Simio Simulation & Scheduling Software

API OVERVIEW

Simio

Forward Thinking

Table of Contents

Simio
Forward Thinking

03 Customizing using the API

If you already know how to build models using the Standard Library, extend the Standard Library using add-on processes, and build your own custom objects, then the next level of modeling might be to explore what custom coding can do for you. Relatively few Simio users need to do custom programming because the features just mentioned are so rich. But if you have at least a moderate level of programming skills, Simio allows you to actually extend the underlying Simio simulation engine itself to further extend and customize the environment to meet your needs.

Simio's architecture provides several extension points where users can integrate their own custom functionality written in a .NET language such as Visual C# or Visual Basic .NET. The types of user extensions supported include:

- User Defined Steps
- User Defined Elements
- User Defined Selection Rules
- Design Time Add-Ins including new buttons on the ribbons
- Table Imports and Binding
- Design or Run Experiments

Simio's extension points have been exposed as a set of interfaces that describe the methods and calling conventions to be implemented by any user extended components. This set of interfaces is referred to as the Simio Application Programming Interface (API). For detailed information on the Simio API, see the file Simio API Reference Guide.chm located where Simio was installed (typically C:\Program Files (x86)\Simio).

While you can create extensions using any .NET programming language, Simio provides extra support for C# users. To help you get started creating Simio user extensions, a number of predefined Simio project and project item templates for Microsoft Visual Studio are available. These templates provide reusable and customizable project and item stubs that may be used to accelerate the development process, removing the need to create new projects and items from scratch.

In addition, several user extension examples are included with the Simio installation. You can explore these examples and customize them to solve your own problems. The examples include:

Binary Gate
An element and three steps for controlling ow through a gate

TextFileReadWrite
An element and two steps for reading and writing text files

ExcelReadWrite
An element and two steps for reading and writing Excel files

DbReadWrite
An element and four steps for reading and writing database files

CSVGridDataProvider
Supports import and export between tables and text files

ExcelGridDataProvider
Supports import and export between tables and Excel files

SelectBestScenario
Illustrates an experiment data analysis add-in

SimioSelectionRules
Contains the implementation of Simio's dynamic selection rules

SourceServerSink
Illustrates a design time add-in that builds a simple facility model

SimioTravelSteeringBehaviors
Supports guidance of entities moving through free space

SimioScheduling
Configure resources, lists, and tables for scheduling applications

These examples can be found in a UserExtensions subfolder of the Simio example models - typically under a Public or All Users folder.

05 Custom Extensions Overview

How to Create and Deploy a User Extension

The general recommended steps to create and deploy a user extension are as follows:

1. Create a new .NET project in Visual Studio, or add an item to an existing project, using one of the Simio Visual Studio templates. Note: in addition to the commercial versions of Visual Studio, Microsoft also offers Express and Community editions which are available as free downloads from http://www.microsoft.com/express/Windows
2. Complete the implementation of the user extension and then build the .NET assembly (.dll) file
3. To deploy the extension, copy the .dll file into the [My Documents]\SimioUserExtensions directory. If the folder does not already exist you must create it. As an alternative, you may also copy it to [Simio Installation Directory]\UserExtensions, but ensure that you have proper permission to copy files here

Using a Deployed User Extension in Simio

A correctly deployed extension will automatically appear at the appropriate location in Simio. In some cases these are clearly identified as user add-ins, for example:

- In a model's Processes Window, all user defined steps will be available from the left hand steps panel under User Defined
- In a model's Definitions Window, when defining elements, all user defined elements will be available via the User Defined button in the Elements tab of the ribbon interface

In other cases it appears as though Simio has new features, for example:

- User defined selection rules are available for use in a model as dynamic selection rules
- Application add-ins are available for use in a project via the Select Add-In button in the Project Home tab of the ribbon
- The Table Imports and Binding add-ins are displayed on the Table ribbon under the Bind To button after at least one table has been added

You can find additional information on this topic in the following pages or by searching the main Simio help for "extensions" or "API". These topics provide a general overview and introduction to the features. More detailed information is available in the help file Simio API Reference Guide.chm that can be found in the Simio folder under Program Files. This provides over 500 pages of very detailed technical information. Although as of this writing there is no training course provided for creating Simio user extensions, an appendix in the Learning Simio slide set does provide additional step by step instructions. If you are a Simio Insider (which we strongly encourage), you can find additional examples and discussions in the forum topics Shared Items and API.

06 Back-End .dll Overview

Overview
The Simio execution engine can be accessed programmatic-ally without running the Simio GUI (graphical user-interface). A .NET assembly is provided (SimioDLL.dll) that can be directly referenced by 3rd-party software. Note that this assembly is built to be architecture- neutral; if the calling .exe is 32-bit, then the assembly will be loaded as a 32-bit assembly, and if the calling .exe is 64-bit, then the assembly will be loaded as 64-bit.

Loading a Project
SimioProjectFactory is used to load a Simio project file into memory. SimioProjectFactory is a static class in the Simio.dll assembly. It exposes the LoadProject method that takes a string containing the path to the file to load, and returns a reference to the resulting ISimioProject. Using the ISimioProject interface you can then manipulate the loaded project in various ways. Note that the API interfaces and classes available under ISimioProject are the same as those available to add-in writers (i.e. those creating user-written steps, elements, rules, and design and experimentation add- ins).

Collections
Simio's API defines many interfaces, several of which are collections of other interfaces. For example, ISimioProject has a Models property, of type IModels, which is a collection of IModel interfaces, each one corresponding to a model in the project. Similarly, the IProperties interface is a collection of IProperty interfaces. In general, an ISomethings interface (plural) is a collection of ISomething interfaces (singular). All Simio collections expose an integer Count property, returning the number of members in the collection, and all collections can be indexed by a zero-based integer index value. Indexing by integer i returns the ith ISomething in the ISomethings collection, or throws an IndexOutOfRange exception if i is outside the range of valid index values. In addition, if the collection contains named members, then the collection can also be "indexed" by using a string containing the name of the desired member. Indexing ISomethings by string s returns the ISomething with name s, if one exists, or returns null if an ISomething with the specified name is not present in the collection. Finally, Simio API collections support enumeration (using C#'s foreach or VB.NET's For Each), providing a convenient way of manipulating all members of a collection.

IModel
ISimioProject exposes Models, which is an IModels collection of IModel references. This IModels collection, like most other collections in the Simio API, can be indexed by integer or string, or enumerated using foreach. Both indexing and enumeration return IModel members.

The IModel interface currently provides programmatic access to certain top-level objects in the model. This includes the properties, states, and events defined on the model, as well as the model's Facility, which contains the facility objects that make up the logic of the model.

IModel also exposes properties for accessing the tables, function tables, and rate tables defined on the model.

IModel.Tables is a collection of ITable, which lets you set values in the model's table data. Use the ITable.Columns collection to determine the table's data schema, indexing by integer or string to return ITableColumn members. Index into the ITable.Rows collection to access the rows in the table, and then index into each IRow by integer or ITableColumn.Nameto get or set the string value of a single data item in the table (currently all table values are exposed as strings in the API, and interpreted internally according to the data type of the column).

Facility Objects
IModel.Facility exposes IntelligentObjects, which is an IIntelligentObjects collection of IIntelligentObject, and provides access to the top-level object instances in the model. Each of these object instances exposes its ObjectName and its Properties collection (of type IProperties), which can be indexed by integer or string to get or set the values of the Simio properties for the object instance. For example, using IIntelligentObject and its Properties collection of IProperty, you can set the "Processing Time" property of a "Server" object instance.

IExperiment
IModel exposes Experiments, an IExperiments collection of IExperiment references representing the experiments defined on this model. IExperiments is also indexed by integer or string, or can be enumerated using foreach, all of which return members of type IExperiment.

The IExperiment interface is used to manipulate and run experiments on the model. Use IExperiment. RunSetup to get or set the starting time, ending time, and warm-up period to be used for the experiment. Use IExperiment's Controls, Responses, and Constraints properties to return IExperimentControl, IExperimentResponse, and IExperimentConstraint interfaces, respectively. These provide access to the definitional part of controls, responses, and constraints.

Use the IExperimentControls and IExperimentControl interfaces to determine the names and types (integer or real) of the controls defined on the experiment. Use the IExperimentResponses and IExperimentResponse interfaces to specify the various settings on each response defined on the experiment. For example, you can get or set the expression used to evaluate the response, as well as the optional upper and/or lower acceptable bounds for the response.

IExperiment.Scenarios is a collection of IScenario, representing the various scenarios defined on the experiment. This collection can be modified via the API by using Clear to remove any existing IScenario entries, and CreateScenario to return a reference to a new scenario.

The IScenario interface is used to manipulate a single scenario (i.e. a single row in an experiment in the Simio desktop application). Use ReplicationsRequired to indicate how many replications of this scenario should be run. Use SetControlValue to provide the value for each experiment-specified control. Call this method with an IExperimentControl interface (supplied by the experiment's Controls collection) and the desired value for the control. After running the experiment, use GetResponseValue to retrieve the average value of each response across all replications, or use GetResponseValueForReplication to retrieve the value of a response for a specific replication. Both of these methods take as input an IExperimentResponse interface (supplied by the experiment's Responses collection) to indicate which response value is being requested.

Running an Experiment
The IExperiment interface provides methods for running the experiment, as well as events to subscribe to for monitoring the state of the run. If the calling program is an interactive application, use the RunAsync method to run an experiment on a background thread. This method returns to the caller when the experiment starts running, allowing the caller to remain responsive to the interactive user.

Alternatively, the Run method can be called, but it is a synchronous (blocking) call, which does not return to the caller until the experiment is done. This is often an appropriate choice for non- interactive uses.
During an interactive run, you may call IExperiment. RunAsyncCancel to cancel running the experiment. Between runs, call IExperiment.Reset to delete any existing results produced by a previous run.

IExperiment.IsBusy returns true if the experiment is currently running.

There are several events defined on IExperiment that can be used to track the progress of the experiment run, and to retrieve the statistics at the end of the run. RunStarted is raised once after the experiment run is initialized but before running actually begins. ScenarioStarted is raised as each scenario is about to begin running its replications. ReplicationStarted is raised once for each replication as it is about to start running.

Similiarly, ReplicationEnded is raised to indicate that a single replication has completed, or has terminated with an error condition. The ReplicationEndedEventArgs can be examined to determine which replication of which scenario just ended, as well as any error text. ScenarioEnded is raised when all replications for a given scenario have completed running. The ScenarioEndedEventArgs provides the model-defined statistical results collected during the run. Its IScenarioResults property is a collection of IScenarioResult items, each one providing the value of a single statistic collected during the run. These statistics are the same values presented in the Simio desktop application's Experiment Results pivot table. Finally, ExperimentCompleted event is raised at the end of the run.

Code Snippets

```
using SimioAPI; namespace UsageSamples
{
public class SnippetClass1
{
//
// Load a Simio project file
//
void LoadSimioProject()
{
string fileName = "test.spfx";

ISimioProject project; try
{
project = SimioProjectFactory.LoadProject(fileName);
}

catch (Exception ex)
{
MessageBox.Show(this, ex.Message, "Load failure");
}
}

//
// Examples of setting various values in a model.
//
void UpdateSomeValues(IModel model)
{
// Retrieve a specific instance of Server by name.
IIntelligentObject myServer = model.Facility.
IntelligentObjects["MyServer"];
// Retrieve a specific property of this Server by name.
IProperty myServerProcessingTime = myServer.
Properties["ProcessingTime"];
// Set the property value. All property values are passed in as
strings. myServerProcessingTime.Value = "12.34";
// Set the unit. Since we know that ProcessingTime is a time,
// we can convert Unit directly to ITimeUnit and set the value.
(myServerProcessingTime.Unit as ITimeUnit).Time = TimeUnit.
Minutes;

// A pedantic way of setting the first rate of the model's first
rate table. IRateTable rateTable = model.RateTables[0];
IRateTableInterval interval = rateTable.Intervals[0]; interval.Rate
= 5.5;
// And a more concise way of doing it. model.RateTables[0].
Intervals[0].Rate = 5.5;

// Set a data value in some row of a table. string tableName =
"MyTable";
string propName = "MyProperty"; int rowIndex = 4;
string newVal = "111.111"; model.Tables[tableName].
Rows[rowIndex].Properties[propName].Value = newVal;
}

//
// Retrieve the names of all tables and table properties in this
model.
//
```

```
void TableExample(IModel model)
{
foreach (ITable table in model.Tables)
{
// Do something with the table name string string1 = table.
Name;
// ...

// Do something with each column name
foreach (ITableColumn column in table.Columns)
{
string string2 = column.DisplayName + " (" + column.Name +
")";
// ...
}
}
}
}

public class SnippetClass2
{
//
// Run an experiment asynchronously. This is a typical pattern
// for running Simio experiments in an interactive application.
//
public void StartRunningExperiment(IExperiment experiment)
{
if (experiment.IsBusy)
return; // It is already running!

// Clear out any existing replication data. experiment.Reset();

// Specify run times.
IRunSetup setup = experiment.RunSetup; setup.StartingTime
= new DateTime(2010, 10, 01); setup.WarmupPeriod =
TimeSpan.FromHours(8.0);
setup.EndingTime = setup.StartingTime + TimeSpan.
FromDays(100.0);

// Let's say that we require at least 5 replications for each
scenario. foreach (IScenario scenario in experiment.Scenarios)
if (scenario.ReplicationsRequired < 5) scenario.
ReplicationsRequired = 5;

// Ready to run. Wire up to the various run events.
// In this snippet we only care about ScenarioEnded and
// RunCompleted, but we could subscribe to other events, too.
experiment.ScenarioEnded += experiment_ScenarioEnded;
experiment.RunCompleted += experiment_RunCompleted;

// Now start the run. RunAsync will return to us after the
// experiment is initialized and is about to start running.
experiment.RunAsync();
}

void experiment_ScenarioEnded(object sender,
ScenarioEndedEventArgs e)

{
// This event handler will be called when all replications for a
// given scenario have completed.  At this point the statistics
// produced by this scenario should be available. IExperiment
experiment = sender as IExperiment;

// Do something with the statistics produced by the various
// modeling constructs (i.e. Sources, Servers, Sinks, etc.)
foreach (IScenarioResult result in e.Results)
{
// Each result contains the name, category, average value,
// min and max value, half-width and standard deviation
// across all replications that were run for this scenario.

// This is where you would record these observations.
// ...
}

// Also retrieve this scenario's average values for any responses
// defined on the experiment.
foreach (IExperimentResponse response in experiment.
Responses)
{
double responseValue = 0.0;
if (e.Scenario.GetResponseValue(response, ref responseValue))
{
// We got a valid value for this response.
// Do something with it here.
// ...
}
else
{
// Didn't get a value.
}
}

}
void experiment_RunCompleted(object sender,
RunCompletedEventArgs e)
{
// This event handler is the last one to be called during the run.
// When running async, this is the correct place to shut things
down. IExperiment experiment = sender as IExperiment;

// Un-wire from the run events when we're done. experiment.
ScenarioEnded -= experiment_ScenarioEnded; experiment.
RunCompleted -= experiment_RunCompleted;
}
}
}
```

10 Interacting with API During Runtime

To interact with the API during runtime, you can use the schedule and fire events.

Read more here:
https://www.simio.com/simio-forum/topic/1229-schedule-and-fire-eventschoose-route/

ScheduleAndFireEvents_DLL.zip
Download Here

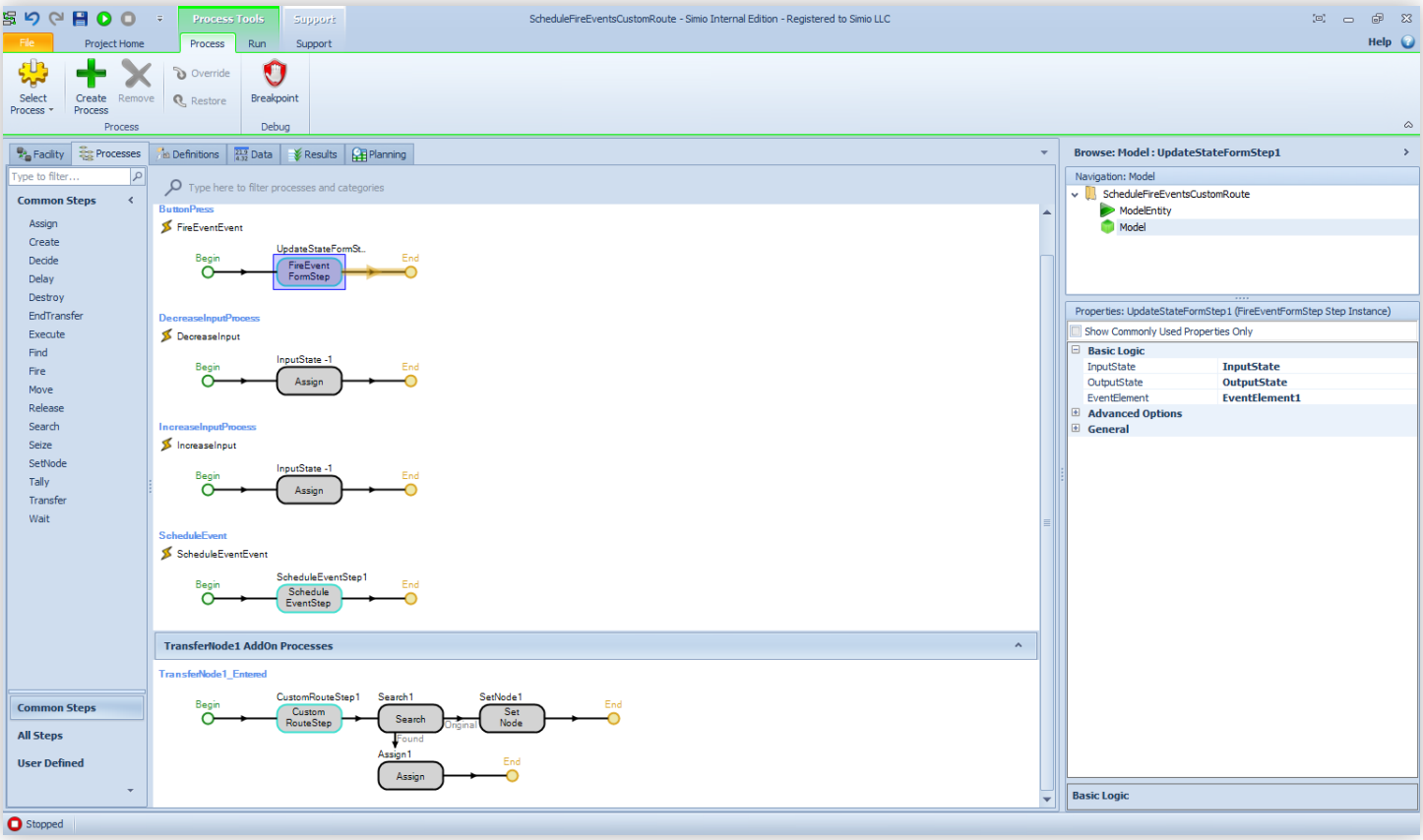
This file contains the dll needed for the custom steps and elements. This dll needs to be placed in the C:\Users\<USER NAME>\Documents\SimioUserExtensions folder. Make sure you UnBlock the dll (right-click and select properties) before trying to use it.

ScheduleAndFireEvents.spfx
Download Here

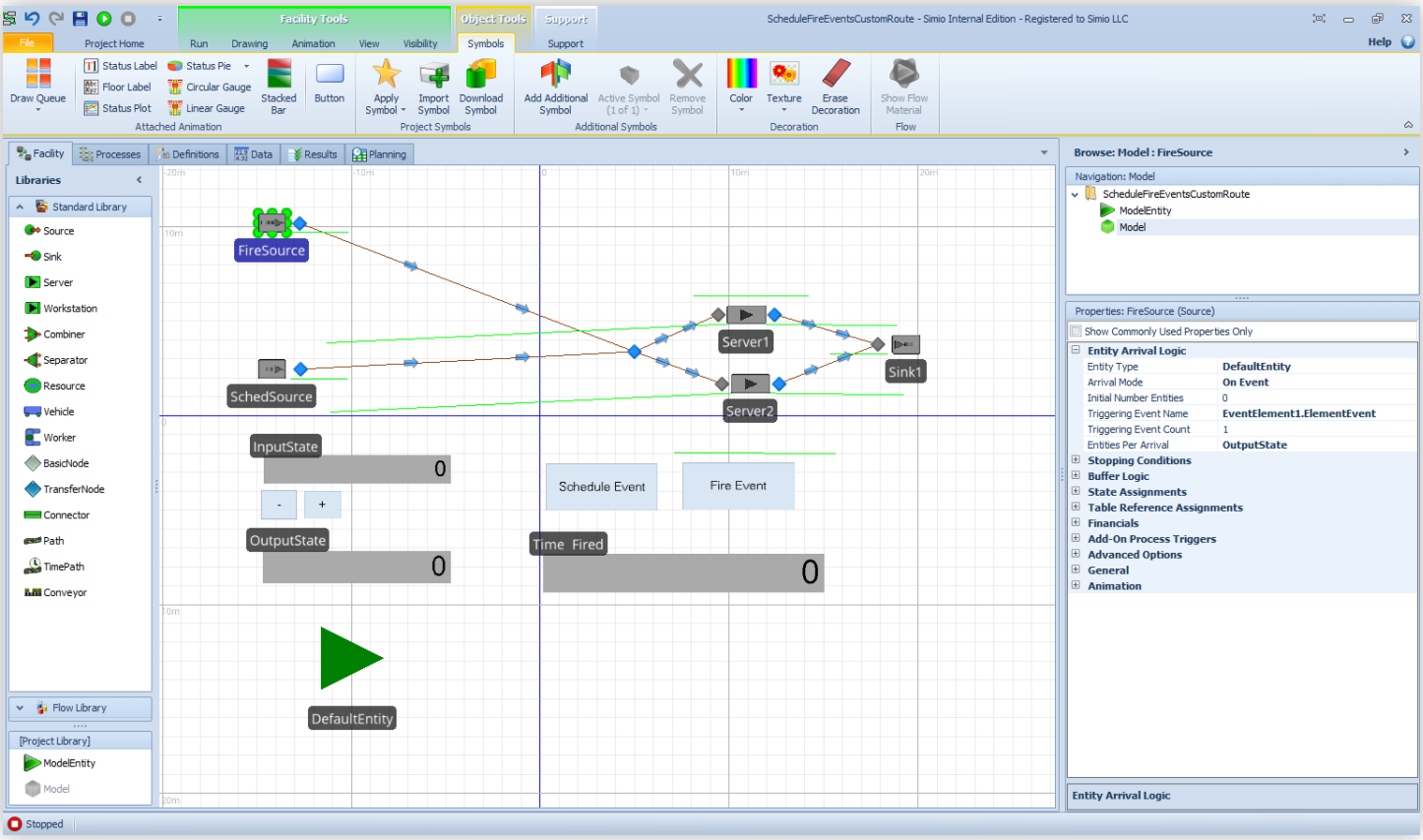
This is a sample model that shows how to use the custom steps and elements built in the dll. Only 1 and 2 are needed to get the example running.

ScheduleAndFireEvents_Code.zip
Download Here

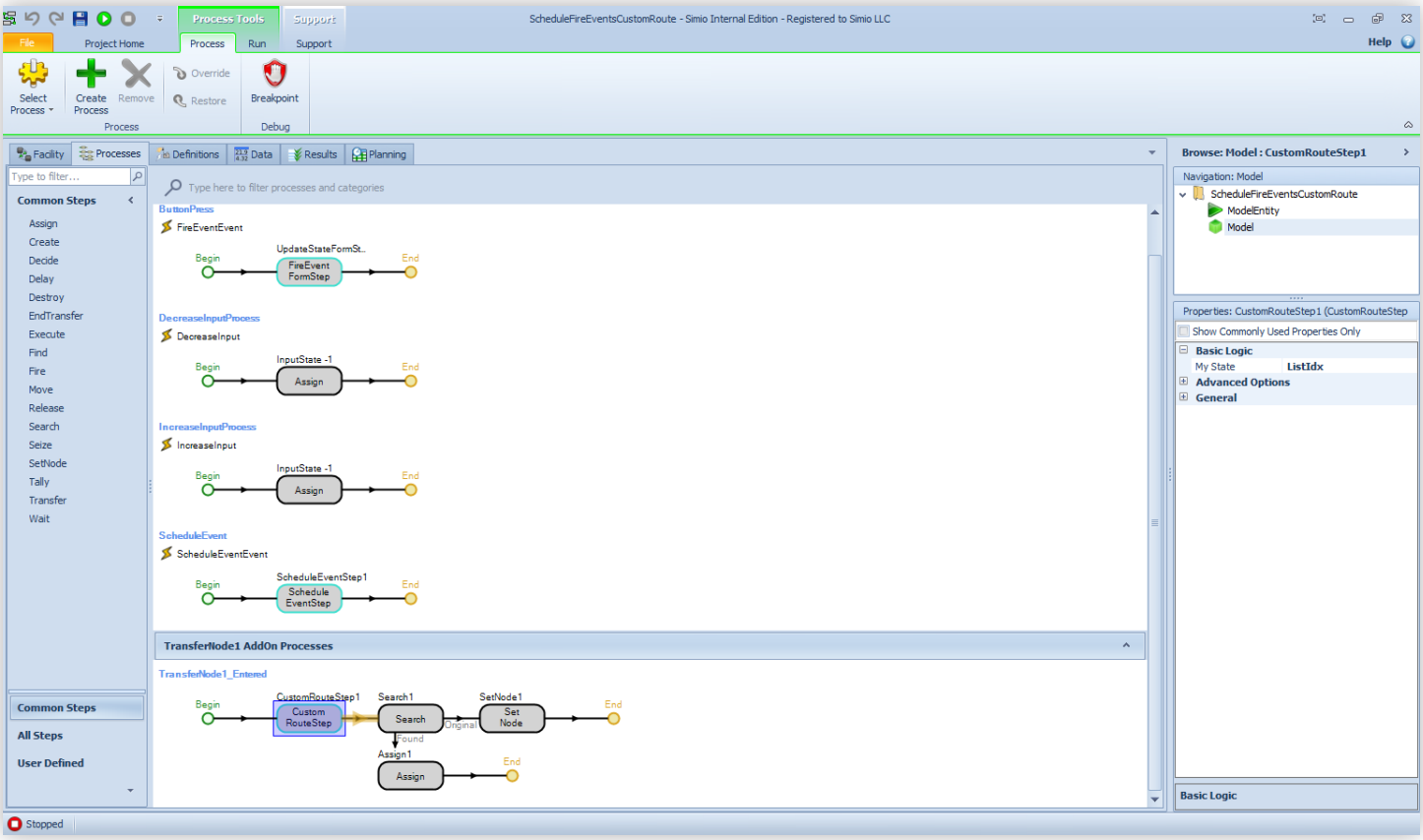
This file contains the code that was used to create the dll. The code was developed based on the Visual Studio templates that come with Simio.



The above image shows how to use a custom step to fire events in C#. The events are attached to an element. Once fired, the events will be sent back to the Simio model during runtime. The step also returns the number of entities to create in the OutputState.



The above image shows the event from C# being received in the model. The event is being received by the Source. The source will be triggered to create entities. The number of entities to create is provided by the OutputState.



The shows a step being called during runtime. The C# code will load a form and enable the user to pick a route for the entity. Based on what is select, either 1 or 2 is returned into the ListIdx state. Then, logic in the model will route the entity based on the return value.

14 Import Objects & Links from a Spreadsheet

This Simio add-in will import a model (objects and links) from a spreadsheet.

Read more here:
https://www.simio.com/simio-forum/topic/763-import-objects-from-spreadsheet/

ImportObjectsAndLinks FromSpreadsheet_Code.zip
Download Here
This file contains the dll needed. This dll needs to be placed in the C:\Users\<USER NAME>\Documents\SimioUserExtensions folder. Make sure you UnBlock the dll (right-click and select properties) before trying to use it.

ImportObjectsLinksVertices.xlsx
Download Here
This file contains the Excel spreadsheet you edit to import the objects and links to populate your model. Note: you must have Microsoft Excel on your computer.

ImportObjectsAndLinks FromSpreadsheetAddIn_DLL.zip
Download Here
This file contains the code that was used to create the dll. The code was developed based on the Visual Studio templates that come with Simio.

15 Export Import Model Using a Text File

This Simio add-in will export your model to a text file (pipe delimited). You will be able to modify the file and import it back into Simio.

Read more here:
https://www.simio.com/simio-forum/topic/1237-export-import-model-using-a-text-file/

ExportImportModelAddIn_DLL.zip
Download Here
This file contains the dll needed. This dll needs to be placed in the C:\Users\<USER NAME>\Documents\SimioUserExtensions folder. Make sure you UnBlock the dll (right-click and select properties) before trying to use it.

ExportImportModel_Code.zip
Download Here
This file contains the code that was used to create the dll. The code was developed based on the Visual Studio templates that come with Simio.

Once you placed the dll file, open Simio and load your model. From the Project Home, select **Add-In button** and select **Export Import Model**. Choose the function that you want to run.

Note: If you export and then import the model without a delete, it will update the object data if the object already exists.

16 Running Experiments from the API

This Simio add-in will place data into a local directory when an experiment is run through the API similar to when experiments are run from inside Simio.

Read more here:
https://www.simio.com/simio-forum/topic/699-running-experiments-from-the-api/

RunExperiments.zip
Download Here
This file contains the dll needed.

This dll needs to be placed in the C:\Program Files (x86)\Simio\UserExtensions folder. Make sure you UnBlock the dll (right-click and select properties) before trying to use it.

RunExperiments_TestModel.spfx
Download Here
This is a sample model that shows how RunExperiments dll.

17 Additional Resources

Simio Insiders

Simio Insiders are individuals granted early access to designs and software on the cutting edge of simulation technology. Each person is encouraged to add ideas so they can have immediate impact on product development. It is also a place where you can share ideas and problems with other Simio users. There is a special section in the forum dedicated to Simio API questions and solutions.

Learn more here:
https://www.simio.com/insiders.php

Visual Studio Templates

If you install Visual Studio templates, you can get started creating visual studio projects that can be used with Simio.

Once installed, there are 3 types of project that you can create:

- Design Time Add-In - Application Add-In
- Custom Rule - User Defined Selection Rule
- Run Time Steps User Defined Step and Element

Learn more here:
https://www.visualstudio.com

504 Beaver Street
Sewickley, PA 15143
Telephone: 412-528-1576
Fax: 412-253-9378
Email: info@simio.com
Web: www.simio.com