Using the Simio back-end .dll

Overview

With the release of Simio Sprint 45, the Simio execution engine can be accessed programmatically *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 I*Somethings* interface (plural) is a collection of I*Something* 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 i^{th} I*Something* in the I*Somethings* 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 I*Somethings* by string *s* returns the I*Something* with name *s*, if one exists, or returns null if an I*Something* 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;
        }
    }
}
```