
			VistaGeometry

Table 11: CFX

Taskgroup	Task	Input	Output
CFX (Beta)	Setup	SimulationGeneratedMesh	CFXSetup
		CFXMesh	SystemCouplingSetupData
		MechanicalSetup	
	Solution	CFXSetup	CFXSolution
		CFXSolution	
CFX	Setup	SimulationGeneratedMesh	CFXSetup
		CFXMesh	SystemCouplingSetupData
		MechanicalSetup	
	Solution	CFXSetup	CFXSolution
		CFXSolution	
	Results	CFXSolution	CFDAnalysis
		FluentSolution	
		ForteSolution	
		ICEData	
		IcePakResults	
		MechanicalSolution	
		NTIOOutput	

Taskgroup	Task	Input	Output
		PolyflowSolutionType	
		VistaTFSolution	

Table 12: Design Assessment

Taskgroup	Task	Input	Output
Design Assessment	Engineering Data	AIMEngineeringDataMaterialOutputProvider	EngineeringData
		ExternalModelOutputProvider	Material
		FEMSetup	
		MatML31	
	Geometry	AnsoftCADOObject	Geometry
		FEMSetup	
		ICEData	
		Geometry	
		MechanicalResults	
		SimulationModelGeneratedPMDB	
		TurboGeometry	
	Model	AIMGeometryMeshOutputProvider	MechanicalMesh
		AIMMechanicalPhysicsDefinitionOutputProvider	MechanicalModel
		CompositeEngineeringData	SimulationEngineeringData
		EngineeringData	SimulationGeneratedMesh
		EnhancedModelData	SimulationModelGeneratedMesh
		ExternalDataSetup	SimulationModelGeneratedPMDB
		ExternalMaterialFieldDataSetup	
		ExternalModelOutputProvider	
		ExternalTraceDataSetup	
		GeneralTransfer	
		Geometry	
		MeshingAssemblyTransferType	
		Modeler	
	Setup	SimulationEngineeringData	MechanicalSetup
		SimulationModelGeneratedMesh	
		SimulationSolutionOutputProvider	
		SolidSectionData	
		MechanicalModel	SimulationSetup
		GeneralTransfer	
	Solution	MechanicalMesh	MechanicalSetup
		MechanicalSolution	
		SimulationSolutionDataInternal	
		SimulationSetup	
			MechanicalSolution

Taskgroup	Task	Input	Output
		GeneralTransfer	SimulationSolution SimulationSolutionDataInternal SimulationSolutionOutputProvider
		Results	MechanicalResults
			SimulationResults

Table 13: Direct Optimization

Taskgroup	Task	Input	Output
Direct Optimization	Optimization	DesignPointsDataTransfer	OptimizationModel

Table 14: Electric

Taskgroup	Task	Input	Output
Electric	Engineering Data	AIMEngineeringDataMaterialOutputProvider	EngineeringData
		ExternalModelOutputProvider	Material
		FEMSetup	
		MatML31	
	Geometry	AnsoftCADObject	Geometry
		FEMSetup	
		Geometry	
		ICEData	
		MechanicalResults	
		SimulationModelGeneratedPMDB	
		TurboGeometry	
	Model	AIMGeometryMeshOutputProvider	MechanicalMesh
		AIMMechanicalPhysicsDefinitionOutputProvider	MechanicalModel
		CompositeEngineeringData	SimulationEngineeringData
		EngineeringData	SimulationGeneratedMesh
		EnhancedModelData	SimulationModelGeneratedMesh
		ExternalDataSetup	SimulationModelGeneratedPMDB
		ExternalMaterialFieldDataSetup	
		ExternalModelOutputProvider	
		ExternalTraceDataSetup	
		GeneralTransfer	
		Geometry	
		MeshingAssemblyTransferType	
		Modeler	
		SimulationEngineeringData	
		SimulationModelGeneratedMesh	

Taskgroup	Task	Input	Output
		SimulationSolutionOutputProvider	
		SolidSectionData	
	Setup	MechanicalModel	SimulationSetup
		MechanicalMesh	MechanicalSetup
		SimulationSolutionDataInternal	
		GeneralTransfer	
	Solution	SimulationSetup	MechanicalSolution
		GeneralTransfer	SimulationSolution
	Results		SimulationSolutionDataInternal
			SimulationSolutionOutputProvider
	Results	SimulationSolution	MechanicalResults
			SimulationResults

Table 15: Engineering Data

Taskgroup	Task	Input	Output
Engineering Data	Engineering Data	AIMEngineeringDataMaterialOutputProvider	EngineeringData
		ExternalModelOutputProvider	Material
		FEMSetup	
		MatML31	

Table 16: Explicit Dynamics

Taskgroup	Task	Input	Output
Explicit Dynamics	Engineering Data	AIMEngineeringDataMaterialOutputProvider	EngineeringData
		ExternalModelOutputProvider	Material
		FEMSetup	
		MatML31	
	Geometry	AnsoftCADObject	Geometry
		FEMSetup	
		Geometry	
		ICEData	
		MechanicalResults	
		SimulationModelGeneratedPMDB	
		TurboGeometry	
	Model	AIMGeometryMeshOutputProvider	MechanicalMesh
		AIMMechanicalPhysicsDefinitionOutputProvider	MechanicalModel
		CompositeEngineeringData	SimulationEngineeringData
		EngineeringData	SimulationGeneratedMesh
		EnhancedModelData	SimulationModelGeneratedMesh
		ExternalDataSetup	SimulationModelGeneratedPMDB

Taskgroup	Task	Input	Output
		ExternalMaterialFieldDataSetup ExternalModelOutputProvider ExternalTraceDataSetup GeneralTransfer Geometry MeshingAssemblyTransferType Modeler SimulationEngineeringData SimulationModelGeneratedMesh SimulationSolutionOutputProvider SolidSectionData	
	Setup	MechanicalModel EnhancedMechanicalModel GeneralTransfer MechanicalMesh SimulationSolutionDataInternal	SimulationSetup MechanicalSetup
	Solution	SimulationSetup GeneralTransfer	MechanicalSolution SimulationSolution SimulationSolutionDataInternal SimulationSolutionOutputProvider
	Results	SimulationSolution	MechanicalResults SimulationResults

Table 17: Explicit Dynamics (LS-DYNA Export) (Beta)

Taskgroup	Task	Input	Output
Explicit Dynamics (LS-DYNA Export) (Beta)	Engineering Data	AIMEngineeringDataMaterialOutputProvider ExternalModelOutputProvider FEMSetup MatML31	EngineeringData Material
	Geometry	AnsoftCADObject FEMSetup Geometry ICEData MechanicalResults SimulationModelGeneratedPMDB TurboGeometry	Geometry
	Model	AIMGeometryMeshOutputProvider AIMMechanicalPhysicsDefinitionOutputProvider CompositeEngineeringData	MechanicalMesh MechanicalModel SimulationEngineeringData

Taskgroup	Task	Input	Output
		EngineeringData	SimulationGeneratedMesh
		EnhancedModelData	SimulationModelGeneratedMesh
		ExternalDataSetup	SimulationModelGeneratedPMDB
		ExternalMaterialFieldDataSetup	
		ExternalModelOutputProvider	
		ExternalTraceDataSetup	
		GeneralTransfer	
		Geometry	
		MeshingAssemblyTransferType	
		Modeler	
		SimulationEngineeringData	
		SimulationModelGeneratedMesh	
		SimulationSolutionOutputProvider	
		SolidSectionData	
	Setup	MechanicalModel	SimulationSetup
		MechanicalMesh	MechanicalSetup
		GeneralTransfer	

Table 18: External Data

Taskgroup	Task	Input	Output
External Data	Setup	None	ExternalDataSetup
			ExternalMaterialFieldDataSetup
			ExternalTraceDataSetup

Table 19: External Model

Taskgroup	Task	Input	Output
External Model	Setup	None	ExternalModelOutputProvider

Table 20: External Connection

Taskgroup	Task	Input	Output
External Connection	External Connection	None	ExternalConnectionProperties

Table 21: Finite Element Modeler

Taskgroup	Task	Input	Output
Finite Element Modeler	Model	MechanicalSetup	FEMMesh
		SimulationGeneratedMesh	FEMSetup
		CompositeEngineeringData	Geometry
		FEMSetup	

Taskgroup	Task	Input	Output
		MAPDLCdb	
		SimulationEngineeringData	
		SolidSectionData	

Table 22: Fluent (with Fluent Meshing)

Taskgroup	Task	Input	Output
Fluent	Mesh	FluentMesh	FluentTGridMesh
		Geometry	
	Setup	AIMFluentMeshOutputProvider	FluentSetup
		AIMFluentPartMeshOutputProvider	
		AIMFluentPartMeshFileOutputProvider	
		AIMFluentPhysicsDefinitionOutputProvider	
		AnsoftHeatLossDataObject	SystemCouplingSetupData
		FluentCase	
		FluentImportable	
		FluentMesh	
		FluentTGridMesh	
		ICEData	
		ICESetupData	
		SimulationGeneratedMesh	
	Solution	FluentSetup	FluentSolution
		FluentSolution	

Table 23: Fluent

Taskgroup	Task	Input	Output
Fluent	Setup	AIMFluentMeshOutputProvider	FluentSetup
		AIMFluentPartMeshFileOutputProvider	
		AIMFluentPartMeshOutpuProvider	
		AIMFluentPhysicsDefinitionOutputProvider	
		AnsoftHeatLossDataObject	
		FluentCase	
		FluentImportable	
		FluentMesh	
		FluentTGridMesh	
		ICEData	
		ICESetupData	
		SimulationGeneratedMesh	
		FluentSetup	FluentSolution
	Solution	FluentSolution	

Taskgroup	Task	Input	Output
		FluentSolution	

Table 24: Fluent (with CFD-Post) (Beta)

Taskgroup	Task	Input	Output
Fluent	Setup	AIMFluentMeshOutputProvider	FluentSetup
		AIMFluentPartMeshFileOutputProvider	SystemCouplingSetupData
		AIMFluentPartMeshOutputProvider	
		AIMFluentPhysicsDefinitionOutputProvider	
		AnsoftHeatLossDataObject	
		FluentCase	
		FluentImportable	
		FluentMesh	
		FluentTGridMesh	
		ICEData	
Fluent	Solution	ICESetupData	CFDAnalysis
		SimulationGeneratedMesh	
	Results	FluentSetup	
		FluentSolution	
	Results	CFXSolution	
		FluentSolution	
		ForteSolution	
		ICEData	
		IcePakResults	
		MechanicalSolution	
		NTIOutput	
		PolyflowSolutionType	
		VistaTFSolution	

Table 25: Fluid Flow – Blow Molding (Polyflow)

Taskgroup	Task	Input	Output
Fluid Flow – Blow Molding (Polyflow)	Geometry	AnsoftCADObject	Geometry
		FEMSetup	
		Geometry	
		ICEData	
		MechanicalResults	
		SimulationModelGeneratedPMDB	
		TurboGeometry	
	Mesh	Geometry	
			GeneratedMeshOutputForAQWAModelProvider
			MechanicalModel

Taskgroup	Task	Input	Output
		MeshingGeneratedMeshOutputProvider	MeshingGeneratedMeshOutputProvider MeshingMesh SimulationGeneratedMesh
		PolyflowTransferMesh	PolyflowSetup
		SimulationGeneratedMesh	
	Solution	PolyflowSetup	PolyflowSolutionType
		PolyflowSolution	ExternalDataSetup PolyflowSolution
		CFXSolution	CFDAnalysis
	Results	ForteSolution	
		FluentSolution	
		ICEData	
		IcePakResults	
		MechanicalSolution	
		NTIOutput	
		PolyflowSolutionType	
		VistaTFSolution	

Table 26: Fluid Flow – Extrusion (Polyflow)

Taskgroup	Task	Input	Output
Fluid Flow – Extrusion (Polyflow)	Geometry	AnsoftCADObject	Geometry
		FEMSetup	
		Geometry	
		ICEData	
		MechanicalResults	
		SimulationModelGeneratedPMDB	
		TurboGeometry	
	Mesh	Geometry	GeneratedMeshOutputForAQWAModelProvider MechanicalModel
		MeshingGeneratedMeshOutputProvider	MeshingGeneratedMeshOutputProvider MeshingMesh SimulationGeneratedMesh
	Setup	PolyflowTransferMesh	PolyflowSetup
		SimulationGeneratedMesh	
	Solution	PolyflowSetup	PolyflowSolutionType
		PolyflowSolution	ExternalDataSetup PolyflowSolution
	Results	CFXSolution	CFDAnalysis
		FluentSolution	

Taskgroup	Task	Input	Output
		ForteSolution ICEData IcePakResults MechanicalSolution NTIOoutput PolyflowSolutionType VistaTFSolution	

Table 27: Fluid Flow (CFX)

Taskgroup	Task	Input	Output
Fluid Flow (CFX)	Geometry	AnsoftCADObject	Geometry
		FEMSetup	
		ICEData	
		Geometry	
		MechanicalResults	
		SimulationModelGeneratedPMDB	
		TurboGeometry	
	Mesh	Geometry	GeneratedMeshOutputForAQWAModelProvider
			MechanicalModel
		MeshingGeneratedMeshOutputProvider	MeshingGeneratedMeshOutputProvider
			MeshtingMesh
			SimulationGeneratedMesh
	Setup	SimulationGeneratedMesh	CFXSetup
		CFXMesh	SystemCouplingSetupData
		MechanicalSetup	
	Solution	CFXSolution	CFXSolution
		CFXSetup	
	Results	CFXSolution	CFDAnalysis
		FluentSolution	
		ForteSolution	
		ICEData	
		IcePakResults	
		MechanicalSolution	
		NTIOoutput	
		PolyflowSolutionType	

Taskgroup	Task	Input	Output
		VistaTFSolution	

Table 28: Fluid Flow (Fluent)

Taskgroup	Task	Input	Output
Fluid Flow (Fluent)	Geometry	AnsoftCADObject	Geometry
		FEMSetup	
		Geometry	
		ICEData	
		MechanicalResults	
		SimulationModelGeneratedPMDB	
		TurboGeometry	
	Mesh	Geometry	GeneratedMeshOutputForAQWAModelProvider
			MechanicalModel
		MeshingGeneratedMeshOutputProvider	MeshingGeneratedMeshOutputProvider
			MechanicalModel
			MeshingMesh
	Setup		SimulationGeneratedMesh
		AIMFluentMeshOutputProvider	FluentSetup
		AIMFluentPartMeshFileOutputProvider	
		AIMFluentPartMeshOutputProvider	
		AIMFluentPhysicsDefinitionOutputProvider	
		AnsoftHeatLossDataObject	
		FluentCase	
		FluentImportable	
		FluentMesh	
		FluentTGridMesh	
	Solution	ICEData	SystemCouplingSetupData
		ICESetupData	
		SimulationGeneratedMesh	
		FluentSetup	
		FluentSolution	
		CFXSolution	
		FluentSolution	
	Results	ForteSolution	CFDAnalysis
		ICEData	
		IcePakResults	
		MechanicalSolution	
		NTIOutput	

Taskgroup	Task	Input	Output
		PolyflowSolutionType	
		VistaTFSolution	

Table 29: Fluid Flow (Polyflow)

Taskgroup	Task	Input	Output
Fluid Flow (Polyflow)	Geometry	AnsoftCADObject	Geometry
		FEMSetup	
		Geometry	
		ICEData	
		MechanicalResults	
		SimulationModelGeneratedPMDB	
	Mesh	TurboGeometry	
		Geometry	GeneratedMeshOutputForAQWAModelProvider
			MechanicalModel
		MeshtingGeneratedMeshOutputProvider	MeshtingGeneratedMeshOutputProvider
	Setup		MeshingMesh
			SimulationGeneratedMesh
	Solution	PolyflowTransferMesh	PolyflowSetup
		SimulationGeneratedMesh	
		PolyflowSetup	PolyflowSolutionType
	Results	PolyflowSolution	ExternalDataSetup
			PolyflowSolution
		CFXSolution	CFDAnalysis
		FluentSolution	
		ForteSolution	
		ICEData	
		IcePakResults	
		MechanicalSolution	
		NTIOutput	

Table 30: Geometry

Taskgroup	Task	Input	Output
Geometry	Geometry	AnsoftCADObject	Geometry
		FEMSetup	
		Geometry	
		ICEData	
		MechanicalResults	

Taskgroup	Task	Input	Output
		SimulationModelGeneratedPMDB	
		TurboGeometry	

Table 31: Harmonic Acoustics

Taskgroup	Task	Input	Output
Harmonic Acoustics	Engineering Data	AIMEngineeringDataMaterialOutputProvider	EngineeringData
		ExternalModelOutputProvider	Material
		FEMSetup	
		MatML31	
	Geometry	AnsoftCADOObject	Geometry
		FEMSetup	
		Geometry	
		ICEData	
		MechanicalResults	
		SimulationModelGeneratedPMDB	
		TurboGeometry	
	Model	AIMGeometryMeshOutputProvider	MechanicalMesh
		AIMMechanicalPhysicsDefinitionOutputProvider	MechanicalModel
		CompositeEngineeringData	SimulationEngineeringData
		EngineeringData	SimulationGeneratedMesh
		EnhancedModelData	SimulationModelGeneratedMesh
		ExternalDataSetup	SimulationModelGeneratedPMDB
		ExternalMaterialFieldDataSetup	
		ExternalModelOutputProvider	
		ExternalTraceDataSetup	
		GeneralTransfer	
		Geometry	
		MeshingAssemblyTransferType	
		Modeler	
		SimulationEngineeringData	
		SimulationModelGeneratedMesh	
		SimulationSolutionOutputProvider	
		SolidSectionData	
	Setup	ExternalDataSetup	MechanicalSetup
		MechanicalMesh	SimulationSetup
		MechanicalModel	
		MechanicalSolution	
	Solution	SimulationSetup	MechanicalSolution
			SimulationSolution

Taskgroup	Task	Input	Output
			SimulationSolutionDataInternal
			SimulationSolutionOutputProvider
	Results	SimulationSolution	MechanicalResults
			SimulationResults

Table 32: Harmonic Response

Taskgroup	Task	Input	Output
Harmonic Response	Engineering Data	AIMEngineeringDataMaterialOutputProvider ExternalModelOutputProvider FEMSetup MatML31	EngineeringData Material
	Geometry	AnsoftCADObject FEMSetup Geometry ICEData MechanicalResults SimulationModelGeneratedPMDB TurboGeometry	Geometry
	Model	AIMGeometryMeshOutputProvider AIMMechanicalPhysicsDefinitionOutputProvider CompositeEngineeringData EngineeringData EnhancedModelData ExternalDataSetup ExternalMaterialFieldDataSetup ExternalModelOutputProvider ExternalTraceDataSetup GeneralTransfer Geometry MeshingAssemblyTransferType Modeler SimulationEngineeringData SimulationModelGeneratedMesh SimulationSolutionOutputProvider SolidSectionData	MechanicalMesh MechanicalModel SimulationEngineeringData SimulationGeneratedMesh SimulationModelGeneratedMesh SimulationModelGeneratedPMDB
	Setup	MechanicalModel AnsoftForceAndMomentDataObject EnhancedMechanicalModel ExternalDataSetup	SimulationSetup MechanicalSetup

Taskgroup	Task	Input	Output
Hydrodynamic Diffraction		GeneralTransfer	
		MechanicalMesh	
		MechanicalSolution	
		SimulationSolutionDataInternal	
	Solution	SimulationSetup	MechanicalSolution
		GeneralTransfer	SimulationSolution
			SimulationSolutionDataInternal
	Results		SimulationSolutionOutputProvider
		SimulationSolution	MechanicalResults
			SimulationResults

Table 33: Hydrodynamic Diffraction

Taskgroup	Task	Input	Output
Hydrodynamic Diffraction	Geometry	AnsoftCADObject	Geometry
		FEMSetup	
		Geometry	
		ICEData	
		MechanicalResults	
		SimulationModelGeneratedPMDB	
		TurboGeometry	
	Model	GeneratedMeshOutputForAQWAModelProvider	AqwaModel
		Geometry	
	Setup	AqwaModel	AqwaSetup
	Solution	AqwaSetup	AqwaSolution ExternalDataSetupForAqwa
	Results	AqwaSolution	AqwaResults

Table 34: Hydrodynamic Response

Taskgroup	Task	Input	Output
Hydrodynamic Response	Geometry	AnsoftCADObject	Geometry
		FEMSetup	
		Geometry	
		ICEData	
		MechanicalResults	
		SimulationModelGeneratedPMDB	
		TurboGeometry	
	Model	Geometry	AqwaModel
		GeneratedMeshOutputForAQWAModelProvider	
	Setup	AqwaModel	AqwaSetup

Taskgroup	Task	Input	Output
		AqwaSolution	
	Solution	AqwaSetup	AqwaSolution ExternalDataSetupForAqwa
	Results	AqwaSolution	AqwaResults

Table 35: IC Engine (Fluent)

Taskgroup	Task	Input	Output
IC Engine (Fluent)	ICE	None	ICEData
	Geometry	AnsoftCADObject	Geometry
		FEMSetup	
		Geometry	
		ICEData	
		MechanicalResults	
		SimulationModelGeneratedPMDB	
	TurboGeometry		
	Mesh	Geometry	GeneratedMeshOutputForAQWAModelProvider MechanicalModel
		MesherGeneratedMeshOutputProvider	MesherGeneratedMeshOutputProvider MesherMesh
			SimulationGeneratedMesh
	ICE Solver Setup	ForteSMGData	ICESetupData
		SimulationGeneralMesh	
		SimulationGeneratedMesh	
	Setup	AIMFluentMeshOutputProvider	FluentSetup
		AIMFluentPartMeshFileOutputProvider	
		AIMFluentPartMeshOutputProvider	
		AIMFluentPhysicsDefinitionOutputProvider	
		FluentImportable	
		AnsoftHeatLossDataObject	SystemCouplingSetupData
		FluentCase	
		FluentMesh	
		FluentTGridMesh	
		ICEData	
	Solution	ICESetupData	
		SimulationGeneratedMesh	
	Results	FluentSetup	FluentSolution
		FluentSolution	
	Results	CFXSolution	CFDAnalysis
		FluentSolution	

Taskgroup	Task	Input	Output
		ForteSolution ICEData IcePakResults MechanicalSolution NTIOutput PolyflowSolutionType VistaTFSolution	

Table 36: IC Engine (Forte)

Taskgroup	Task	Input	Output
IC Engine (Forte)	ICE	None	ICEData
	Geometry	AnsoftCADObject	Geometry
		FEMSetup	
		Geometry	
		ICEData	
		MechanicalResults	
		SimulationModelGeneratedPMDB	
	Mesh	TurboGeometry	
		Geometry	GeneratedMeshOutputForAQWAModelProvider
			MechanicalModel
		MesherGeneratedMeshOutputProvider	MesherGeneratedMeshOutputProvider
	ICE Setup		MeshingMesh
			SimulationGeneratedMesh
		ForteSMGData	ICESetupData
	Results	SimulationGeneralMesh	
		SimulationGeneratedMesh	
		Forte	ICESetupData
		ForteSolution	CFDAnalysis
		CFXSolution	
		FluentSolution	
		ForteSolution	
		ICEData	

Taskgroup	Task	Input	Output
		VistaTFSolution	

Table 37: ICEM CFD

Taskgroup	Task	Input	Output
ICEM CFD	Model	FluentImportable	MeshingAssemblyTransferType
		Geometry	SimulationGeneratedMesh
		MechanicalMesh	
		MeshingMesh	
		Modeler	

Table 38: Icepak

Taskgroup	Task	Input	Output
Icepak	Setup	AnsoftHeatLossDataObject	IcePakSetup
		Geometry	
	Solution	IcePakSetup	IcePakResults

Table 39: Eigenvalue Buckling

Taskgroup	Task	Input	Output
Eigenvalue Buckling	Engineering Data	AIMEngineeringDataMaterialOutputProvider	EngineeringData
		ExternalModelOutputProvider	Material
		FEMSetup	
		MatML31	
	Geometry	AnsoftCADObject	Geometry
		FEMSetup	
		Geometry	
		ICEData	
		MechanicalResults	
		SimulationModelGeneratedPMDB	
		TurboGeometry	
	Model	AIMGeometryMeshOutputProvider	MechanicalMesh
		AIMMechanicalPhysicsDefinitionOutputProvider	MechanicalModel
		CompositeEngineeringData	SimulationEngineeringData
		EngineeringData	SimulationGeneratedMesh
		EnhancedModelData	SimulationModelGeneratedMesh
		ExternalDataSetup	SimulationModelGeneratedPMDB
		ExternalMaterialFieldDataSetup	
		ExternalModelOutputProvider	
		ExternalTraceDataSetup	
		GeneralTransfer	

Taskgroup	Task	Input	Output
		Geometry MeshingAssemblyTransferType Modeler SimulationEngineeringData SimulationModelGeneratedMesh SimulationSolutionOutputProvider SolidSectionData	
	Setup	MechanicalModel SimulationSolutionDataInternal EnhancedMechanicalModel GeneralTransfer MechanicalMesh	SimulationSetup MechanicalSetup
	Solution	SimulationSetup GeneralTransfer	MechanicalSolution SimulationSolution SimulationSolutionDataInternal SimulationSolutionOutputProvider
	Results	SimulationSolution	ExternalModelOutputProvider MechanicalResults SimulationResults

Table 40: EigenValue Buckling (Samcef)

Taskgroup	Task	Input	Output
Eigenvalue Buckling (Samcef)	Engineering Data	AIMEngineeringDataMaterialOutputProvider ExternalModelOutputProvider FEMSetup MatML31	EngineeringData Material
	Geometry	AnsoftCADObject FEMSetup Geometry ICEData MechanicalResults SimulationModelGeneratedPMDB TurboGeometry	Geometry
	Model	AIMGeometryMeshOutputProvider AIMMechanicalPhysicsDefinitionOutputProvider CompositeEngineeringData EngineeringData EnhancedModelData ExternalDataSetup	MechanicalMesh MechanicalModel SimulationEngineeringData SimulationGeneratedMesh SimulationModelGeneratedMesh SimulationModelGeneratedPMDB

Taskgroup	Task	Input	Output
		ExternalMaterialFieldDataSetup ExternalModelOutputProvider ExternalTraceDataSetup GeneralTransfer Geometry MeshingAssemblyTransferType Modeler SimulationEngineeringData SimulationModelGeneratedMesh SimulationSolutionOutputProvider SolidSectionData	
	Setup	MechanicalModel GeneralTransfer MechanicalMesh SimulationSolutionDataInternal	SimulationSetup MechanicalSetup
	Solution	SimulationSetup GeneralTransfer	MechanicalSolution SimulationSolution SimulationSolutionDataInternal SimulationSolutionOutputProvider
	Results	SimulationSolution	MechanicalResults SimulationResults

Table 41: Magnetostatic

Taskgroup	Task	Input	Output
Magnetostatic	Engineering Data	AIMEngineeringDataMaterialOutputProvider ExternalModelOutputProvider FEMSetup MatML31	EngineeringData Material
	Geometry	AnsoftCADObject FEMSetup Geometry ICEData MechanicalResults SimulationModelGeneratedPMDB TurboGeometry	Geometry
	Model	AIMGeometryMeshOutputProvider AIMMechanicalPhysicsDefinitionOutputProvider CompositeEngineeringData EngineeringData	MechanicalMesh MechanicalModel SimulationEngineeringData SimulationGeneratedMesh

Taskgroup	Task	Input	Output
		EnhancedModelData	SimulationModelGeneratedMesh
		ExternalDataSetup	SimulationModelGeneratedPMDB
		ExternalMaterialFieldDataSetup	
		ExternalModelOutputProvider	
		ExternalTraceDataSetup	
		GeneralTransfer	
		Geometry	
		MeshingAssemblyTransferType	
		Modeler	
		SimulationEngineeringData	
		SimulationModelGeneratedMesh	
		SimulationSolutionOutputProvider	
		SolidSectionData	
	Setup	MechanicalModel	SimulationSetup
		GeneralTransfer	MechanicalSetup
		MechanicalMesh	
	Solution	SimulationSetup	MechanicalSolution
		GeneralTransfer	SimulationSolution
			SimulationSolutionDataInternal
			SimulationSolutionOutputProvider
	Results	SimulationSolution	MechanicalResults
			SimulationResults

Table 42: Feedback Iterator

Taskgroup	Task	Input	Output
Feedback Iterator	Feedback Iterator	FeedbackIteratorSetup	FeedbackIteratorEntity

Table 43: Mechanical APDL

Taskgroup	Task	Input	Output
Mechanical APDL	Analysis	FEMSetup	None
		Geometry	
		MAPDLSolution	
		MAPDLDatabase	
		MAPDLResults	
		MAPDLCdb	
		MechanicalSetup	
		MechanicalSolution	
		SimulationGeneratedMesh	

Taskgroup	Task	Input	Output
		SolidSectionData	

Table 44: Mechanical Model

Taskgroup	Task	Input	Output
Mechanical Model	Engineering Data	AIMEngineeringDataMaterialOutputProvider	EngineeringData
		ExternalModelOutputProvider	Material
		FEMSetup	
		MatML31	
	Geometry	AnsoftCADObject	Geometry
		FEMSetup	
		Geometry	
		ICEData	
		MechanicalResults	
		SimulationModelGeneratedPMDB	
		TurboGeometry	
	Model	AIMGeometryMeshOutputProvider	MechanicalMesh
		AIMMechanicalPhysicsDefinitionOutputProvider	MechanicalModel
		CompositeEngineeringData	SimulationEngineeringData
		EngineeringData	SimulationGeneratedMesh
		EnhancedModelData	SimulationModelGeneratedMesh
		ExternalDataSetup	SimulationModelGeneratedPMDB
		ExternalMaterialFieldDataSetup	
		ExternalModelOutputProvider	
		ExternalTraceDataSetup	
		GeneralTransfer	
		Geometry	
		MeshingAssemblyTransferType	
		Modeler	
		SimulationEngineeringData	
		SimulationModelGeneratedMesh	
		SimulationSolutionOutputProvider	
		SolidSectionData	

Table 45: Mesh

Taskgroup	Task	Input	Output
Mesh	Geometry	AnsoftCADObject	Geometry
		FEMSetup	
		Geometry	
		ICEData	

Taskgroup	Task	Input	Output
		MechanicalResults	
		SimulationModelGeneratedPMDB	
		TurboGeometry	
	Mesh	Geometry	GeneratedMeshOutputForAQWAModelProvider MechanicalModel
		MesherGeneratedMeshOutputProvider	MesherGeneratedMeshOutputProvider MesherMesh
			SimulationGeneratedMesh

Table 46: Microsoft Office Excel

Taskgroup	Task	Input	Output
Microsoft Office Excel	Analysis	None	MSExcelSetup

Table 47: Modal (ABAQUS)

Taskgroup	Task	Input	Output
Modal (ABAQUS)	Engineering Data	AIMEngineeringDataMaterialOutputProvider	EngineeringData
		ExternalModelOutputProvider	Material
		FEMSetup	
		MatML31	
	Geometry	AnsoftCADObject	Geometry
		FEMSetup	
		Geometry	
		ICEData	
		MechanicalResults	
		SimulationModelGeneratedPMDB	
		TurboGeometry	
	Model	AIMGeometryMeshOutputProvider	MechanicalMesh
		AIMMechanicalPhysicsDefinitionOutputProvider	MechanicalModel
		CompositeEngineeringData	SimulationEngineeringData
		EngineeringData	SimulationGeneratedMesh
		EnhancedModelData	SimulationModelGeneratedMesh
		ExternalDataSetup	SimulationModelGeneratedPMDB
		ExternalMaterialFieldDataSetup	
		ExternalModelOutputProvider	
		ExternalTraceDataSetup	
		GeneralTransfer	
		Geometry	

Taskgroup	Task	Input	Output
		MeshingAssemblyTransferType Modeler SimulationEngineeringData SimulationModelGeneratedMesh SimulationSolutionOutputProvider SolidSectionData	
	Setup	MechanicalModel GeneralTransfer MechanicalMesh SimulationSolutionDataInternal	SimulationSetup MechanicalSetup
	Solution	SimulationSetup GeneralTransfer	MechanicalSolution SimulationSolution SimulationSolutionDataInternal SimulationSolutionOutputProvider
	Results	SimulationSolution	MechanicalResults SimulationResults

Table 48: Modal

Taskgroup	Task	Input	Output
Modal	Engineering Data	AIMEngineeringDataMaterialOutputProvider ExternalModelOutputProvider FEMSetup MatML31	EngineeringData Material
	Geometry	AnsoftCADObject FEMSetup Geometry ICEData MechanicalResults SimulationModelGeneratedPMDB TurboGeometry	Geometry
	Model	AIMGeometryMeshOutputProvider AIMMechanicalPhysicsDefinitionOutputProvider CompositeEngineeringData EngineeringData EnhancedModelData ExternalDataSetup ExternalMaterialFieldDataSetup ExternalModelOutputProvider ExternalTraceDataSetup	MechanicalMesh MechanicalModel SimulationEngineeringData SimulationGeneratedMesh SimulationModelGeneratedPMDB SimulationModelGeneratedMesh

Taskgroup	Task	Input	Output
		GeneralTransfer Geometry MeshingAssemblyTransferType Modeler SimulationEngineeringData SimulationModelGeneratedMesh SimulationSolutionOutputProvider SolidSectionData	
	Setup	MechanicalModel EnhancedMechanicalModel GeneralTransfer MechanicalMesh SimulationSolutionDataInternal	SimulationSetup MechanicalSetup
	Solution	SimulationSetup GeneralTransfer	MechanicalSolution SimulationSolution SimulationSolutionDataInternal SimulationSolutionOutputProvider
	Results	SimulationSolution	ExternalModelOutputProvider MechanicalResults SimulationResults

Table 49: Modal (NASTRAN) (Beta)

Taskgroup	Task	Input	Output
Modal (NASTRAN) (Beta)	Engineering Data	AIMEngineeringDataMaterialOutputProvider ExternalModelOutputProvider FEMSetup MatML31	EngineeringData Material
	Geometry	AnsoftCADObject FEMSetup Geometry ICEData MechanicalResults SimulationModelGeneratedPMDB TurboGeometry	Geometry
	Model	AIMGeometryMeshOutputProvider AIMMechanicalPhysicsDefinitionOutputProvider CompositeEngineeringData EngineeringData EnhancedModelData	MechanicalMesh MechanicalModel SimulationGeneratedMesh SimulationEngineeringData SimulationModelGeneratedMesh

Taskgroup	Task	Input	Output
		ExternalDataSetup	SimulationModelGeneratedPMDB
		ExternalMaterialFieldDataSetup	
		ExternalModelOutputProvider	
		ExternalTraceDataSetup	
		GeneralTransfer	
		Geometry	
		MeshingAssemblyTransferType	
		Modeler	
		SimulationEngineeringData	
		SimulationModelGeneratedMesh	
		SimulationSolutionOutputProvider	
		SolidSectionData	
	Setup	GeneralTransfer	SimulationSetup
		MechanicalMesh	MechanicalSetup
		MechanicalModel	
	Solution	SimulationSetup	MechanicalSolution
		GeneralTransfer	SimulationSolution
			SimulationSolutionDataInternal
			SimulationSolutionOutputProvider
	Results	SimulationSolution	MechanicalResults
			SimulationResults

Table 50: Modal (Samcef)

Taskgroup	Task	Input	Output
Modal (Samcef)	Engineering Data	AIMEngineeringDataMaterialOutputProvider	EngineeringData
		ExternalModelOutputProvider	Material
		FEMSetup	
		MatML31	
	Geometry	AnsoftCADObject	Geometry
		FEMSetup	
		Geometry	
		ICEData	
		MechanicalResults	
		SimulationModelGeneratedPMDB	
		TurboGeometry	
	Model	AIMGeometryMeshOutputProvider	MechanicalMesh
		AIMMechanicalPhysicsDefinitionOutputProvider	MechanicalModel
		CompositeEngineeringData	SimulationEngineeringData
		EngineeringData	SimulationGeneratedMesh

Taskgroup	Task	Input	Output
		EnhancedModelData	SimulationModelGeneratedMesh
		ExternalDataSetup	SimulationModelGeneratedPMDB
		ExternalMaterialFieldDataSetup	
		ExternalModelOutputProvider	
		ExternalTraceDataSetup	
		GeneralTransfer	
		Geometry	
		MeshingAssemblyTransferType	
		Modeler	
		SimulationEngineeringData	
		SimulationModelGeneratedMesh	
		SimulationSolutionOutputProvider	
		SolidSectionData	
	Setup	MechanicalModel	MechanicalSetup
		GeneralTransfer	SimulationSetup
		MechanicalMesh	
		SimulationSolutionDataInternal	
	Solution	SimulationSetup	MechanicalSolution
		GeneralTransfer	SimulationSolution
	Results	SimulationSolution	SimulationSolutionDataInternal
		SimulationSolution	SimulationSolutionOutputProvider
	Results	SimulationSolution	MechanicalResults
		SimulationSolution	SimulationResults

Table 51: Modal Acoustics

Taskgroup	Task	Input	Output
Modal Acoustics (Beta)	Engineering Data	AIMEngineeringDataMaterialOutputProvider	EngineeringData
		ExternalModelOutputProvider	Material
		FEMSetup	
		MatML31	
	Geometry	AnsoftCADObject	Geometry
		FEMSetup	
		Geometry	
		ICEData	
		MechanicalResults	
		SimulationModelGeneratedPMDB	
		TurboGeometry	
	Model	AIMGeometryMeshOutputProvider	MechanicalMesh
		AIMMechanicalPhysicsDefinitionOutputProvider	MechanicalModel

Taskgroup	Task	Input	Output
		CompositeEngineeringData	SimulationEngineeringData
		EngineeringData	SimulationGeneratedMesh
		EnhancedModelData	SimulationModelGeneratedMesh
		ExternalDataSetup	SimulationModelGeneratedPMDB
		ExternalMaterialFieldDataSetup	
		ExternalModelOutputProvider	
		ExternalTraceDataSetup	
		GeneralTransfer	
		Geometry	
		MeshingAssemblyTransferType	
		Modeler	
		SimulationEngineeringData	
		SimulationModelGeneratedMesh	
		SimulationSolutionOutputProvider	
		SolidSectionData	
Setup	MechanicalMesh	MechanicalSetup	
	MechanicalModel	SimulationSetup	
Solution	SimulationSetup	MechanicalSolution	
		SimulationSolution	
		SimulationSolutionDataInternal	
		SimulationSolutionOutputProvider	
Results	SimulationSolution	MechanicalResults	
		SimulationResults	

Table 52: Parameters Correlation

Taskgroup	Task	Input	Output
Parameters Correlation	Parameters Correlation	ResponseSurfaceDataTransfer	CorrelationModel
			DesignPointsDataTransfer

Table 53: Polyflow – Blow Molding

Taskgroup	Task	Input	Output
Polyflow – Blow Molding	Setup	PolyflowTransferMesh	PolyflowSetup
		SimulationGeneratedMesh	
	Solution	PolyflowSetup	PolyflowSolutionType
		PolyflowSolution	

Taskgroup	Task	Input	Output
			PolyflowSolution

Table 54: Polyflow – Extrusion

Taskgroup	Task	Input	Output
Polyflow - Extrusion	Setup	PolyflowTransferMesh	PolyflowSetup
		SimulationGeneratedMesh	
	Solution	PolyflowSetup	PolyflowSolutionType
		PolyflowSolution	ExternalDataSetup PolyflowSolution

Table 55: Polyflow

Taskgroup	Task	Input	Output
Polyflow	Setup	PolyflowTransferMesh	PolyflowSetup
		SimulationGeneratedMesh	
	Solution	PolyflowSetup	PolyflowSolutionType
		PolyflowSolution	ExternalDataSetup PolyflowSolution

Table 56: Random Vibration

Taskgroup	Task	Input	Output
Random Vibration	Engineering Data	AIMEngineeringDataMaterialOutputProvider	EngineeringData
		ExternalModelOutputProvider	Material
		FEMSetup	
		MatML31	
	Geometry	AnsoftCADObject	Geometry
		FEMSetup	
		Geometry	
		ICEData	
		MechanicalResults	
		SimulationModelGeneratedPMDB	
		TurboGeometry	
	Model	AIMGeometryMeshOutputProvider	MechanicalMesh
		AIMMechanicalPhysicsDefinitionOutputProvider	MechanicalModel
		CompositeEngineeringData	SimulationEngineeringData
		EngineeringData	SimulationGeneratedMesh
		EnhancedModelData	SimulationModelGeneratedMesh
		ExternalDataSetup	SimulationModelGeneratedPMDB
		ExternalMaterialFieldDataSetup	
		ExternalModelOutputProvider	

Taskgroup	Task	Input	Output
		ExternalTraceDataSetUp	
		GeneralTransfer	
		Geometry	
		MesherAssemblyTransferType	
		Modeler	
		SimulationEngineeringData	
		SimulationModelGeneratedMesh	
		SimulationSolutionOutputProvider	
		SolidSectionData	
	Setup	MechanicalModel	SimulationSetup
		EnhancedMechanicalModel	MechanicalSetup
		GeneralTransfer	
		MechanicalMesh	
		SimulationSolutionDataInternal	
	Solution	SimulationSetup	MechanicalSolution
		GeneralTransfer	SimulationSolution
			SimulationSolutionDataInternal
			SimulationSolutionOutputProvider
	Results	SimulationSolution	MechanicalResults
			SimulationResults

Table 57: Response Spectrum

Taskgroup	Task	Input	Output
Response Spectrum	Engineering Data	AIMEngineeringDataMaterialOutputProvider	EngineeringData
		ExternalModelOutputProvider	Material
		FEMSetup	
		MatML31	
	Geometry	AnsoftCADObject	Geometry
		FEMSetup	
		Geometry	
		ICEData	
		MechanicalResults	
		SimulationModelGeneratedPMDB	
		TurboGeometry	
	Model	AIMGeometryMeshOutputProvider	MechanicalMesh
		AIMMechanicalPhysicsDefinitionOutputProvider	MechanicalModel
		CompositeEngineeringData	SimulationEngineeringData
		EngineeringData	SimulationGeneratedMesh
		EnhancedModelData	SimulationModelGeneratedPMDB

Taskgroup	Task	Input	Output
		ExternalDataSetup	SimulationModelGeneratedMesh
		ExternalMaterialFieldDataSetup	
		ExternalModelOutputProvider	
		ExternalTraceDataSetup	
		GeneralTransfer	
		Geometry	
		MeshingAssemblyTransferType	
		Modeler	
		SimulationEngineeringData	
		SimulationModelGeneratedMesh	
		SimulationSolutionOutputProvider	
		SolidSectionData	
	Setup	MechanicalModel	SimulationSetup
		EnhancedMechanicalModel	MechanicalSetup
		GeneralTransfer	
		MechanicalMesh	
		SimulationSolutionDataInternal	
	Solution	SimulationSetup	MechanicalSolution
		GeneralTransfer	SimulationSolution
			SimulationSolutionDataInternal
			SimulationSolutionOutputProvider
	Results	SimulationSolution	MechanicalResults
			SimulationResults

Table 58: Response Surface

Taskgroup	Task	Input	Output
Response Surface	Design of Experiments	None	DesignPointsDataTransfer
			DOEModel
			ParametricContext
	Response Surface	ParametricContext	ResponseSurfaceModel
		DOEModel	DesignPointsDataTransfer
			ResponseSurfaceDataTransfer

Table 59: Response Surface Optimization

Taskgroup	Task	Input	Output
Response Surface Optimization	Design of Experiments	None	ParametricContext
			DOEModel
			DesignPointsDataTransfer

Taskgroup	Task	Input	Output
	Response Surface	ParametricContext	ResponseSurfaceModel
		DOEModel	DesignPointsDataTransfer ResponseSurfaceDataTransfer
	Optimization	ParametricContext ResponseSurfaceModel	OptimizationModel

Table 60: Results

Taskgroup	Task	Input	Output
Results	Results	CFXSolution	CFDAnalysis
		FluentSolution	
		ForteSolution	
		ICEData	
		IcePakResults	
		MechanicalSolution	
		NTIOutput	
		PolyflowSolutionType	
		VistaTFSolution	

Table 61: Rigid Dynamics

Taskgroup	Task	Input	Output
Rigid Dynamics	Engineering Data	AIMEngineeringDataMaterialOutputProvider	EngineeringData
		ExternalModelOutputProvider	Material
		FEMSetup	
		MatML31	
	Geometry	AnsoftCADObject	Geometry
		FEMSetup	
		Geometry	
		ICEData	
		MechanicalResults	
		SimulationModelGeneratedPMDB	
		TurboGeometry	
	Model	AIMGeometryMeshOutputProvider	MechanicalMesh
		AIMMechanicalPhysicsDefinitionOutputProvider	MechanicalModel
		CompositeEngineeringData	
		EngineeringData	SimulationGeneratedMesh
		EnhancedModelData	SimulationModelGeneratedMesh
		ExternalDataSetup	SimulationModelGeneratedPMDB
		ExternalMaterialFieldDataSetup	
		ExternalModelOutputProvider	

Taskgroup	Task	Input	Output
		ExternalTraceDataSetup GeneralTransfer Geometry MeshingAssemblyTransferType Modeler SimulationEngineeringData SimulationModelGeneratedMesh SimulationSolutionOutputProvider SolidSectionData	
	Setup	MechanicalModel GeneralTransfer MechanicalMesh	SimulationSetup MechanicalSetup
	Solution	SimulationSetup GeneralTransfer	MechanicalSolution SimulationSolution SimulationSolutionDataInternal SimulationSolutionOutputProvider
	Results	SimulationSolution	MechanicalResults SimulationResults

Table 62: Shape Optimization (Beta)

Taskgroup	Task	Input	Output
Shape Optimization (Beta)	Engineering Data	AIMEngineeringDataMaterialOutputProvider ExternalModelOutputProvider FEMSetup MatML31	EngineeringData Material
	Geometry	AnsoftCADObject FEMSetup Geometry ICEData MechanicalResults SimulationModelGeneratedPMDB TurboGeometry	Geometry
	Model	AIMGeometryMeshOutputProvider AIMMechanicalPhysicsDefinitionOutputProvider CompositeEngineeringData EngineeringData EnhancedModelData ExternalDataSetup ExternalMaterialFieldDataSetup	MechanicalMesh MechanicalModel SimulationEngineeringData SimulationGeneratedMesh SimulationModelGeneratedMesh SimulationModelGeneratedPMDB

Taskgroup	Task	Input	Output
		ExternalModelOutputProvider ExternalTraceDataSetup GeneralTransfer Geometry MeshingAssemblyTransferType Modeler SimulationEngineeringData SimulationModelGeneratedMesh SimulationSolutionOutputProvider SolidSectionData	
	Setup	MechanicalModel CFXSolution FluentSolution GeneralTransfer IcePakResults MechanicalMesh MechanicalSolution SimulationSolutionDataInternal	SimulationSetup MechanicalSetup
	Solution	SimulationSetup GeneralTransfer	MechanicalSolution SimulationSolution SimulationSolutionDataInternal SimulationSolutionOutputProvider
	Results	SimulationSolution	MechanicalResults SimulationResults

Table 63: Six Sigma Analysis

Taskgroup	Task	Input	Output
Six Sigma Analysis	Design of Experiments (SSA)		DesignPointsDataTransfer
			DOEModel
			ParametricContext
	Response Surface (SSA)	DOEModel	ResponseSurfaceModel
		ParametricContext	DesignPointsDataTransfer ResponseSurfaceDataTransfer
	Six Sigma Analysis	ParametricContext	SixSigmaModel

Taskgroup	Task	Input	Output
		ResponseSurfaceModel	

Table 64: Static Structural (ABAQUS)

Taskgroup	Task	Input	Output
Static Structural (ABAQUS)	Engineering Data	AIMEngineeringDataMaterialOutputProvider	EngineeringData
		ExternalModelOutputProvider	Material
		FEMSetup	
		MatML31	
	Geometry	AnsoftCADObject	Geometry
		FEMSetup	
		Geometry	
		ICEData	
		MechanicalResults	
		SimulationModelGeneratedPMDB	
		TurboGeometry	
	Model	AIMGeometryMeshOutputProvider	MechanicalMesh
		AIMMechanicalPhysicsDefinitionOutputProvider	MechanicalModel
		CompositeEngineeringData	SimulationEngineeringData
		EngineeringData	SimulationGeneratedMesh
		EnhancedModelData	SimulationModelGeneratedMesh
		ExternalDataSetup	SimulationModelGeneratedPMDB
		ExternalMaterialFieldDataSetup	
		ExternalModelOutputProvider	
		ExternalTraceDataSetup	
		Geometry	
		GeneralTransfer	
		MeshingAssemblyTransferType	
		Modeler	
		SimulationEngineeringData	
		SimulationModelGeneratedMesh	
		SimulationSolutionOutputProvider	
		SolidSectionData	
	Setup	MechanicalModel	SimulationSetup
		CFXSolution	MechanicalSetup
		ExternalDataSetup	
		FluentSolution	
		GeneralTransfer	
		IcePakResults	
		MechanicalMesh	

Taskgroup	Task	Input	Output
		MechanicalSolution	
		SimulationSolutionDataInternal	
	Solution	SimulationSetup	MechanicalSolution
		GeneralTransfer	SimulationSolution SimulationSolutionDataInternal SimulationSolutionOutputProvider
	Results	SimulationSolution	MechanicalResults
			SimulationResults

Table 65: Static Structural

Taskgroup	Task	Input	Output
Static Structural	Engineering Data	AIMEngineeringDataMaterialOutputProvider	EngineeringData
		ExternalModelOutputProvider	Material
		FEMSetup	
		MatML31	
	Geometry	AnsoftCADObject	Geometry
		FEMSetup	
		Geometry	
		ICEData	
		MechanicalResults	
		SimulationModelGeneratedPMDB	
		TurboGeometry	
	Model	AIMGeometryMeshOutputProvider	MechanicalMesh
		AIMMechanicalPhysicsDefinitionOutputProvider	MechanicalModel
		CompositeEngineeringData	SimulationEngineeringData
		EngineeringData	SimulationGeneratedMesh
		EnhancedModelData	SimulationModelGeneratedMesh
		ExternalDataSetup	SimulationModelGeneratedPMDB
		ExternalMaterialFieldDataSetup	
		ExternalModelOutputProvider	
		ExternalTraceDataSetup	
		GeneralTransfer	
		Geometry	
		MeshingAssemblyTransferType	
		Modeler	

Taskgroup	Task	Input	Output
Setup	MechanicalModel	MechanicalModel	SimulationSetup
	MechanicalMesh	MechanicalMesh	MechanicalSetup
	AnsoftForceDataObject	AnsoftForceDataObject	SystemCouplingSetupData
	CFXSolution	CFXSolution	
	EnhancedMechanicalModel	EnhancedMechanicalModel	
	ExternalDataSetup	ExternalDataSetup	
	ExternalDataSetupForAqwa	ExternalDataSetupForAqwa	
	FluentSolution	FluentSolution	
	GeneralTransfer	GeneralTransfer	
	IcePakResults	IcePakResults	
Solution	MechanicalSolution	MechanicalSolution	MechanicalSolution
	SimulationSolutionDataInternal	SimulationSolutionDataInternal	SimulationSolution
	SimulationSetup	SimulationSetup	SimulationSolutionDataInternal
	GeneralTransfer	GeneralTransfer	SimulationSolutionOutputProvider
Results	SimulationSolution	SimulationSolution	ExternalModelOutputProvider
			MechanicalResults
			SimulationResults

Table 66: Static Structural (Samcef)

Taskgroup	Task	Input	Output
Static Structural (Samcef)	Engineering Data	AIMEngineeringDataMaterialOutputProvider	EngineeringData
		ExternalModelOutputProvider	Material
		FEMSetup	
		MatML31	
	Geometry	AnsoftCADObject	Geometry
		FEMSetup	
		Geometry	
		ICEData	
		MechanicalResults	
		SimulationModelGeneratedPMDB	
		TurboGeometry	
	Model	AIMGeometryMeshOutputProvider	MechanicalMesh
		AIMMechanicalPhysicsDefinitionOutputProvider	MechanicalModel
		CompositeEngineeringData	SimulationEngineeringData
		EngineeringData	SimulationGeneratedMesh
		EnhancedModelData	SimulationModelGeneratedMesh
		ExternalDataSetup	SimulationModelGeneratedPMDB

Taskgroup	Task	Input	Output
		ExternalMaterialFieldDataSetup ExternalModelOutputProvider ExternalTraceDataSetup GeneralTransfer Geometry MeshingAssemblyTransferType Modeler SimulationEngineeringData SimulationModelGeneratedMesh SimulationSolutionOutputProvider SolidSectionData	
	Setup	MechanicalModel CFXSolution ExternalDataSetup FluentSolution GeneralTransfer IcePakResults MechanicalMesh MechanicalSolution SimulationSolutionDataInternal	SimulationSetup MechanicalSetup
	Solution	SimulationSetup GeneralTransfer	MechanicalSolution SimulationSolution SimulationSolutionDataInternal SimulationSolutionOutputProvider
	Results	SimulationSolution	MechanicalResults SimulationResults

Table 67: Steady-State Thermal (ABAQUS)

Taskgroup	Task	Input	Output
Steady-State Thermal (ABAQUS)	Engineering Data	AIMEngineeringDataMaterialOutputProvider ExternalModelOutputProvider FEMSetup MatML31	EngineeringData Material
	Geometry	AnsoftCADObject FEMSetup Geometry ICEData MechanicalResults SimulationModelGeneratedPMDB	Geometry

Taskgroup	Task	Input	Output
Model		TurboGeometry	
	AIM	GeometryMeshOutputProvider	MechanicalMesh
	AIM	MechanicalPhysicsDefinitionOutputProvider	MechanicalModel
	Composite	EngineeringData	SimulationEngineeringData
	Engineering	Data	SimulationGeneratedMesh
	Enhanced	ModelData	SimulationModelGeneratedMesh
	External	DataSetup	SimulationModelGeneratedPMDB
	External	MaterialFieldDataSetup	
	External	ModelOutputProvider	
	External	TraceDataSetup	
	General	Transfer	
	Geometry		
	Meshing	AssemblyTransferType	
	Modeler		
	Simulation	EngineeringData	
	Simulation	ModelGeneratedMesh	
	Simulation	SolutionOutputProvider	
	SolidSectionData		
Setup	Mechanical	Model	SimulationSetup
	CFX	Solution	MechanicalSetup
	External	DataSetup	
	Fluent	Solution	
	General	Transfer	
	IcePak	Results	
	Mechanical	Mesh	
	Mechanical	Solution	
	Simulation	SolutionDataInternal	
Solution	Simulation	Setup	MechanicalSolution
	General	Transfer	SimulationSolution
Results	Simulation	Solution	MechanicalResults
			SimulationResults

Table 68: Steady-State Thermal

Taskgroup	Task	Input	Output
Steady-State Thermal	Engineering Data	AIMEngineeringDataMaterialOutputProvider	EngineeringData
		ExternalModelOutputProvider	Material
		FEMSetup	

Taskgroup	Task	Input	Output
Geometry		MatML31	Geometry
		AnsoftCADObject	
		FEMSetup	
		Geometry	
		ICEData	
		MechanicalResults	
		SimulationModelGeneratedPMDB	
		TurboGeometry	
Model		AIMGeometryMeshOutputProvider	MechanicalMesh
		AIMMechanicalPhysicsDefinitionOutputProvider	MechanicalModel
		CompositeEngineeringData	SimulationEngineeringData
		EngineeringData	SimulationGeneratedMesh
		EnhancedModelData	SimulationModelGeneratedMesh
		ExternalDataSetup	SimulationModelGeneratedPMDB
		ExternalMaterialFieldDataSetup	
		ExternalModelOutputProvider	
		ExternalTraceDataSetup	
		GeneralTransfer	
		Geometry	
		MeshingAssemblyTransferType	
		Modeler	
		SimulationEngineeringData	
		SimulationModelGeneratedMesh	
		SimulationSolutionOutputProvider	
		SolidSectionData	
Setup		MechanicalModel	SimulationSetup
		MechanicalMesh	MechanicalSetup
		AnsoftHeatLossDataObject	SystemCouplingSetupData
		CFXSolution	
		ExternalDataSetup	
		FluentSolution	
		GeneralTransfer	
		IcePakResults	
		MechanicalSolution	
		SimulationSolutionDataInternal	
Solution		SimulationSetup	MechanicalSolution
		GeneralTransfer	SimulationSolution
			SimulationSolutionDataInternal

Taskgroup	Task	Input	Output
	Results	SimulationSolution	SimulationSolutionOutputProvider
			MechanicalResults
			SimulationResults

Table 69: Steady-State Thermal (Samcef)

Taskgroup	Task	Input	Output
Steady-State Thermal	Engineering Data	AIMEngineeringDataMaterialOutputProvider	EngineeringData
		ExternalModelOutputProvider	Material
		FEMSetup	
		MatML31	
	Geometry	AnsoftCADObject	Geometry
		FEMSetup	
		Geometry	
		ICEData	
		MechanicalResults	
		SimulationModelGeneratedPMDB	
		TurboGeometry	
	Model	AIMGeometryMeshOutputProvider	MechanicalMesh
		AIMMechanicalPhysicsDefinitionOutputProvider	MechanicalModel
		CompositeEngineeringData	SimulationEngineeringData
		EngineeringData	SimulationGeneratedMesh
		EnhancedModelData	SimulationModelGeneratedMesh
		ExternalDataSetup	SimulationModelGeneratedPMDB
		ExternalMaterialFieldDataSetup	
		ExternalModelOutputProvider	
		ExternalTraceDataSetup	
		Geometry	
		GeneralTransfer	
		MeshingAssemblyTransferType	
		Modeler	
	Setup	SimulationEngineeringData	MechanicalSetup
		SimulationModelGeneratedMesh	
		SimulationSolutionOutputProvider	
		SolidSectionData	
		CFXSolution	SimulationSetup
		ExternalDataSetup	
		FluentSolution	
		GeneralTransfer	
		IcePakResults	

Taskgroup	Task	Input	Output
System Coupling		MechanicalMesh	
		MechanicalModel	
		MechanicalSolution	
		SimulationSolutionDataInternal	
	Solution	SimulationSetup	MechanicalSolution
		GeneralTransfer	SimulationSolution
			SimulationSolutionDataInternal
	Results		SimulationSolutionOutputProvider
		SimulationSolution	MechanicalResults
			SimulationResults

Table 70: System Coupling

Taskgroup	Task	Input	Output
System Coupling	Setup	ExternalDataSetup	CouplingSetupProvider
		SystemCouplingSetupData	
	Solution	CouplingSetupProvider	None

Table 71: Thermal-Electric

Taskgroup	Task	Input	Output
Thermal-Electric	Engineering Data	AIMEngineeringDataMaterialOutputProvider	EngineeringData
		ExternalModelOutputProvider	Material
		FEMSetup	
		MatML31	
	Geometry	AnsoftCADObject	Geometry
		FEMSetup	
		Geometry	
		ICEData	
		MechanicalResults	
		SimulationModelGeneratedPMDB	
		TurboGeometry	
	Model	AIMGeometryMeshOutputProvider	MechanicalMesh
		AIMMechanicalPhysicsDefinitionOutputProvider	MechanicalModel
		CompositeEngineeringData	SimulationEngineeringData
		EngineeringData	SimulationGeneratedMesh
		EnhancedModelData	SimulationModelGeneratedMesh
		ExternalDataSetup	SimulationModelGeneratedPMDB
		ExternalMaterialFieldDataSetup	
		ExternalModelOutputProvider	
		ExternalTraceDataSetup	

Taskgroup	Task	Input	Output
		Geometry GeneralTransfer MeshingAssemblyTransferType Modeler SimulationEngineeringData SimulationModelGeneratedMesh SimulationSolutionOutputProvider SolidSectionData	
	Setup	MechanicalModel CFXSolution ExternalDataSetup FluentSolution IcePakResults MechanicalMesh GeneralTransfer	SimulationSetup MechanicalSetup SystemCouplingSetupData
	Solution	SimulationSetup GeneralTransfer	MechanicalSolution SimulationSolution SimulationSolutionDataInternal SimulationSolutionOutputProvider
	Results	SimulationSolution	MechanicalResults SimulationResults

Table 72: Throughflow

Taskgroup	Task	Input	Output
Throughflow	Geometry	AnsoftCADObject FEMSetup Geometry ICEData MechanicalResults SimulationModelGeneratedPMDB TurboGeometry	Geometry
	Setup	Geometry VistaGeometry VistaTFPhysics	VistaTFSetup
	Solution	VistaTFSetup VistaTFSolution	VistaTFSolution
	Results	CFXSolution FluentSolution ForteSolution	CFDAnalysis

Taskgroup	Task	Input	Output
		ICEData IcePakResults MechanicalSolution NTIOutput PolyflowSolutionType VistaTFSolution	

Table 73: Throughflow (BladeGen)

Taskgroup	Task	Input	Output
Throughflow (BladeGen)	Blade Design	None	TurboGeometry VistaGeometry
Setup		Geometry	VistaTFSetup
		VistaGeometry	
		VistaTFPhysics	
Solution	VistaTFSetup		VistaTFSolution
	VistaTFSolution		
Results		CFXSolution	CFDAnalysis
		FluentSolution	
		ForteSolution	
		ICEData	
		IcePakResults	
		MechanicalSolution	
		NTIOutput	
		PolyflowSolutionType	
		VistaTFSolution	

Table 74: Transient Structural (ABAQUS)

Taskgroup	Task	Input	Output
Transient Structural (ABAQUS)	Engineering Data	AIMEngineeringDataMaterialOutputProvider	EngineeringData
		ExternalModelOutputProvider	Material
		FEMSetup	
		MatML31	
Geometry	AnsoftCADObject		Geometry
	FEMSetup		
	Geometry		
	ICEData		
	MechanicalResults		
	SimulationModelGeneratedPMDB		
	TurboGeometry		

Taskgroup	Task	Input	Output
Transient Structural	Model	AIMGeometryMeshOutputProvider	MechanicalMesh
		AIMMechanicalPhysicsDefinitionOutputProvider	MechanicalModel
		CompositeEngineeringData	SimulationEngineeringData
		EngineeringData	SimulationGeneratedMesh
		EnhancedModelData	SimulationModelGeneratedMesh
		ExternalDataSetup	SimulationModelGeneratedPMDB
		ExternalMaterialFieldDataSetup	
		ExternalModelOutputProvider	
		ExternalTraceDataSetup	
		Geometry	
		GeneralTransfer	
		MeshingAssemblyTransferType	
		Modeler	
		SimulationEngineeringData	
		SimulationModelGeneratedMesh	
		SimulationSolutionOutputProvider	
		SolidSectionData	
	Setup	MechanicalModel	SimulationSetup
		CFXSolution	MechanicalSetup
		FluentSolution	
		ExternalDataSetup	
		GeneralTransfer	
		IcePakResults	
		MechanicalMesh	
		MechanicalSolution	
	Solution	SimulationSolutionDataInternal	
		SimulationSetup	MechanicalSolution
		GeneralTransfer	SimulationSolution
			SimulationSolutionDataInternal
	Results		SimulationSolutionOutputProvider
		SimulationSolution	MechanicalResults
			SimulationResults

Table 75: Transient Structural

Taskgroup	Task	Input	Output
Transient Structural	Engineering Data	AIMEngineeringDataMaterialOutputProvider	EngineeringData
		ExternalModelOutputProvider	Material
		FEMSetup	
		MatML31	

Taskgroup	Task	Input	Output
Geometry	AnsoftCADObject	AnsoftCADObject	Geometry
	FEMSetup	FEMSetup	
	Geometry	Geometry	
	ICEData	ICEData	
	MechanicalResults	MechanicalResults	
	SimulationModelGeneratedPMDB	SimulationModelGeneratedPMDB	
	TurboGeometry	TurboGeometry	
	AIMGeometryMeshOutputProvider	AIMGeometryMeshOutputProvider	MechanicalMesh
	AIMMechanicalPhysicsDefinitionOutputProvider	AIMMechanicalPhysicsDefinitionOutputProvider	MechanicalModel
	CompositeEngineeringData	CompositeEngineeringData	SimulationEngineeringData
	EngineeringData	EngineeringData	SimulationGeneratedMesh
	EnhancedModelData	EnhancedModelData	SimulationModelGeneratedMesh
	ExternalDataSetup	ExternalDataSetup	SimulationModelGeneratedPMDB
	ExternalMaterialFieldDataSetup	ExternalMaterialFieldDataSetup	
	ExternalModelOutputProvider	ExternalModelOutputProvider	
	ExternalTraceDataSetup	ExternalTraceDataSetup	
	GeneralTransfer	GeneralTransfer	
	Geometry	Geometry	
	MeshingAssemblyTransferType	MeshingAssemblyTransferType	
	Modeler	Modeler	
	SimulationEngineeringData	SimulationEngineeringData	
	SimulationModelGeneratedMesh	SimulationModelGeneratedMesh	
Setup	SimulationSolutionOutputProvider	SimulationSolutionOutputProvider	SystemCouplingSetupData
	SolidSectionData	SolidSectionData	
	MechanicalModel	MechanicalModel	
	MechanicalMesh	MechanicalMesh	
	AnsoftForceDataObject	AnsoftForceDataObject	
	CFXSolution	CFXSolution	
	EnhancedMechanicalModel	EnhancedMechanicalModel	
	ExternalDataSetup	ExternalDataSetup	
	FluentSolution	FluentSolution	
	GeneralTransfer	GeneralTransfer	
Solution	IcePakResults	IcePakResults	MechanicalSolution
	MechanicalSolution	MechanicalSolution	
	SimulationSolutionDataInternal	SimulationSolutionDataInternal	
	SimulationSetup	SimulationSetup	
	GeneralTransfer	GeneralTransfer	

Taskgroup	Task	Input	Output
	Results	SimulationSolution	SimulationSolutionOutputProvider
			ExternalModelOutputProvider
			MechanicalResults
			SimulationResults

Table 76: Transient Structural (Samcef)

Taskgroup	Task	Input	Output
Transient Structural (Samcef)	Engineering Data	AIMEngineeringDataMaterialOutputProvider	EngineeringData
		ExternalModalOutputProvider	Material
		FEMSetup	
		MatML31	
	Geometry	AnsoftCADObject	Geometry
		FEMSetup	
		Geometry	
		ICEData	
		MechanicalResults	
		SimulationModelGeneratedPMDB	
		TurboGeometry	
	Model	AIMGeometryMeshOutputProvider	MechanicalMesh
		AIMMechanicalPhysicsDefinitionOutputProvider	MechanicalModel
		CompositeEngineeringData	SimulationEngineeringData
		EngineeringData	SimulationGeneratedMesh
		EnhancedModelData	SimulationModelGeneratedMesh
		ExternalDataSetup	SimulationModelGeneratedPMDB
		ExternalMaterialFieldDataSetup	
		ExternalModelOutputProvider	
		ExternalTraceDataSetup	
		GeneralTransfer	
		Geometry	
		MeshingAssemblyTransferType	
		Modeler	
		SimulationEngineeringData	
		SimulationModelGeneratedMesh	
		SimulationSolutionOutputProvider	
		SolidSectionData	
	Setup	MechanicalModel	SimulationSetup
		CFXSolution	MechanicalSetup
		ExternalDataSetup	
		FluentSolution	

Taskgroup	Task	Input	Output
Transient Thermal (ABAQUS)		GeneralTransfer	
		IcePakResults	
		MechanicalMesh	
		MechanicalSolution	
		SimulationSolutionDataInternal	
	Solution	SimulationSetup	MechanicalSolution
		GeneralTransfer	SimulationSolution
			SimulationSolutionDataInternal
	Results		SimulationSolutionOutputProvider
		SimulationSolution	MechanicalResults
			SimulationResults

Table 77: Transient Thermal (ABAQUS)

Taskgroup	Task	Input	Output
Transient Thermal (ABAQUS)	Engineering Data	AIMEngineeringDataMaterialOutputProvider	EngineeringData
		ExternalModelOutputProvider	Material
		FEMSetup	
		MatML31	
	Geometry	AnsoftCADObject	Geometry
		FEMSetup	
		Geometry	
		ICEData	
		MechanicalResults	
		SimulationModelGeneratedPMDB	
		TurboGeometry	
	Model	AIMGeometryMeshOutputProvider	MechanicalMesh
		AIMMechanicalPhysicsDefinitionOutputProvider	MechanicalModel
		CompositeEngineeringData	SimulationEngineeringData
		EngineeringData	SimulationGeneratedMesh
		EnhancedModelData	SimulationModelGeneratedMesh
		ExternalDataSetup	SimulationModelGeneratedPMDB
		ExternalMaterialFieldDataSetup	
		ExternalModelOutputProvider	
		ExternalTraceDataSetup	
		GeneralTransfer	
		Geometry	
		MeshingAssemblyTransferType	
		Modeler	
		SimulationEngineeringData	

Taskgroup	Task	Input	Output
Transient Thermal	Setup	SimulationModelGeneratedMesh	
		SimulationSolutionOutputProvider	
		SolidSectionData	
	Setup	MechanicalModel	SimulationSetup
		CFXSolution	MechanicalSetup
		ExternalDataSetup	
		FluentSolution	
		GeneralTransfer	
		IcePakResults	
		MechanicalMesh	
		MechanicalSolution	
	Solution	SimulationSolutionDataInternal	
		SimulationSetup	
		GeneralTransfer	
		SimulationSolution	
	Results	SimulationSolutionDataInternal	
		SimulationSolutionOutputProvider	
	Results	SimulationSolution	MechanicalResults
			SimulationResults

Table 78: Transient Thermal

Taskgroup	Task	Input	Output
Transient Thermal	Engineering Data	AIMEngineeringDataMaterialOutputProvider	EngineeringData
		ExternalModelOutputProvider	Material
		FEMSetup	
		MatML31	
	Geometry	AnsoftCADObject	Geometry
		FEMSetup	
		Geometry	
		ICEData	
		MechanicalResults	
		SimulationModelGeneratedPMDB	
	Model	TurboGeometry	
		AIMGeometryMeshOutputProvider	MechanicalMesh
		AIMMechanicalPhysicsDefinitionOutputProvider	MechanicalModel
		CompositeEngineeringData	SimulationEngineeringData
		EngineeringData	SimulationGeneratedMesh
		EnhancedModelData	SimulationModelGeneratedMesh
		ExternalDataSetup	SimulationModelGeneratedPMDB
		ExternalMaterialFieldDataSetup	

Taskgroup	Task	Input	Output
		ExternalModelOutputProvider ExternalTraceDataSetup GeneralTransfer Geometry MeshingAssemblyTransferType Modeler SimulationEngineeringData SimulationModelGeneratedMesh SimulationSolutionOutputProvider SolidSectionData	
	Setup	MechanicalModel MechanicalMesh AnsoftHeatLossDataObject CFXSolution ExternalDataSetup FluentSolution GeneralTransfer IcePakResults MechanicalSolution SimulationSolutionDataInternal	SimulationSetup MechanicalSetup SystemCouplingSetupData
	Solution	SimulationSetup GeneralTransfer	MechanicalSolution SimulationSolution SimulationSolutionDataInternal SimulationSolutionOutputProvider
	Results	SimulationSolution	MechanicalResults SimulationResults

Table 79: Transient Thermal (Samcef)

Taskgroup	Task	Input	Output
Transient Thermal (Samcef)	Engineering Data	AIMEngineeringDataMaterialOutputProvider ExternalModelOutputProvider FEMSetup MatML31	EngineeringData Material
	Geometry	AnsoftCADObject FEMSetup Geometry ICEData MechanicalResults SimulationModelGeneratedPMDB	Geometry

Taskgroup	Task	Input	Output
Model	TurboGeometry		
	AIMGeometryMeshOutputProvider	MechanicalMesh	
	AIMMechanicalPhysicsDefinitionOutputProvider	MechanicalModel	
	CompositeEngineeringData	SimulationEngineeringData	
	EngineeringData	SimulationGeneratedMesh	
	EnhancedModelData	SimulationModelGeneratedMesh	
	ExternalDataSetup	SimulationModelGeneratedPMDB	
	ExternalMaterialFieldDataSetup		
	ExternalModelOutputProvider		
	ExternalTraceDataSetup		
	GeneralTransfer		
	Geometry		
	MeshingAssemblyTransferType		
	Modeler		
	SimulationEngineeringData		
	SimulationModelGeneratedMesh		
	SimulationSolutionOutputProvider		
	SolidSectionData		
Setup	MechanicalModel	SimulationSetup	
	CFXSolution	MechanicalSetup	
	ExternalDataSetup		
	FluentSolution		
	GeneralTransfer		
	IcePakResults		
	MechanicalMesh		
	MechanicalSolution		
	SimulationSolutionDataInternal		
Solution	SimulationSetup	MechanicalSolution	
	GeneralTransfer	SimulationSolution	
Results	SimulationSolution	SimulationSolutionOutputProvider	

Table 80: Turbomachinery Fluid Flow (BladeEditor) (Beta)

Taskgroup	Task	Input	Output
Turbomachinery Fluid Flow	Geometry	AnsoftCADObject	Geometry
		FEMSetup	

Taskgroup	Task	Input	Output
(BladeEditor) (Beta)		Geometry	
		ICEData	
		MechanicalResults	
		SimulationModelGeneratedPMDB	
		TurboGeometry	
	Turbo Mesh	TurboGeometry	TurboMesh
		Geometry	CFXMesh
			FluentImportable
	Setup	SimulationGeneratedMesh	CFXSetup
		CFXMesh	SystemCouplingSetupData
		MechanicalSetup	
	Solution	CFXSetup	CFXSolution
		CFXSolution	
	Results	CFXSolution	CFDAnalysis
		FluentSolution	
		ForteSolution	
		ICEData	
		IcePakResults	
		MechanicalSolution	
		NTIOutput	
		PolyflowSolutionType	
		VistaTFSolution	

Table 81: TurboGrid

Taskgroup	Task	Input	Output
TurboGrid	Turbo Mesh	TurboGeometry	TurboMesh
		Geometry	CFXMesh
			FluentImportable

Table 82: Vista TF

Taskgroup	Task	Input	Output
Vista TF	Setup	VistaGeometry	VistaTFSetup
		VistaTFPhysics	
		Geometry	
	Solution	VistaTFSetup	VistaTFSolution
		VistaTFSolution	
	Results	CFXSolution	CFDAnalysis
		FluentSolution	
		ForteSolution	

Taskgroup	Task	Input	Output
		ICEData	
		IcePakResults	
		MechanicalSolution	
		NTIOutput	
		PolyflowSolutionType	
		VistaTFSolution	

Table 83: Vista AFD

Taskgroup	Task	Input	Output
Vista AFD	Meanline	None	VistaAFDMeanlineProvider
	Design	VistaAFDMeanlineProvider	VistaAFDDesignProvider
	Analysis	VistaAFDDesignProvider	None

Table 84: Vista CCD

Taskgroup	Task	Input	Output
Vista CCD	Blade Design	None	VistaCCDBladeDesignProvider

Table 85: Vista CCD (with CCM)

Taskgroup	Task	Input	Output
Vista CCD (with CCM)	Blade Design	None	VistaCCDBladeDesignProvider
	Performance Map	VistaCCDBladeDesignProvider	None

Table 86: Vista CPD

Taskgroup	Task	Input	Output
Vista CPD	Blade Design	None	None

Table 87: Vista RTD

Taskgroup	Task	Input	Output
Vista RTD	Blade Design	None	None

Table 88: Vista RTD (Beta)

Taskgroup	Task	Input	Output
Vista RTD (Beta)	Blade Design	None	VistaGeometry

Taskgroup	Task	Input	Output
			VistaTFPhysics

Table 89: ACP (Pre)

Taskgroup	Task	Input	Output
ACP (Pre)	Engineering Data	AIMEngineeringDataMaterialOutputProvider	EngineeringData
		ExternalModelOutputProvider	Material
		FEMSetup	
		MatML31	
	Geometry	AnsoftCADObject	Geometry
		FEMSetup	
		Geometry	
		ICEData	
		MechanicalResults	
		SimulationModelGeneratedPMDB	
		TurboGeometry	
	Model	AIMGeometryMeshOutputProvider	MechanicalMesh
		AIMMechanicalPhysicsDefinitionOutputProvider	MechanicalModel
		CompositeEngineeringData	SimulationEngineeringData
		EngineeringData	SimulationGeneratedMesh
		EnhancedModelData	SimulationModelGeneratedMesh
		ExternalDataSetup	SimulationModelGeneratedPMDB
		ExternalMaterialFieldDataSetup	
		ExternalModelOutputProvider	
		ExternalTraceDataSetup	
		GeneralTransfer	
		Geometry	
		MeshingAssemblyTransferType	
		Modeler	
		SimulationEngineeringData	
		SimulationModelGeneratedMesh	
		SimulationSolutionOutputProvider	
		SolidSectionData	
	Setup	ACPSetupData	ACPSetupData
		EngineeringData	CompositeEngineeringData
		Geometry	EnhancedModelData
		SimulationEngineeringData	SimulationEngineeringData
		SimulationGeneratedMesh	SimulationModelGeneratedMesh

Taskgroup	Task	Input	Output
		SimulationModelGeneratedMesh	SolidSectionData

Table 90: ACP (Post)

Taskgroup	Task	Input	Output
ACP (Post)	Engineering Data	AIMEngineeringDataMaterialOutputProvider	EngineeringData
		ExternalModeACP (PrIOutputProvider	Material
		FEMSetup	
		MatML31	
	Geometry	AnsoftCADObject	Geometry
		FEMSetup	
		Geometry	
		ICEData	
		MechanicalResults	
		SimulationModelGeneratedPMDB	
		TurboGeometry	
	Model	AIMGeometryMeshOutputProvider	MechanicalMesh
		AIMMechanicalPhysicsDefinitionOutputProvider	MechanicalModel
		CompositeEngineeringData	SimulationEngineeringData
		EngineeringData	SimulationGeneratedMesh
		EnhancedModelData	SimulationModelGeneratedMesh
		ExternalDataSetup	SimulationModelGeneratedPMDB
		ExternalMaterialFieldDataSetup	
		ExternalModelOutputProvider	
		ExternalTraceDataSetup	
		GeneralTransfer	
		Geometry	
		MeshingAssemblyTransferType	
		Modeler	
		SimulationEngineeringData	
		SimulationModelGeneratedMesh	
		SimulationSolutionOutputProvider	
		SolidSectionData	
	Results	EngineeringData	
		MAPDLSolution	
		MechanicalSolution	

Taskgroup	Task	Input	Output
		SimulationGeneratedMesh	

Table 91: Maxwell 3D

Taskgroup	Task	Input	Output
Maxwell 3D	Geometry	AnsoftCADGeometryEntity	AnsoftCADObject
		AnsoftGeometryManagerData Object	AnsoftCellInOutEntity
		Geometry	AnsoftGeometryManagerDataObject
	Setup	AnsoftCellInOutEntity	MatML31
			AnsoftCellInOutEntity
	Solution	AnsoftCellInOutEntity	FeedbackIteratorSetup
			AnsoftForceAndMomentDataObject
			AnsoftForceDataObject
			AnsoftHeatLossDataObject

Table 92: Maxwell 2D

Taskgroup	Task	Input	Output
Maxwell 2D	Geometry	AnsoftCADGeometryEntity	AnsoftCADObject
		AnsoftGeometryManagerData Object	AnsoftCellInOutEntity
		Geometry	AnsoftGeometryManagerDataObject
	Setup	AnsoftCellInOutEntity	MatML31
			AnsoftCellInOutEntity
	Solution	AnsoftCellInOutEntity	FeedbackIteratorSetup
			AnsoftForceAndMomentDataObject
			AnsoftForceDataObject
			AnsoftHeatLossDataObject

Table 93: RMxprt

Taskgroup	Task	Input	Output
RMxprt	Setup		AnsoftCellInOutEntity
	Solution	AnsoftCellInOutEntity	

Table 94: Simpler

Taskgroup	Task	Input	Output
Simplorer	Setup	MechanicalSetup	AnsoftCellInOutEntity

Taskgroup	Task	Input	Output
	Solution	AnsoftCellInOutEntity	

Table 95: Turbo Setup

Taskgroup	Task	Input	Output
Turbo Setup	Turbo Setup		

Table 96: Turbo Machinery Fluid Flow (Bladegen) (Beta)

Taskgroup	Task	Input	Output
Turbomachinery Fluid Flow (BladeGen) (Beta)	Blade Design	None	TurboGeometry
			VistaGeometry
	Turbo Mesh	Geometry	CFXMesh
		TurboGeometry	FluentImportable
			TurboMesh
	Setup	CFXMesh	CFXSetup
		MechanicalSetup	SystemCouplingSetup Data
		SimulationGeneratedMesh	
	Solution	CFXSetup	CFXSolution
		CFXSolution	
	Results	CFXSolution	CFDAnalysis
		FluentSolution	
		ForteSolution	
		ICEData	
		IcePakResults	
		MechanicalSolution	
		NTIOoutput	
		PolyflowSolutionType	
		VistaTFSolution	

Table 97: Turbo Machinery Fluid Flow

Taskgroup	Task	Input	Output
Turbomachinery Fluid Flow	Turbo Mesh	Geometry	CFXMesh
		TurboGeometry	FluentImportable
			TurboMesh
	Setup	CFXMesh	CFXSetup
		MechanicalSetup	SystemCouplingSetup Data
		SimulationGeneratedMesh	
	Solution	CFXSetup	CFXSolution
		CFXSolution	

Taskgroup	Task	Input	Output
Results	CFXSolution		CFDAnalysis
	FluentSolution		
	ForteSolution		
	ICEData		
	IcePakResults		
	MechanicalSolution		
	NTIOoutput		
	PolyflowSolutionType		
	VistaTFSolution		

Table 98: Performance Map

Taskgroup	Task	Input	Output
Performance Map	Performance Map		

Table 99: Topology Optimization

Taskgroup	Task	Input	Output
Topology Optimization	Engineering Data	AIMEngineeringDataMaterialOutputProvider	EngineeringData
		ExternalModelOutputProvider	Material
		FEMSetup	
		MatML31	
	Geometry	AnsoftCADObject	Geometry
		FEMSetup	
		Geometry	
		ICEData	
		MechanicalResults	
		SimulationModelGeneratedPMDB	
		TurboGeometry	
	Model	AIMGeometryMeshOutputProvider	MechanicalMesh
		AIMMechanicalPhysicsDefinitionOutputProvider	MechanicalModel
		CompositeEngineeringData	SimulationEngineeringData
		EngineeringData	SimulationGeneratedMesh
		EnhancedModelData	SimulationModelGeneratedMesh
		ExternalDataSetup	SimulationModelGeneratedPMDB
		ExternalMaterialFieldDataSetup	
		ExternalModelOutputProvider	
		ExternalTraceDataSetup	
		GeneralTransfer	
		Geometry	

Taskgroup	Task	Input	Output
		MesherAssemblyTransferType	
		Modeler	
		SimulationEngineeringData	
		SimulationModelGeneratedMesh	
		SimulationSolutionOutputProvider	
		SolidSectionData	
	Setup	GeneralTransfer	MechanicalSetup
		MechanicalMesh	SimulationSetup
		MechanicalModel	
		SimulationSolutionDataInternal	
	Solution	GeneralTransfer	MechanicalSolution
		SimulationSetup	SimuationSolution SimuationSolutionDataInternal SimuationSolutionOutputProvider
	Results	SimulationSolution	MechanicalResults
			SimulationResults

Table 100: Designer Circuit

Taskgroup	Task	Input	Output
Designer Circuit	Setup		AnsoftCellInOutEntity
	Solution	AnsoftCellInOutEntity	

Table 101: Designer Circuit Netlist

Taskgroup	Task	Input	Output
Designer Circuit Netlist	Setup		AnsoftCellInOutEntity
	Solution	AnsoftCellInOutEntity	

Table 102: HFSS

Taskgroup	Task	Input	Output
HFSS	Geometry	AnsoftCADGeometryEntity	AnsoftCADOObject
		AnsoftGeometryManagerDataObject	AnsoftCellInOutEntity
		Geometry	AnsoftGeometryManagerDataObject
			MatML31
	Setup	AnsoftCellInOutEntity	AnsoftCellInOutEntity
			FeedbackIteratorSetup
	Solution	AnsoftCellInOutEntity	AnsoftForceDataObject

Taskgroup	Task	Input	Output
			AnsoftHeatLossDataObject

Table 103: HFSS 3D Layout Design

Taskgroup	Task	Input	Output
HFSS 3D Layout Design	Setup		AnsoftCellInOutEntity
	Solution	AnsoftCellInOutEntity	

Table 104: HFSS-IE

Taskgroup	Task	Input	Output
HFSS-IE	Geometry	AnsoftCADGeometryEntity	AnsoftCADOObject
		AnsoftGeometryManagerDataObject	AnsoftCellInOutEntity
		Geometry	AnsoftGeometryManagerDataObject
	Setup		MatML31
		AnsoftCellInOutEntity	AnsoftCellInOutEntity
	Solution	AnsoftCellInOutEntity	FeedbackIteratorSetup
			AnsoftHeatLossDataObject

Table 105: Q3D 2D Extractor

Taskgroup	Task	Input	Output
Q3D 2D Extractor	Geometry	AnsoftCADGeometryEntity	AnsoftCADOObject
		AnsoftGeometryManagerDataObject	AnsoftCellInOutEntity
		Geometry	AnsoftGeometryManagerDataObject
	Setup		MatML31
		AnsoftCellInOutEntity	AnsoftCellInOutEntity
	Solution	AnsoftCellInOutEntity	FeedbackIteratorSetup
			AnsoftHeatLossDataObject

Table 106: Q3D Extractor

Taskgroup	Task	Input	Output
Q3D Extractor	Geometry	AnsoftCADGeometryEntity	AnsoftCADOObject
		AnsoftGeometryManagerDataObject	AnsoftCellInOutEntity
		Geometry	AnsoftGeometryManagerDataObject
	Setup		MatML31
		AnsoftCellInOutEntity	AnsoftCellInOutEntity
	Solution	AnsoftCellInOutEntity	FeedbackIteratorSetup
			AnsoftHeatLossDataObject

Appendix B. ANSYS Workbench Internally Defined System Template and Component Names

Table 107: ACP (Pre)

Component Name	Component Display Name
EngDataCellTemplate	Engineering Data
GeometryCellTemplate	Geometry
SimulationModelCellTemplate	Model
ACPSetupCellTemplate	Setup

Table 108: ACP (Post)

Component Name	Component Display Name
EngDataCellTemplate	Engineering Data
GeometryCellTemplate	Geometry
SimulationModelCellTemplate	Model
ACPResultsCellTemplate	Results

Table 109: Feedback Iterator

Component Name	Component Display Name
FeedbackIteratorComponentTemplate	Feedback Iterator

Table 110: Mechanical APDL

Component Name	Component Display Name
ANSYSSetupCellTemplate	Analysis

Table 111: Hydrodynamic Diffraction

Component Name	Component Display Name
GeometryCellTemplate	Geometry
AQWAModelCellTemplate	Model
AQWASetupHDCellTemplate	Setup
AQWAAnalysisCellTemplate	Solution
AQWAResultsCellTemplate	Results

Table 112: Hydrodynamic Response

Component Name	Component Display Name
GeometryCellTemplate	Geometry

Component Name	Component Display Name
AQWAModelCellTemplate	Model
AQWASetupHDCellTemplate	Setup
AQWAAnalysisCellTemplate	Solution
AQWAResultsCellTemplate	Results

Table 113: Autodyn

Component Name	Component Display Name
AUTODYN_Solution	Setup
AUTODYN_Results	Analysis

Table 114: Results

Component Name	Component Display Name
CFDPostTemplate	Results

Table 115: CFX

Component Name	Component Display Name
CFXPhysicsTemplate	Setup
CFXResultsTemplate	Solution
CFDPostTemplate	Results

Table 116: Fluid Flow (CFX)

Component Name	Component Display Name
GeometryCellTemplate	Geometry
SimulationMeshingModelCellTemplate	Mesh
CFXPhysicsTemplate	Setup
CFXResultsTemplate	Solution
CFDPostTemplate	Results

Table 117: CFX (Beta)

Component Name	Component Display Name
CFXPhysicsTemplate	Setup
CFXResultsTemplate	Solution

Table 118: Response Surface Optimization

Component Name	Component Display Name
DXDOECellTemplate	Design of Experiments
DXResponseSurfaceCellTemplate	Response Surface

Component Name	Component Display Name
DXOptimizationCellTemplate_GDO	Optimization

Table 119: Parameters Correlation

Component Name	Component Display Name
DXCorrelationCellTemplate	DXCorrelationCellTemplate

Table 120: Direct Optimization

Component Name	Component Display Name
DXDirectOptimizationCellTemplate	Optimization

Table 121: Response Surface

Component Name	Component Display Name
DXDOECellTemplate	Design of Experiments
DXResponseSurfaceCellTemplate	Response Surface

Table 122: Six Sigma Analysis

Component Name	Component Display Name
DXDOECellForSixSigmaTemplate	Design of Experiments (SSA)
DXResponseSurfaceCellForSixSigmaTemplate	Response Surface (SSA)
DXSixSigmaCellTemplate	Six Sigma Analysis

Table 123: External Connection

Component Name	Component Display Name
ExternalConnectionTemplate	External Connection

Table 124: External Data

Component Name	Component Display Name
ExternalLoadSetupCellTemplate	Setup

Table 125: External Model

Component Name	Component Display Name
ExternalModelSetupCellTemplate	Setup

Table 126: Fluent (with Fluent Meshing)

Component Name	Component Display Name
FluentTGridCellTemplate	Mesh
FluentSetupCellTemplate	Setup

Component Name	Component Display Name
FluentResultsCellTemplate	Solution

Table 127: Fluent

Component Name	Component Display Name
FluentSetupCellTemplate	Setup
FluentResultsCellTemplate	Solution

Table 128: Fluid Flow (Fluent)

Component Name	Component Display Name
GeometryCellTemplate	Geometry
SimulationMeshingModelCellTemplate	Mesh
FluentSetupCellTemplate	Setup
FluentResultsCellTemplate	Solution
CFDPostTemplate	Results

Table 129: Fluent (with CFD-Post) (Beta)

Component Name	Component Display Name
FluentSetupCellTemplate	Setup
FluentResultsCellTemplate	Solution
CFDPostTemplate	Results

Table 130: ICEM CFD

Component Name	Component Display Name
ICEMCFD	Model

Table 131: IC Engine (Fluent)

Component Name	Component Display Name
ICEComponentTemplate	ICE
GeometryCellTemplate	Geometry
SimulationMeshingModelCellTemplate	Mesh
ICESetupComponentTemplate	ICE Setup
FluentSetupCellTemplate	Setup
FluentResultsCellTemplate	Solution
CFDPostTemplate	Results

Table 132: IC Engine (Forte)

Component Name	Component Display Name
ICEComponentTemplate	ICE
GeometryCellTemplate	Geometry
SimulationMeshingModelCellTemplate	Mesh

Component Name	Component Display Name
ICESetupComponentTemplate	ICE Setup
CFDPostTemplate	Results

Table 133: Icepak

Component Name	Component Display Name
IcePakSetupCellTemplate	Setup
IcePakSolutionCellTemplate	Solution

Table 134: Mechanical Model

Component Name	Component Display Name
EngDataCellTemplate	Engineering Data
GeometryCellTemplate	Geometry
SimulationModelCellTemplate	Model

Table 135: Electric

Component Name	Component Display Name
EngDataCellTemplate	Engineering Data
GeometryCellTemplate	Geometry
SimulationModelCellTemplate	Model
SimulationSetupCellTemplate_ElectricStaticANSYS	Setup
SimulationSolutionCellTemplate_ElectricStaticANSYS	Solution
SimulationResultsCellTemplate_ElectricStaticANSYS	Results

Table 136: Eigenvalue Buckling

Component Name	Component Display Name
EngDataCellTemplate	Engineering Data
GeometryCellTemplate	Geometry
SimulationModelCellTemplate	Model
SimulationSetupCellTemplate_StructuralBucklingANSYS	Setup
SimulationSolutionCellTemplate_StructuralBucklingANSYS	Solution
SimulationResultsCellTemplate_StructuralBucklingANSYS	Results

Table 137: Eigenvalue Buckling (Samcef)

Component Name	Component Display Name
EngDataCellTemplate	Engineering Data
GeometryCellTemplate	Geometry
SimulationModelCellTemplate	Model
SimulationSetupCellTemplate_StructuralBucklingSamcef	Setup
SimulationSolutionCellTemplate_StructuralBucklingSamcef	Solution

Component Name	Component Display Name
SimulationResultsCellTemplate_StructuralBucklingSamcef	Results

Table 138: Magnetostatic

Component Name	Component Display Name
EngDataCellTemplate	Engineering Data
GeometryCellTemplate	Geometry
SimulationModelCellTemplate	Model
SimulationSetupCellTemplate_ElectromagneticsMagnetostaticsANSYS	Setup
SimulationSolutionCellTemplate_ElectromagneticsMagnetostaticsANSYS	Solution
SimulationResultsCellTemplate_ElectromagneticsMagnetostaticsANSYS	Results

Table 139: Modal

Component Name	Component Display Name
EngDataCellTemplate	Engineering Data
GeometryCellTemplate	Geometry
SimulationModelCellTemplate	Model
SimulationSetupCellTemplate_StructuralModalANSYS	Setup
SimulationSolutionCellTemplate_StructuralModalANSYS	Solution
SimulationResultsCellTemplate_StructuralModalANSYS	Results

Table 140: Modal (Samcef)

Component Name	Component Display Name
EngDataCellTemplate	Engineering Data
GeometryCellTemplate	Geometry
SimulationModelCellTemplate	Model
SimulationSetupCellTemplate_StructuralModalSamcef	Setup
SimulationSolutionCellTemplate_StructuralModalSamcef	Solution
SimulationResultsCellTemplate_StructuralModalSamcef	Results

Table 141: Modal (NASTRAN) (Beta)

Component Name	Component Display Name
EngDataCellTemplate	Engineering Data
GeometryCellTemplate	Geometry
SimulationModelCellTemplate	Model
SimulationSetupCellTemplate_StructuralModalNASTRAN	Setup
SimulationSolutionCellTemplate_StructuralModalNASTRAN	Solution

Component Name	Component Display Name
SimulationResultsCellTemplate_StructuralModalNASTRAN	Results

Table 142: Modal (ABAQUS)

Component Name	Component Display Name
EngDataCellTemplate	Engineering Data
GeometryCellTemplate	Geometry
SimulationModelCellTemplate	Model
SimulationSetupCellTemplate_StructuralModalABAQUS	Setup
SimulationSolutionCellTemplate_StructuralModalABAQUS	Solution
SimulationResultsCellTemplate_StructuralModalABAQUS	Results

Table 143: Random Vibration

Component Name	Component Display Name
EngDataCellTemplate	Engineering Data
GeometryCellTemplate	Geometry
SimulationModelCellTemplate	Model
SimulationSetupCellTemplate_StructuralRandomVibrationANSYS	Setup
SimulationSolutionCellTemplate_StructuralRandomVibrationANSYS	Solution
SimulationResultsCellTemplate_StructuralRandomVibrationANSYS	Results

Table 144: Response Spectrum

Component Name	Component Display Name
EngDataCellTemplate	Engineering Data
GeometryCellTemplate	Geometry
SimulationModelCellTemplate	Model
SimulationSetupCellTemplate_StructuralResponseSpectrumANSYS	Setup
SimulationSolutionCellTemplate_StructuralResponseSpectrumANSYS	Solution
SimulationResultsCellTemplate_StructuralResponseSpectrumANSYS	Results

Table 145: Topology Optimization

Component Name	Component Display Name
EngDataCellTemplate	Engineering Data
GeometryCellTemplate	Geometry
SimulationModelCellTemplate	Model
SimulationSetupCellTemplate_StructuralShapeOptimizationANSYS	Setup
SimulationSolutionCellTemplate_StructuralShapeOptimizationANSYS	Solution

Component Name	Component Display Name
SimulationResultsCellTemplate_StructuralShapeOptimizationANSYS	Results

Table 146: Shape Optimization (Beta)

Component Name	Component Display Name
EngDataCellTemplate	Engineering Data
GeometryCellTemplate	Geometry
SimulationModelCellTemplate	Model
SimulationSetupCellTemplate_StructuralShapeOptimizationANSYS	Setup
SimulationSolutionCellTemplate_StructuralShapeOptimizationANSYS	Solution
SimulationResultsCellTemplate_StructuralShapeOptimizationANSYS	Results

Table 147: Static Structural

Component Name	Component Display Name
EngDataCellTemplate	Engineering Data
GeometryCellTemplate	Geometry
SimulationModelCellTemplate	Model
SimulationSetupCellTemplate_StructuralStaticANSYS	Setup
SimulationSolutionCellTemplate_StructuralStaticANSYS	Solution
SimulationResultsCellTemplate_StructuralStaticANSYS	Results

Table 148: Static Structural (ABAQUS)

Component Name	Component Display Name
EngDataCellTemplate	Engineering Data
GeometryCellTemplate	Geometry
SimulationModelCellTemplate	Model
SimulationSetupCellTemplate_StructuralStaticABAQUS	Setup
SimulationSolutionCellTemplate_StructuralStaticABAQUS	Solution
SimulationResultsCellTemplate_StructuralStaticABAQUS	Results

Table 149: Static Structural (Samcef)

Component Name	Component Display Name
EngDataCellTemplate	Engineering Data
GeometryCellTemplate	Geometry
SimulationModelCellTemplate	Model
SimulationSetupCellTemplate_StructuralStaticSamcef	Setup
SimulationSolutionCellTemplate_StructuralStaticSamcef	Solution

Component Name	Component Display Name
SimulationResultsCellTemplate_StructuralStaticSamcef	Results

Table 150: Steady-State Thermal

Component Name	Component Display Name
EngDataCellTemplate	Engineering Data
GeometryCellTemplate	Geometry
SimulationModelCellTemplate	Model
SimulationSetupCellTemplate_ThermalSteadyStateANSYS	Setup
SimulationSolutionCellTemplate_ThermalSteadyStateANSYS	Solution
SimulationResultsCellTemplate_ThermalSteadyStateANSYS	Results

Table 151: Steady-State Thermal (Samcef)

Component Name	Component Display Name
EngDataCellTemplate	Engineering Data
GeometryCellTemplate	Geometry
SimulationModelCellTemplate	Model
SimulationSetupCellTemplate_ThermalSteadyStateSamcef	Setup
SimulationSolutionCellTemplate_ThermalSteadyStateSamcef	Solution
SimulationResultsCellTemplate_ThermalSteadyStateSamcef	Results

Table 152: Steady-State Thermal (ABAQUS)

Component Name	Component Display Name
EngDataCellTemplate	Engineering Data
GeometryCellTemplate	Geometry
SimulationModelCellTemplate	Model
SimulationSetupCellTemplate_ThermalSteadyStateABAQUS	Setup
SimulationSolutionCellTemplate_ThermalSteadyStateABAQUS	Solution
SimulationResultsCellTemplate_ThermalSteadyStateABAQUS	Results

Table 153: Transient Structural (ABAQUS)

Component Name	Component Display Name
EngDataCellTemplate	Engineering Data
GeometryCellTemplate	Geometry
SimulationModelCellTemplate	Model
SimulationSetupCellTemplate_StructuralTransientABAQUS	Setup
SimulationSolutionCellTemplate_StructuralTransientABAQUS	Solution

Component Name	Component Display Name
SimulationResultsCellTemplate_StructuralTransientABAQUS	Results

Table 154: Transient Structural (Samcef)

Component Name	Component Display Name
EngDataCellTemplate	Engineering Data
GeometryCellTemplate	Geometry
SimulationModelCellTemplate	Model
SimulationSetupCellTemplate_StructuralTransientSamcef	Setup
SimulationSolutionCellTemplate_StructuralTransientSamcef	Solution
SimulationResultsCellTemplate_StructuralTransientSamcef	Results

Table 155: Transient Structural

Component Name	Component Display Name
EngDataCellTemplate	Engineering Data
GeometryCellTemplate	Geometry
SimulationModelCellTemplate	Model
SimulationSetupCellTemplate_StructuralTransientANSYS	Setup
SimulationSolutionCellTemplate_StructuralTransientANSYS	Solution
SimulationResultsCellTemplate_StructuralTransientANSYS	Results

Table 156: Rigid Dynamics

Component Name	Component Display Name
EngDataCellTemplate	Engineering Data
GeometryCellTemplate	Geometry
SimulationModelCellTemplate	Model
SimulationSetupCellTemplate_StructuralTransientRBD	Setup
SimulationSolutionCellTemplate_StructuralTransientRBD	Solution
SimulationResultsCellTemplate_StructuralTransientRBD	Results

Table 157: Explicit Dynamics

Component Name	Component Display Name
EngDataCellTemplate	Engineering Data
GeometryCellTemplate	Geometry
SimulationModelCellTemplate	Model
SimulationSetupCellTemplate_StructuralExplicitAUTODYN	Setup
SimulationSolutionCellTemplate_StructuralExplicitAUTODYN	Solution

Component Name	Component Display Name
SimulationResultsCellTemplate_StructuralExplicitAUTODYN	Results

Table 158: Thermal-Electric

Component Name	Component Display Name
EngDataCellTemplate	Engineering Data
GeometryCellTemplate	Geometry
SimulationModelCellTemplate	Model
SimulationSetupCellTemplate_ThermalElectromagneticsStaticANSYS	Setup
SimulationSolutionCellTemplate_ThermalElectromagneticsStaticANSYS	Solution
SimulationResultsCellTemplate_ThermalElectromagneticsStaticANSYS	Results

Table 159: Explicit Dynamics (LS-DYNA Export) (Beta)

Component Name	Component Display Name
EngDataCellTemplate	Engineering Data
GeometryCellTemplate	Geometry
SimulationModelCellTemplate	Model
SimulationSetupCellTemplate_StructuralExplicitLSDYNA	Setup

Table 160: Transient Thermal

Component Name	Component Display Name
EngDataCellTemplate	Engineering Data
GeometryCellTemplate	Geometry
SimulationModelCellTemplate	Model
SimulationSetupCellTemplate_ThermalTransientANSYS	Setup
SimulationSolutionCellTemplate_ThermalTransientANSYS	Solution
SimulationResultsCellTemplate_ThermalTransientANSYS	Results

Table 161: Transient Thermal (ABAQUS)

Component Name	Component Display Name
EngDataCellTemplate	Engineering Data
GeometryCellTemplate	Geometry
SimulationModelCellTemplate	Model
SimulationSetupCellTemplate_ThermalTransientABAQUS	Setup
SimulationSolutionCellTemplate_ThermalTransientABAQUS	Solution
SimulationResultsCellTemplate_ThermalTransientABAQUS	Results

Table 162: Transient Thermal (Samcef)

Component Name	Component Display Name
EngDataCellTemplate	Engineering Data

Component Name	Component Display Name
GeometryCellTemplate	Geometry
SimulationModelCellTemplate	Model
SimulationSetupCellTemplate_ThermalTransientSamcef	Setup
SimulationSolutionCellTemplate_ThermalTransientSamcef	Solution
SimulationResultsCellTemplate_ThermalTransientSamcef	Results

Table 163: Design Assessment

Component Name	Component Display Name
EngDataCellTemplate	Engineering Data
GeometryCellTemplate	Geometry
SimulationModelCellTemplate	Model
SimulationSetupCellTemplate_CustomizableDesignAssessmentANSYS	Setup
SimulationSolutionCellTemplate_CustomizableDesignAssessmentANSYS	Solution
SimulationResultsCellTemplate_CustomizableDesignAssessmentANSYS	Results

Table 164: Engineering Data

Component Name	Component Display Name
EngDataCellTemplate	Engineering Data

Table 165: Microsoft Office Excel

Component Name	Component Display Name
MSExcelComponentTemplate	Analysis

Table 166: System Coupling

Component Name	Component Display Name
SystemCouplingSetupCellTemplate	Setup
SystemCouplingSolutionCellTemplate	Solution

Table 167: Polyflow

Component Name	Component Display Name
PolyflowSetupCellTemplate	Setup
PolyflowSolveCellTemplate	Solution

Table 168: Fluid Flow (Polyflow)

Component Name	Component Display Name
GeometryCellTemplate	Geometry
SimulationMeshingModelCellTemplate	Mesh
PolyflowSetupCellTemplate	Setup
PolyflowSolveCellTemplate	Solution

Component Name	Component Display Name
CFDPostTemplate	Results

Table 169: Fluid Flow - Extrusion (Polyflow)

Component Name	Component Display Name
GeometryCellTemplate	Geometry
SimulationMeshingModelCellTemplate	Mesh
PolyflowSetupCellTemplate	Setup
PolyflowSolveCellTemplate	Solution
CFDPostTemplate	Results

Table 170: Polyflow - Extrusion

Component Name	Component Display Name
PolyflowSetupCellTemplate	Setup
PolyflowSolveCellTemplate	Solution

Table 171: Fluid Flow - Blow Molding (Polyflow)

Component Name	Component Display Name
GeometryCellTemplate	Geometry
SimulationMeshingModelCellTemplate	Mesh
PolyflowSetupCellTemplate	Setup
PolyflowSolveCellTemplate	Solution
CFDPostTemplate	Results

Table 172: Polyflow - Blow Molding

Component Name	Component Display Name
PolyflowSetupCellTemplate	Setup
PolyflowSolveCellTemplate	Solution

Table 173: Throughflow

Component Name	Component Display Name
GeometryCellTemplate	Geometry
VistaTFSetupTemplate	Setup
VistaTFSolutionTemplate	Solution
CFDPostTemplate	Results

Table 174: Throughflow (BladeGen)

Component Name	Component Display Name
TSGeometryTemplate	Blade Design
VistaTFSetupTemplate	Setup
VistaTFSolutionTemplate	Solution

Component Name	Component Display Name
CFDPostTemplate	Results

Table 175: Vista TF

Component Name	Component Display Name
VistaTFSetupTemplate	Setup
VistaTFSolutionTemplate	Solution
CFDPostTemplate	Results

Table 176: BladeGen

Component Name	Component Display Name
TSGeometryTemplate	Blade Design

Table 177: BladeGen (Beta)

Component Name	Component Display Name
TSGeometryTemplateBeta	Blade Design

Table 178: TurboGrid

Component Name	Component Display Name
TSMeshTemplate	Turbo Mesh

Table 179: Turbo Setup

Component Name	Component Display Name
TSSetupTemplate	Turbo Setup

Table 180: Vista RTD

Component Name	Component Display Name
TSVistaRTDTemplate	Blade Design

Table 181: Vista RTD (Beta)

Component Name	Component Display Name
TSVistaRTDTemplate	Blade Design

Table 182: Vista CCD

Component Name	Component Display Name
TSVistaCCDTemplate	Blade Design

Table 183: Vista CCD (with CCM)

Component Name	Component Display Name
TSVistaCCDTemplate	Blade Design

Component Name	Component Display Name
TSVistaCCMTemplate	Performance Map

Table 184: Vista CPD

Component Name	Component Display Name
TSVistaCPDTemplate	Blade Design

Table 185: Vista AFD

Component Name	Component Display Name
TSVistaAFDMeanlineTemplate	Meanline
TSVistaAFDDesignTemplate	Design
TSVistaAFDAnalysisTemplate	Analysis

Table 186: Turbomachinery Fluid Flow (BladeEditor) (Beta)

Component Name	Component Display Name
GeometryCellTemplate	Geometry
TSMeshTemplate	Turbo Mesh
CFXPhysicsTemplate	Setup
CFXResultsTemplate	Solution
CFDPostTemplate	Results

Table 187: Turbomachinery Fluid Flow (BladeGen) (Beta)

Component Name	Component Display Name
TSGeometryTemplate	Blade Design
TSMTSMeshTemplate	Turbo Mesh
CFXPhysicsTemplate	Setup
CFXResultsTemplate	Solution
CFDPostTemplate	Results

Table 188: Mesh

Component Name	Component Display Name
GeometryCellTemplate	Blade Design

Component Name	Component Display Name
SimulationMeshingModelCellTemplate	Mesh

Table 189: Geometry

Component Name	Component Display Name
GeometryCellTemplate	Geometry

Table 190: Finite Element Modeler

Component Name	Component Display Name
FESetupCellTemplate	Model

Table 191: Performance Map

Component Name	Component Display Name
PerfMapTemplate	Performance Map

Table 192: Designer Circuit

Component Name	Component Display Name
DesignerCircuitSetup	Setup
DesignerCircuitSolution	Solution

Table 193: Designer Circuit Netlist

Component Name	Component Display Name
NexximNetlistSetup	Setup
NexximNetlistSolution	Solution

Table 194: Fluent (with Fluent Meshing)

Component Name	Component Display Name
FluentTGridCellTemplate	Mesh
FluentSetupCellTemplate	Setup
FluentResultsCellTemplate	Solution

Table 195: HFSS

Component Name	Component Display Name
HFSSGeometry	Geometry
HFSSSetup	Setup
HFSSSolution	Solution

Table 196: HFSS 3D Layout Design

Component Name	Component Display Name
DesignerEMSetup	Setup

Component Name	Component Display Name
DesignerEMSSolution	Solution

Table 197: HFSS-IE

Component Name	Component Display Name
HFSS-IEGeometry	Geometry
HFSS-IESetup	Setup
HFSS-IESolution	Solution

Table 198: Maxwell 2D

Component Name	Component Display Name
Maxwell2DGeometry	Geometry
Maxwell2DSetup	Setup
Maxwell2DSolution	Solution

Table 199: Maxwell 3D

Component Name	Component Display Name
Maxwell3DGeometry	Geometry
Maxwell3DSetup	Setup
Maxwell3DSolution	Solution

Table 200: Modal Acoustics (Beta)

Component Name	Component Display Name
EngDataCellTemplate	Engineering Data
GeometryCellTemplate	Geometry
SimulationModelCellTemplate	Model
SimulationSetupCellTemplate_MultiphysicsModalANSYS	Setup
SimulationSolutionCellTemplate_MultiphysicsModalANSYS	Solution
SimulationResultsCellTemplate_MultiphysicsModalANSYS	Results

Table 201: Q3D 2D Extractor

Component Name	Component Display Name
Q3D2DExtractorGeometry	Geometry
Q3D2DExtractorSetup	Setup
Q3D2DExtractorSolution	Solution

Table 202: Q3D Extractor

Component Name	Component Display Name
Q3DExtractorGeometry	Geometry
Q3DExtractorSetup	Setup

Component Name	Component Display Name
Q3DExtractorSolution	Solution

Table 203: RMxprt

Component Name	Component Display Name
RMxprtSetup	Setup
RMxprtSolution	Solution

Table 204: Simpler

Component Name	Component Display Name
SimplorerSetup	Setup
SimplorerSolution	Solution

Table 205: Turbomachinery Fluid Flow

Component Name	Component Display Name
TSMeshTemplate	Turbo Mesh
CFXPhysicsTemplate	Setup
CFXResultsTemplate	Solution
CFDPostTemplate	Results

Table 206: Harmonic Acoustics

Component Name	Component Display Name
EngDataCellTemplate	Engineering Data
GeometryCellTemplate	Geometry
SimulationModelCellTemplate	Model
SimulationSetupCellTemplate_MultiphysicsHarmonicANSYS	Setup
SimulationSolutionCellTemplate_MultiphysicsHarmonicANSYS	Solution
SimulationResultsCellTemplate_MultiphysicsHarmonicANSYS	Results

Appendix C. Data Transfer Types

Table 207: Data Transfer Types and Properties

Transfer Type	Property
ACPSetupData	ACPFileReference
	ACPPreFileReference
AIMEngineeringDataMaterialOutputProvider	
AIMFluentMeshOutputProvider	
AIMGeometryMeshOutputProvider	
AnsoftCADObject	
AnsoftHeatLossDataObject	AnsoftTransferXMLString
	AnsoftProjectResultsFolderAtCurrentDP
AnsoftForceDataObject	AnsoftTransferXMLString
	AnsoftProjectResultsFolderAtCurrentDP
AnsoftForceAndMomentDataObject	AnsoftTransferXMLString
	AnsoftProjectResultsFolderAtCurrentDP
FeedbackIteratorEntity	
FeedbackIteratorSetup	
MAPDLSolution	TransferFile
	AuxiliaryFiles
MAPDLDatabase	TransferFile
	AuxiliaryFiles
MAPDLResults	AuxiliaryFiles
MAPDLCdb	TransferFile
	AuxiliaryFiles
AqwaModel	
AqwaSetup	
AqwaSolution	
AqwaResults	
AutodynSetup	
CFDAnalysis	PostStateFile
CFXSetup	CFXSolverInputFile
	MAPDLSolverInputFile
CFXSolution	MResLoadOption
	CFXResultsFile
	AuxiliaryFiles

Transfer Type	Property
	MAPDLResultsFile
Geometry	GeometryFilePath
	PlugInName
ParametricContext	
DOEModel	
DesignPointsDataTransfer	
ResponseSurfaceModel	
ResponseSurfaceDataTransfer	
OptimizationModel	
CorrelationModel	
SixSigmaModel	
EngineeringData	TransferFile
Material	
ExternalConnectionProperties	
ExternalDataSetup	TransferFile
ExternalModelOutputProvider	TransferFile
	InputFiles
SolidSectionData	TransferFile
	AuxiliaryFiles
	CompositeSectionFiles
EnhancedMechanicalModel	
EnhancedModelData	
FEMMesh	ACMOFile
FEMSetup	FEModelerFile
	ANSYSInputFile
	ParasolidFile
	FiniteElementModelMaterials
	AuxiliaryFiles
FluentTGridMesh	TransferFile
FluentSetup	CaseFile
	ModelInfoFile
SystemCouplingSetupData	
FluentCase	MeshFile
	TransferFile
FluentSolution	CaseFile
	DataFile
ICEData	
ICESetupData	

Transfer Type	Property
IcePakSetup	
IcePakResults	
MechanicalModel	File EdaFile
MesherGeneratedMeshOutputProvider	PMDBFile ACMOFile Mechdb
MeshingMesh	TransferFile
SimulationGeneralMesh	TransferFile
SimulationGeneratedMesh	TransferFile
MSExcelSetup	
CouplingSetupProvider	TransferFile
PolyflowSetup	
PolyflowSolutionType	DataFile PubFile GeneratedFiles
PolyflowSolution	
MechanicalMesh	TransferFile
SimulationEngineeringData	TransferFiles
SimulationModelGeneratedMesh	TransferFile
SimulationSetup	
MechanicalSetup	TransferFile
MechanicalSolution	
SimulationSolutionDataInternal	
SimulationSolution	
MechanicalResults	
SimulationResults	
TurboGeometry	INFFilename GeometryFilename
TurboMesh	FileName
CFXMesh	FileName PreFileType
FluentImportable	MeshFile FileType Dimension
VistaGeometry	GeoData TransferData
VistaTFPhysics	TransferData

Transfer Type	Property
VistaCCDBladeDesignProvider	TransferData
VistaAFDMeanlineProvider	TransferData
VistaAFDDesignProvider	TransferData
VistaTFSetup	ControlFilename
	GeoFilename
	AeroFilename
	CorrelationsFilename
VistaTFSolution	ResultsFile
	RestartFile
AUTODYN_Remap	
MatML31	TransferFile
CompositeEngineeringData	TransferFile
FluentMesh	TransferFile
PolyflowTransferMesh	TransferFile
ExternalTraceDataSetup	
ForteSetupData	DataFile
ForteSMGData	
ForteSolution	
ForteSolutionData	
Imported FLUENT Mesh File Type	MeshFile
Modeler	
SimulationSolutionOutputProvider	
GeneralTransfer	
SimulationModelGeneratedPMDB	TransferFile
ExternalMaterialFieldDataSetup	
AIMFluentPartMeshFileOutputProvider	
AIMFluentPartMeshOutputProvider	
GeneratedMeshOutputForAQWAModelProvider	PMDBFile
	ACMOFile
	Mechdb
MeshtingAssemblyTransferType	TransferFile
AIMFluentPhysicsDefinitionOutputProvider	
AIMMechanicalPhysicsDefinitionOutputProvider	
AnsoftCADGeometryEntity	
AnsoftCellInOutEntity	
AnsoftGeometryManagerDataObject	
ExternalDataSetupForAqwa	
NTIOutput	

Appendix D. Addin Data Types and Data Transfer Formats

The following table lists the data types and data transfer formats supported for each addin.

Note

Where listed, a value of "n/a" indicates that you should use an empty format string.

Addin	Data Type	Format
AIM	AIMEngineeringDataMaterialOutputProvider	n/a
	AIMFluentPartMeshFileOutputProvider	n/a
	AIMFluentPartMeshOutputProvider	n/a
	AIMFluentPhysicsDefinitionOutputProvider	n/a
	AIMMechanicalPhysicsDefinitionOutputProvider	n/a
ANSYS	MAPDLCdb	n/a
	MAPDLDatabase	n/a
	MAPDLSolution	n/a
	MAPDLResults	n/a
AQWA	AQWAModel	transfer not supported
	AQWASetup	transfer not supported
	AQWASolution	transfer not supported
	AQWAResults	transfer not supported
AUTODYN	AutodynSetup	transfer not supported
	AutodynAnalysis	transfer not supported
CFX	CFXSetup	n/a
	SystemCouplingSetupData	n/a
	CFXSolution	n/a
DesignModeler	Geometry	n/a
DX		transfer not supported
EBU (Q3D, Maxwell, HFSS)	AnsoftCADGeometryEntity	n/a
	AnsoftCADObject	n/a
	AnsoftCellInOutEntity	n/a
	AnsoftForceDataObject	n/a
	AnsoftForceAndMomentDataObject	n/a
	AnsoftGeometryDataObject	n/a
	AnsoftGeometryManagerDataObject	n/a
	AnsoftHeatLossDataObject	n/a

Addin	Data Type	Format
EKM	No Output Types	
Engineering Data	Engineering Data	materials
External Load	ExternalDataSetup	ExternalDataSetup
	ExternalDataSetupForAqwa	ExternalDataSetupForAqwa
	ExternalTraceDataSetup	ExternalTraceDataSetup
	ExternalModelOutputProvider	ExternalModelOutputProvider
	ExternalMaterialFieldDataSetup	ExternalMaterialFieldDataSetup
ExternalModelCoupling	SolidSectionData	n/a
	EnhancedModelData	EnhancedModelData
FEModeler	Geometry	
	FEMMesh	ACMOFile
	FEMSetup	FacetedGeometry
		ParasolidGeometry
		InputFile
		FiniteElementModelMaterial
Fluent	FluentTGridMesh	n/a
	FluentSetup	transfer not supported
	SystemCouplingSetupData	SystemCouplingSetupData
	FluentSolution	n/a
Forte	ForteSolution	transfer not supported
	ForteSolutionData	transfer not supported
	ForteSMGData	transfer not supported
	ForteSetupData	n/a
ICEMCFD	SimulationGeneratedMesh	FluentMesh
		Imported FLUENT Mesh FileType
		CMDBMesh
		POLYFLOWMesh
		AnsysMesh
	MeshingAssemblyTransferType	FluentMesh
		Imported FLUENT Mesh FileType
		CMDBMesh
		POLYFLOWMesh
		AnsysMesh
ICEngine	ICEData	ICEData
	ICESetupData	n/a
IcePak	IcePakSetup	transfer not supported
	IcePakResults	n/a

Addin	Data Type	Format
Mesher	MechanicalModel	transfer not supported
	MesherMesh	CMDBMesh
		FluentMesh
		POLYFLOWMesh
	SimulationGeneratedMesh	CMDBMesh
		FluentMesh
		POLYFLOWMesh
	MeshingGeneratedMeshOutputProvider	MeshingGeneratedMeshOutputProvider
	GeneratedMeshForAQWAModelProvider	n/a
MSExcel	MSExcelSetup	transfer not supported
Multiphysics Coupling	CouplingSetupProvider	CouplingSetupProvider
NTI	NTIOOutput	n/a
Polyflow	PolyflowSetup	transfer not supported
	PolyflowSolutionType	n/a
	PolyflowSolution	transfer not supported
	ExternalDataSetup	ExternalDataSetup
Simulation	MechanicalModel	transfer not supported
	MechanicalMesh	CMDBMesh
		FluentMesh
		POLYFLOWMesh
	SimulationGeneratedMesh	CMDBMesh
		FluentMesh
		POLYFLOWMesh
	SimulationEngineeringData	SimulationEngineeringData
	SimulationModelGeneratedMesh	SimulationModelGeneratedMesh
	SimulationModelGeneratedPMDB	SimulationModelGeneratedPMDB
	SimulationSetup	n/a
	MechanicalSetup	n/a
	MechanicalSolution	n/a
	SimulationSolutionDataInternal	transfer not supported
	SimulationSolution	transfer not supported
	SimulationSolutionOutputProvider	SimulationSolutionOutputProvider
	SimulationResults	n/a
	MechanicalResults	n/a
	GeneralTransfer	n/a
TurboSystem	VistaGeometry	n/a
	TurboMesh	transfer not supported
	CFXMesh	n/a

Addin	Data Type	Format
	FluentImportable	n/a
	VistaATFPhysics	n/a
	VistaCCDBladeDesignProvider	n/a
	VistaAFDMeanlineProvider	n/a
	VistaAFDDesignProvider	n/a
VistaTF	VistaTFSetup	n/a
	VistaTFSolution	n/a

Appendix E. Pre-Installed Custom Workflow Support

The following components can consume general data transfer provided by ACT custom tasks. For an explanation of general data transfer, see [Defining Task-Level General Data Transfer \(p. 125\)](#).

Table 208: Task Groups and Tasks

Task Groups	Tasks
ACP (Pre)	Model
ACP (Post)	Model
Mechanical Model	Model
Electric	Model
	Setup
	Solution
Eigenvalue Buckling	Model
	Setup
	Solution
Eigenvalue Buckling (Samcef)	Model
	Setup
	Solution
Magnetostatic	Model
	Setup
	Solution
Modal	Model
	Setup
	Solution
Modal (Samcef)	Model
	Setup
	Solution
Modal (NASTRAN) (Beta)	Model
	Setup
	Solution
Modal (ABAQUS)	Model
	Setup
	Solution
Random Vibration	Model
	Setup

Task Groups	Tasks
	Solution
Response Spectrum	Model
	Setup
	Solution
Topological Optimization (Beta)	Model
	Setup
	Solution
Shape Optimization (Beta)	Model
	Setup
	Solution
Static Structural	Model
	Setup
	Solution
Static Structural (ABAQUS)	Model
	Setup
	Solution
Static Structural (Samcef)	Model
	Setup
	Solution
Steady-State Thermal	Model
	Setup
	Solution
Steady-State Thermal (Samcef)	Model
	Setup
	Solution
Steady-State Thermal (ABAQUS)	Model
	Setup
	Solution
Transient Structural (ABAQUS)	Model
	Setup
	Solution
Transient Structural (Samcef)	Model
	Setup
	Solution
Transient Structural	Model
	Setup
	Solution
Rigid Dynamics	Model

Task Groups	Tasks
	Setup
	Solution
Explicit Dynamics	Model
	Setup
	Solution
Thermal-Electric	Model
	Setup
	Solution
Explicit Dynamics (LS-DYNA Export)	Model
	Setup
	Solution
Transient Thermal	Model
	Setup
	Solution
Transient Thermal (ABAQUS)	Model
	Setup
	Solution
Transient Thermal (Samcef)	Model
	Setup
	Solution
Design Assessment	Model
	Setup
	Solution

