SIMPROCESS Expression Demo Models

The ExpressionDemos directory includes a number of SIMPROCESS models that demonstrate one or more of the SIMPROCESS Expression Language features.

- **ActivateGenerate.spm**
- **ActivityReference.spm**
- **ConnectorDelay.spm**
- **CreateArray.spm**
- **DatabaseDemoExp.spm**
- **ExpressionPlots.spm**
- **ExternalCall.spm**
- **FileDemo.spm**
- **GenerateEntity.spm**
- **GetResourceandFreeResource.spm**
- **ReleaseEntity.spm**
- **SpreadsheetDemoExp.spm**
- **ConfigureWithDatabase.spm**
- **InterruptDemo.spm**

Appendix F of the *SIMPROCESS User's Manual Appendices* contains a complete listing of all SIMPROCESS System Methods along with an examples section. Also, Chapter 2 of Part B of the *SIMPROCESS User's Manual* contains discussions of various System Methods. The models described here are simple models designed to show how to use the following SIMPROCESS System Methods.

- **`ActivateGenerate`**
- **`AddPlotLegend`**
- **`Attribute`**
- **`ClearMap`**
- **`CloseDatabase`**
- **`CloseFile`**
- **`CloseSpreadsheet`**
- **`CreateArray`**
- **`CreateAttribute`**
- **`CreateEntityType`**
- **`CreateMap`**
- **`CreatePlot`**
- **`CreateResource`**
- **`DateTime`**
- **`DisplayPlot`**
- **`ElapsedTime`**
- **`ExternalCall`**
- **`FireTrigger`**

- **FreeResource**
- **Gate**
- **GenerateEntity**
- **GetArrayValue**
- **GetEntity**
- **GetFromMap**
- **GetNext**
- **GetResource**
- **GetResult**
- **InterruptActivity**
- **InterruptEntity**
- **OpenDatabase**
- **OpenFile**
- **OpenSpreadsheet**
- **PlotValue**
- **PutInMap**
- **ReadFromDatabase**
- **ReadFromFile**
- **ReadFromSpreadsheet**
- **ReleaseEntity**
- **SetActivityExpression**
- **SetArrayValue**
- **SetResourceCost**
- **SetResourceDowntime**
- **SetResourceExpression**
- **WriteToDatabase**
- **WriteToFile**
- **WriteToSpreadsheet**

The following commands are used but not specifically discussed.

- **DrawIntegerSample**
- **DrawRealSample**
- **FLOAT**
- **GetEntityStatistic**
- **GetResourceStatistic**
- **HALT**
- **INTTOSTR**
- **OUTPUT**
- **REALTOSTR**
- **SimTime**
- **SUBSTR**

ActivateGenerate.spm

This model demonstrates the **`ActivateGenerate`** System Method, which causes the Generate Activity specified in the command to begin generation. **`ActivateGenerate`** is used when the time needed for Entity generation to start is based on a condition in the model.

The model generates 10 **Entity20** Entities. Once the final **Entity20** has reached **Dispose Entity20**, **Generate Entity21** begins and generates 10 **Entity21** Entities. For a Generate Activity to not begin generation at the beginning of a simulation, and for a Generate Activity to wait for the **`ActivateGenerate`** command to begin generation, the **Start** for the Generate Activity must be greater than or equal to the **End** of the simulation (**Simulate/Run Settings**). Open the properties for **Generate Entity21** and look at the **Start/End** tab to see this. The **AcceptEntity** Expression of **Dispose Entity20** contains the **`ActivateGenerate`** command that starts **Generate Entity21**.

Note that since the parameter of **`ActivateGenerate`** is the name of a Generate Activity, the name entered must be unique within the model.

ActivityReference.spm

This model demonstrates four methods of referencing Activities. The model represents a manufacturing operation that has two manufacturing lines: **Line1** and **Line2** (Processes). When a part arrives it must be routed to the manufacturing line that has the least number of parts in process and waiting to process. In order to do this, those values from each manufacturing line must be known. Activities have a System Attribute called **`NumberIn`**. This Attribute contains the number of Entities at an Activity. It is the number of Entities in process plus the number of Entities waiting to process. **Which Line?** is a Branch Activity in the **Manufacturing Lines** Process. This Branch Activity uses **`NumberIn`** from **Line 1 Delay** (inside **Line1** Process) and **`NumberIn`** from **Line 2 Delay** (inside **Line2** Process) to determine which manufacturing line is least busy. However, **Line 1 Delay** and **Line 2 Delay** are in separate Processes from **Which Line?** and from each other.

There are four ways for **Which Line?** to get a reference to **Line 1 Delay** and **Line 2 Delay**. The four alternatives (**Array**, **Attribute**, **Sibling and Child**, and **Map**) of the **Manufacturing Line** Process demonstrate each of these ways.

Array Alternative

This alternative uses an array to hold the Activities **Line 1 Delay** and **Line 2 Delay**. (See **CreateArray.spm** and Appendix F of the *SIMPROCESS User's Manual Appendices* for discussions on creating and using arrays.) In the **Start Run** Expression of the **Model Expressions** (**Define/Model Expressions**), an array of type **`ANYOBJ`** is created and assigned to the **Model.ActArray** Attribute. The array is a one dimensional array with a length of three for the dimension. Since indices for arrays are zero based, the length was set to three so 1 and 2 could be used to reference each manufacturing line. Thus, the zero position is not used. **Line 1**

Delay and **Line 2 Delay** are added to the array in the **Start Simulation** Expression of each Activity.

In **Line 1 Delay**: `SetArrayValue(Model.ActArray, 1, Self);`
In **Line 2 Delay**: `SetArrayValue(Model.ActArray, 2, Self);`

The System Attribute **`Self`** is used for the reference to the Activity. The **Accept Entity** Expression of **Which Line?** retrieves a reference to each Activity, then compares the **`NumberIn`** values to determine how to route the incoming Entity.

Attribute Alternative

This alternative uses Model Attributes with a mode of **`Object`** to hold the **Line 1 Delay** and **Line 2 Delay** Activities. In the **Start Simulation** Expression of each Activity the Activity is assigned to either **Model.Line1** or **Model.Line2**.

In **Line 1 Delay**: `Model.Line1 := Self;`
In **Line 2 Delay**: `Model.Line2 := Self;`

These Attributes are used in the **Accept Entity** Expression of **Which Line?** to determine the routing of the incoming Entity. This option is most useful when only a few Activities need to be referenced since an Attribute must be created for each Activity.

Sibling and Child Alternative

This alternative requires no Model Attributes, and no Expression Language code is required in **Line 1 Delay** or **Line 2 Delay**. In the **Accept Entity** Expression of **Which Line?** the **`Sibling`** and **`Child`** System Attributes are used to get references to **Line 1 Delay** and **Line 2 Delay**. (There is also a **`Parent`** System Attribute, but it is not used in this example.) **`Sibling`** and **`Child`** each require a **`STRING`** parameter that is the name of another Activity. **`Sibling`** returns a reference to an Activity or Process at the same hierarchical level as the Activity. **`Child`** is only a System Attribute of a Process since only Processes can have children. Thus, **`Child`** returns a reference to an Activity or Process that is inside of a Process. In the Expression, the sibling Processes of **Which Line?**, **Line1** and **Line2**, are retrieved first. These Processes are assigned to local variables of type **`ANYOBJ`**.

```
line1Process := Sibling("Line1");
line2Process := Sibling("Line2");
```

Since **`line1Process`** and **`line2Process`** reference Processes, the **`Child`** System Attribute is used to get the reference to the Delay Activities inside of each.

```
line1Delay := line1Process.Child("Line 1 Delay");
line2Delay := line2Process.Child("Line 2 Delay");
```

(See the Expression itself for an alternative coding.) The **NumberIn** System Attribute of each is then used to determine the routing of the incoming Entity.

Since no Attributes need to be defined, this method is useful if the needed Activities are relatively close to the Activity requiring the Activity references. However, the chain of Activity/Process references could be quite long if that is not the case.

Map Alternative

This alternative demonstrates the use of **CreateMap**, **PutInMap**, **GetFromMap**, and **ClearMap**. (See Appendix F of the *SIMPROCESS User's Manual Appendices* for a full discussion on using Maps in SIMPROCESS.) In the **Start Run** Expression of the **Model Expressions**, the Map is created and assigned to the Attribute **Model.ActMap**. **Line 1 Delay** and **Line 2 Delay** are added to the Map in the **Start Simulation** Expression of each Activity. The **Name** of the Activity is the key for the value (**Self**).

```
PutInMap(Model.ActMap, Name, Self);
```

The **Accept Entity** Expression of **Which Line?** uses **GetFromMap** to retrieve a reference to each Activity (**line1** and **line2** are local variables of type **ANYOBJ**).

```
line1 := GetFromMap(Model.ActMap, "Line 1 Delay");
line2 := GetFromMap(Model.ActMap, "Line 2 Delay");
```

As with the other alternatives, the **NumberIn** System Attribute is then used to determine the routing of the incoming Entity.

When using Maps it is good practice to clear the Maps at the end of a simulation, which frees up memory. Thus, **ClearMap** is used in the **End Run** Expression of the **Model Expressions**. The parameter is the Model Attribute that has the reference to the Map (**Model.ActMap**).

ConnectorDelay.spm

This model demonstrates the use of arrays and Connector delays. The model is very similar to **CreateArray.spm** because three origins and two destinations are modeled. A two dimensional array is used in both models to hold information about each origin/destination combination. (See **CreateArray.spm** and Appendix F of the *SIMPROCESS User's Manual Appendices* for discussions on creating and using arrays.) In this model, the array contains the distances from each origin (Seattle, Denver, and Los Angeles) to each destination (New York and Atlanta). The array is created in the **Start Run** Expression of the **Model Expressions** and populated in the **Start Simulation** Expression of the **Model Expressions**.

The **Accept Entity** Expression of **Select Destination** uses random numbers to determine the origin/destination values. Then the distance is retrieved from the array and placed in the Attribute **Entity.Distance**. This Attribute, along with **Model.AvgMilesPerHour**, is used to

determine the travel time on the Connectors from **Select Destination** to **Arrive New York** and from **Select Destination** to **Arrive Atlanta**.

CreateArray.spm

This model demonstrates the **CreateArray**, **GetArrayValue**, and **SetArrayValue** System Methods. Also, the **DrawIntegerSample** and **DrawRealSample** System Methods are used. **CreateArray** is used to create a multi-dimensional array. The type of array can be **REAL**, **INTEGER**, **BOOLEAN**, **STRING**, or **ANYOBJ**. **GetArrayValue** retrieves a value from an array, and **SetArrayValue** places a value into an array.

The model creates a two-dimensional array for **REAL** numbers. The length of the first dimension is 3, and the length of the second dimension is 2. The first dimension represents three origin locations, and the second dimension represents two destination locations. So the array contains the travel time from each origin to each destination.

View the **Start Run** Expression of the **Model Expressions** to see the array created. Any array created must be returned to an Attribute of type **Object** (in this case **TimeArray**). Since the array is created in the **Start Run** Expression, the **TimeArray** Attribute must have **Do Not Reset Before Each Replication** selected on its properties dialog. The first parameter of **CreateArray** is the type of array (**"REAL"**, **"INTEGER"**, **"BOOLEAN"**, **"STRING"**, or **"ANYOBJ"**). Note that the type must be entered as a STRING. The number of dimensions of the array is determined by the number of parameters that follow the type. In this example, the array is two-dimensional since two **INTEGER** values (a 3 and a 2) follow the type. Thus, the parameters represent the length of each dimension.

The array is populated in the **Start Simulation** Expression of the **Model Expressions**. The first parameter of **SetArrayValue** must be the Attribute that references the array. The next parameters must be the indices of the particular location in the array that will hold the value. The indices must be **INTEGER**, and the number of indices must match the array dimensions. Also, indices are zero based. Thus, if a particular dimension has a length of 2, the allowable indices are 0 and 1. The final parameter is the value to place in the array. The type (**REAL**, **INTEGER**, etc.) of the value must match the type of the array.

The **Accept Entity** Expression of **Travel from Origin to Destination** shows **GetArrayValue**. As with **SetArrayValue**, the first parameter must be the Attribute that references the array. The remaining parameters are the indices of the value to retrieve. The indices must be **INTEGER**, and the number of indices must match the array dimensions.

DatabaseDemoExp.spm

This model demonstrates the **OpenDatabase**, **CloseDatabase**, **ReadFromDatabase**, **GetNext**, and **GetResult** System Methods. Also, the **GetResourceStatistic**, **INTTOSTR**, **REALTOSTR**, and **FLOAT** System Methods are used. This demonstration will only run on Windows since the included sample database is an Access database.

The model reads Resource levels from a database table. There are two Resources, **SalesRep** and **ServiceRep**. The database table **ResourceInfo** contains Resource levels for each Resource for three locations (Chicago, Atlanta, and Phoenix). The desired location for the model run is entered at the beginning of the simulation. The model then queries the database for the Resource levels at the entered location. At the end of 10 replications, the average utilization across replications for each Resource is placed into the **Results** table.

`OpenDatabase` is used in the **Start Run** Expression of the **Model Expressions**. An Attribute of type **`Object`** must be defined to hold a reference to the database connection (in this case **Database**). Since the Attribute **Database** is used in the **Start Run** Expression, the **Do Not Reset Before Each Replication** option must be selected on the Attribute properties dialog. The parameter for the **`OpenDatabase`** command is the string **`"example.properties"`** which refers to the file by that name in the model's directory. This file tells SIMPROCESS how to connect to the database (**`ExampleDB.mdb`**). (See "Interfacing With a Database" in Chapter 2 of Part B of the *SIMPROCESS User's Manual* for more information on using a properties file.) The **Start Run** Expression also reads in the Resource levels using **`ReadFromDatabase`**. The first parameter is the Attribute that contains the database connection reference (**Model.Database**). The second parameter is a **`STRING`** that will be used to identify the ResultSet that is returned from the query. The final parameter is a **`STRING`** that contains the SQL query. The ResultSet that is returned consists of rows and columns. The number of rows in the ResultSet is determined by the number of fields requested in the query. The number of columns is the number of records returned. **`GetNext`** is used to move through the ResultSet. When a ResultSet is returned from a query, the row pointer is pointing to before the first row. **`GetNext`** moves the pointer to the next row. It returns **`TRUE`** if there is a next row in the ResultSet, **`FALSE`** if not. **`GetResult`** returns a value from the ResultSet. The first parameter is the name of the ResultSet assigned in the **`ReadFromDatabase`** command. The second parameter is the name of the field value to return. The type (**`REAL`**, **`INTEGER`**, etc.) of the Attribute or local variable receiving the value must match the type of the value returned.

The **`WriteToDatabase`** command is used in the **End Run** Expression of the **Model Expressions**. The first parameter is the Attribute that contains the database connection reference. The second parameter is a **`STRING`** that contains the SQL statement. The bulk of the **End Run** Expression is commented out. See **Report/Export Results/Database** for the SQL statements that are being executed after the simulation completes. (Also, see "Exporting Results to a Database" in Chapter 4 of Part B of the *SIMPROCESS User's Manual*.) **`CloseDatabase`** is used in the **End Run** Expression to close the database connection. The only parameter is the Attribute that contains the database connection reference.

`GetResourceStatistic` is used in the commented out portion of the **End Run** Expression and in the **End Simulation** Expression of each Resource.

The model **`DatabaseDemo.spm`** uses the Input Sources and Export Results features to read from and write to databases. See Chapter 3 of Part A of the *SIMPROCESS User's Manual* for

information on using Input Sources and Chapter 4 of Part B of the *SIMPROCESS User's Manual* for information on exporting results to a database.

ExpressionPlots.spm

This model demonstrates the **`CreatePlot`**, **`AddPlotLegend`**, **`PlotValue`**, and **`DisplayPlot`** System Methods. The **`GetEntityStatistic`** System method is also used. A trace plot and a histogram plot are created using Expressions. The model runs for 10 replications. Values from each replication are plotted. Thus, this model demonstrates plotting across replications.

The **Start Run** Expression of the **Model Expressions** shows the creation of the plots using **`CreatePlot`**. Attributes of type **`Object`** (**ProcessTimeTrace** and **ProcessTimeHistogram** for this model) are required for **`CreatePlot`**. These Attributes hold the references to the created plots. Since the plots are created in the **Start Run** Expression, **Do Not Reset Before Each Replication** must be selected on the Attribute properties dialog. The first parameter of **`CreatePlot`** is a **`STRING`** which designates the type of plot ("Trace" or "Histogram"). The second parameter is the plot title, which is also a **`STRING`**. The X Axis label and the Y Axis label (third and fourth parameters) are optional. However, there must be an X Axis label (even if it is just " ") if there is a Y Axis label. Note that **`CreatePlot`** does not cause the plot to appear. Thus, a plot can be created and populated before being made visible.

`AddPlotLegend` is also demonstrated in the **Start Run** Expression. The use of **`AddPlotLegend`** is optional. Plots can be created and displayed without legends. The first parameter of **`AddPlotLegend`** is the Attribute that contains the plot reference. The second parameter is an **`INTEGER`** that represents the Dataset that the legend will identify. Datasets must be greater than or equal to 0. Normally, Datasets will number consecutively starting with 0. The text (**`STRING`**) of the label that will display in the legend is the next parameter. The final parameter is Color (**`STRING`**) and is optional. If Color is not specified, SIMPROCESS will automatically assign a color. The allowable colors are listed in the "SIMPROCESS Color Table" in Appendix F of the *SIMPROCESS User's Manual Appendices*.

The **End Simulation** Expression of the **Model Expressions** shows the **`PlotValue`** System Method. The average processing time of each entity type is retrieved using **`GetEntityStatistic`**. These values are then converted to Minutes (since the simulation clock is in Hours) and plotted. The first parameter of **`PlotValue`** is the Attribute that contains the reference to the plot. The **`INTEGER`** Dataset is the next parameter. The number of parameters following Dataset depends on the type of plot. Trace plots are expecting to receive an X and Y value. Histogram plots only require one value. The values to plot can be **`REAL`** or **`INTEGER`**.

The plots are displayed in the **End Run** Expression of the **Model Expressions**. The only parameter is the Attribute that holds the reference to the plot. Plots created through Expressions can also be displayed by using the menu (**Report/Display Real-Time Plots**) or the **Display Plot** button on the toolbar.

ExternalCall.spm

This model demonstrates the use of the **`ExternalCall`** System Method. **`ExternalCall`** is used to make calls to Java classes external to SIMPROCESS. There is a **`classes`** directory in this model's directory. In the **`classes`** directory are the files **`ArraySort.java`** and **`as.jar`**. Within **`as.jar`** is the **`com.demo.ArraySort`** class, which is the compiled version of **`ArraySort.java`**. External Java classes should be in a **`classes`** directory inside the model's directory where they can be packaged in a jar file or inside the appropriate package directory structure. External Java classes can also be placed in the **`ext`** directory, which is in the SIMPROCESS installation directory. However, this should only be done if multiple models require the same external Java classes.

The model is a very simple representation of task order fulfillment. Task Orders arrive approximately every 10 hours. There are multiple tasks that are required to fulfill the task order. The number of tasks required is set by a model parameter (**Model.TotalTasks**). There are 5 Worker Resources available to accomplish the tasks. Each task requires from 1 to 5 Workers. The task times and number of workers needed for each task are stored in a two dimensional array in the Entity Attribute Task (**Entity.Task**) when the Task Order is generated (see the Release Entity Expression of the Generate Activity). **`ArraySort.java`** is used to sort the array of tasks so the tasks are ordered according to the number of Workers required. Thus, the tasks with the lowest number of workers happen first, and the tasks with the highest number of workers happen last. The class has a method to sort a one dimensional array and a method to sort a two dimensional array.

`ArraySort.java` has the following methods that can be called from a model:

- **`ArraySort()`** – constructor for instantiating **`ArraySort`**
- **`sortArray(Object array)`** – returns a sorted one dimensional array. The array type must be int[] (INTEGER), double[] (REAL), or String[] (STRING).
- **`sort2DArray(Object array, Integer idx)`** – returns a sorted two dimensional array. The array type must be int[] (INTEGER), double[] (REAL), or String[] (STRING). The **`idx`** parameter determines the dimension to sort (must be 0 or 1).

`ExternalCall` is a Function System Method. That is, it always returns a value. Thus, the type (**`REAL`**, **`INTEGER`**, etc.) of the Attribute or local variable receiving the value from ExternalCall must match the return type of the Java method called. Also, the type of a parameter passed to an external method must match the type of the parameter specified in the external method. The table below shows the mapping of SIMPROCESS types to Java types.

SIMPROCESS Type	Java Type
INTEGER	int or Integer
REAL	double or Double
BOOLEAN	boolean or Boolean
STRING	String
ANYOBJ	Any Java Object

If the return type of the Java method is **`void`**, a value of **`TRUE`** is returned to SIMPROCESS. So any time **`ExternalCall`** is used with a Java method that has a **`void`** return type, a **`BOOLEAN`** Attribute or local variable must be used as the receiving variable.

The first parameter of an **`ExternalCall`** System Method is the name of the Java class entered as a **`STRING`**. The second parameter is the **`STRING`** name of the method within the class to call. Any other parameters required are determined by the method being invoked. If the method invoked has no parameters, then **`ExternalCall`** should have no parameters past the method name. The number and type of parameters following the name of the method on **`ExternalCall`** should match the number and type of the parameters of the method itself.

In the **Start Run** Expression of the **Model Expressions**, **`ExternalCall`** is used to create a new instance of **`com.demo.ArraySort`**. This new instance is returned to an Attribute of type **`Object`** (**Model.ArraySort**). Since the **`ArraySort`** constructor requires no parameters, no other parameters are listed after the constructor name in **`ExternalCall`**.

The **Generate Task Orders** Activity has an **`ExternalCall`** System Method in its **Release Entity** Expression. The parameters are the Model Attribute that contains the instantiated class (**Model.ArraySort**), the name of the method (**`sort2DArray`**), the array itself (**Entity.Task**), and the dimension to sort. The **`sort2DArray`** method returns a sorted array back to the Attribute **Entity.Task**. Note that if the original array needs to be maintained, the sorted array can be returned to another Object Attribute or to an **`ANYOBJ`** local variable.

FileDemo.spm

This model demonstrates the **`OpenFile`**, **`CloseFile`**, **`ReadFromFile`**, and **`WriteToFile`** System Methods. There are two ASCII files in the model's directory, **`DelayTimes.txt`** and **`ResourceLevels.txt`**. **`DelayTimes.txt`** contains the delay durations for **Delay 1** and **Delay 2**. **`ResourceLevels.txt`** contains the number of units for the Resources **Resource1** and **Resource2**.

In the **Start Simulation** Expression of the **Model Expressions** (**Define/Model Expressions**), **`ResourceLevels.txt`** is opened for input and is assigned to the Attribute **Model.InFile** (Model Attribute with mode **`Object`**). **`CycleTimes.txt`** is opened for output and is assigned to the Attribute **Model.OutFile** (Model Attribute with mode **`Object`**). **`OpenFile`** requires two **`STRING`** parameters. The first parameter is either **`"Input"`** or **`"Output"`**. The second parameter is the name of the file. If a complete path is not included, the file is assumed to be in the model's directory. The file will be created if the type is **`"Output"`** and the file does not exist (as is the case for **`CycleTimes.txt`**). Next, the levels for **Resource1** and **Resource2** are read from **`ResourceLevels.txt`**. Note that the first parameter of **`ReadFromFile`** is the Attribute that has the reference to the file. The remaining parameters are the variables that will hold the values read from the file. The number of parameters required depends on the format of the file being read. The type of the parameters (**`INTEGER`**, **`REAL`**, etc.) must match the type of

the value being read from the file. In this example, the first value is a **STRING** and the second value is an **INTEGER**. Below is the content of **ResourceLevels.txt**.

```
|Resource1| 4
|Resource2| 4
```

The first value is the name of the Resource, and the second value is the number of units of the Resource. Since the first value is a **STRING**, vertical bars (|) are placed around the value. In this instance the vertical bars are not required since the **STRING** value has no spaces. However, it is good practice to put vertical bars around all **STRING** values in an input file.

Next, **ResourceLevels.txt** is closed using the **CloseFile** System Method. The only parameter for **CloseFile** is the Attribute that references the file. **Model.InFile** is then reused to open **DelayTimes.txt**. Below are the first three rows of the file.

```
|Delay1 times|    |Delay2 times|
8.810151858 17.52339693
0.753893822 13.22932585
```

The file contains two columns of values. **ReadFromFile** is used to read the header row. Note that vertical bars are required since the **STRING** values contain spaces. The last statement in the **Start Simulation** expression is **WriteToFile**. This statement writes out a header to the output file. As with **ReadFromFile**, the first parameter for **WriteToFile** is the Attribute that has the reference to the file. The remaining parameters contain what is to be written to the file. The character **"^"** is a tab, and the character **"/"** is for a new line. **WriteToFile** does not start a new line automatically, it simply appends to the end of the file. Thus **"/"** must be used to force a new line. Similarly **ReadFromFile** does not automatically start at the next line of text.

The **Release Entity** Expression of the Generate Activity reads the next values from the file. The values are stored in **Entity.DelayTime1** and **Entity.DelayTime2**. These Entity Attributes have a mode of **REAL** and are used in the **Delay 1** and **Delay 2** Activities respectively.

The **Accept Entity** Expression of the Dispose Activity uses **WriteToFile** to output the current simulation time (**SimTime**) and the cycle time of the Entity.

In the **End Simulation** Expression of the **Model Expressions**, **CloseFile** closes **DelayTimes.txt** and **CycleTimes.txt**. The files must be closed with **CloseFile** for SIMPROCESS to release the files.

GenerateEntity.spm

This model demonstrates the **GenerateEntity** System Method. This System Method causes the Generate Activity specified in the command to generate at least one Entity.

The model generates 10 **Entity20** Entities. When each **Entity20** reaches **Dispose Entity20**, **Generate Entity** generates an **Entity21** Entity. This is caused by the **`GenerateEntity`** command in the **Accept Entity** Expression of **Dispose Entity20**. In this example, the **Generate Entity** Activity does not have any Entity generation schedules. However, the **`GenerateEntity`** System Method will work for any Generate Activity, whether schedules exist or not. The first parameter must be the name of the Generate Activity that will generate an Entity. If the Generate Activity name is the only parameter, then the default Entity and the default quantity set in the Generate Activity will be used to determine how many of what Entity to generate. The **Accept Entity** Expression of **Dispose Entity20** shows the alternatives of including the name of the Entity to generate, the quantity to generate, or both. **`GenerateEntity`** also has an optional time parameter that is not demonstrated in this model. The time parameter allows the Entity generation to be scheduled at a specific time. See **ConfigureWithDatabase.spm** for a **`GenerateEntity`** example that uses the time parameter.

Note that since the first parameter of **`GenerateEntity`** is the name of a Generate Activity, the name entered must be unique within the model.

GetResourceandFreeResource.spm

This model demonstrates the **`GetResource`** and **`FreeResource`** System Methods. Also, the **`DrawRealSample`** System Method is used. **`GetResource`** and **`FreeResource`** can **only** be used in the **Accept Entity** Expression of Activities that can process Resources. **`GetResource`** can be used in any Activity that can acquire Resources (thus, the Free Resource Activity is excluded), and **`FreeResource`** can be used in any Activity that can release Resources (thus, the Get Resource Activity is excluded).

There are three types of Resources, **Truck A**, **Truck B**, and **Truck C**. An entering customer randomly requires a particular type of Resource. This is set in the **Accept Entity** Expression of **Get Truck**. Based on a probability, **`"Truck A"`**, **`"Truck B"`**, or **`"Truck C"`** is assigned to **Entity.TruckType**. **Entity.TruckType** is then used in a **`GetResource`** command to request that type of Resource. The first parameter is the name of the Resource to acquire. The second parameter is the number of units to acquire. This parameter can be **`REAL`**, **`INTEGER`**, or **`STRING`**. **`STRING`** would be used for a distribution (like "Int(2, 5)"); The third parameter is optional. It is a **`STRING`** parameter that is the tag. Just as the Get Resource Activity can designate a tag, the **`GetResource`** System Method can do the same thing.

The **`FreeResource`** System Method is used in the **Accept Entity** Expression of **Deliver Load**. Even though the **`FreeResource`** System Method must be used in the Accept Entity Expression, Resources are not released until the Entity has finished processing in that Activity. The first parameter of the **`FreeResource`** System Method is the name of the Resource to release. In this example, **`"AnyResource"`** is used. This means that any Resources being used by the Entity will be released. If a specific Resource is entered, then only that Resource will be released. There are two other optional parameters. The tag can be specified, and a **`BOOLEAN`** that sets whether or not to consume consumable Resources.

See “Getting and Freeing Resource Using Expressions” in Chapter 2 of Part B of the *SIMPROCESS User’s Manual* for a full discussion of **`GetResource`** and **`FreeResource`**.

ReleaseEntity.spm

This model demonstrates the **`Gate`**, **`GetEntity`**, and **`ReleaseEntity`** System Methods. These System Methods are used to manipulate Gate Activities. Note that these System Methods do not have to be used in Gate Activities. Also, the **`DrawIntegerSample`** System Method is used.

The model generates 10 **Truck** Entities, each having a different load capacity (see **Release Entity** Expression of **Generate Trucks**). These Trucks are held in the **Hold Truck for Order** Gate Activity. **Generate Orders** generates three **Order** Entities every two hours. The **Order** Entities enter **Get Order Resource** to acquire the **Order** Resource. This is done so the **Order** Entities will process one at a time. The size of each **Order** is set in the **Release Entity** Expression of **Generate Orders**. Since the **Order** size varies, a **Truck** with a load capacity large enough to handle the **Order** must be release from **Hold Truck for Order**. The **Release Entity** Expression of **Get Order Resource** searches the queue of **Truck** Entities in the Gate Activity **Hold Truck for Order** for a **Truck** large enough to handle the **Order**.

The **`Gate`** System Method returns a reference to a Gate Activity only. If the name entered does not exist or is not a Gate Activity, an error will occur. This reference is returned to the local **`ANYOBJ`** variable **`gate`**. The Gate Activity System Attribute **`NumberOnHold`** is used to loop through the Entities held at the Gate Activity. The **`GetEntity`** System Method returns an Entity reference. The first parameter must be the reference to the Gate Activity. The second parameter is the position in the queue and must be an **`INTEGER`**. **`ReleaseEntity`** causes the specified Entity at the designated Gate Activity to release from the queue. Again, the first parameter is the reference to the Gate Activity. The second parameter is not the position in the queue, but is the **`SequenceNum`** (Entity System Attribute) of the Entity to release, which must be an **`INTEGER`**. **`ReleaseEntity`** returns **`TRUE`** if the command was successful, **`FALSE`** otherwise. Thus, the **`BOOLEAN`** Entity Attribute **OrderFilled** is set to **`TRUE`** if the request for release was successful. This attribute (**Entity.OrderFilled**) is used in the Branch Activity **Order Filled?** to determine the proper path.

There is another Gate related System Method called **`EntityExists`** that is not demonstrated in this model. The first parameter is the reference to the Gate Activity, and the second parameter is the **`SequenceNum`** of the Entity to find. **`EntityExists`** returns the position (an **`INTEGER`**) in the queue of an Entity. Zero is returned if the Entity is not being held at the specified Gate Activity. **`EntityExists`** eliminates looping to find an Entity if the **`SequenceNum`** is already known.

SpreadsheetDemoExp.spm

This model demonstrates the **OpenSpreadsheet**, **CloseSpreadsheet**, **ReadFromSpreadsheet**, and **WriteToSpreadsheet** System Methods. There are two input files located in the model's directory: **ssdemo.xls** and **ssdemo.xml**. The first file (**ssdemo.xls**) is an Excel workbook, and **ssdemo.xml** is an XML spreadsheet. SIMPROCESS reads from and writes to a Workbook and an XML spreadsheet. The model defaults to use **ssdemo.xml** for the input file. When the simulation is run, the output file **ssdemoout.xls** will be created in the model's directory.

In the **StartSimulation** Expression of the **Model Expressions** (**Define/Model Expressions**), the two spreadsheets (**ssdemo.xml** and **ssdemoout.xls**) are opened, one for input and one for output. (The files **ssdemo.xml** and **ssdemo.xls** contain the delay times for each of the Delay Activities. The current simulation time (**SimTime**) and the cycle time of each Entity are written to **ssdemoout.xls**.) The **OpenSpreadsheet** System Method requires two parameters, both of type **STRING**. The first parameter is either **"Input"** or **"Output"**. The second parameter is the name of the file. If a complete path is not included, the file is assumed to be in the model's directory. If the type is **"Output"**, and the file does not exist, the file will be created. A reference to the spreadsheet file is returned so an Attribute of type **Object** must be used as the variable being assigned.

The **StartSimulation** Expression of the **ModelExpressions** and the **AcceptEntity** Expression of **Dispose** both contain **WriteToSpreadsheet** commands. The first parameter is the Attribute that contains the reference to the spreadsheet file. The second parameter is a **STRING**, which is the name of the sheet. The sheet will be created if it does not exist. The row is the third parameter, and the column is the fourth parameter. Both the row and column parameters must be of type **INTEGER** and must be greater than or equal to 1. The final parameter is the value to place in the specified cell. The value can be **REAL**, **INTEGER**, **BOOLEAN**, or **STRING**.

The **Accept Entity** Expression of each Delay Activity contains a **ReadFromSpreadsheet** command. Again, the first parameter must be the Attribute that contains the reference to the spreadsheet file. The name of the sheet is the second parameter (**STRING**). The third and fourth parameters are the row and column to read. The row and column parameters must be type **INTEGER**. The final parameter is the local variable or Attribute that is to receive the value from the spreadsheet. An error will occur if the type of the variable does not match the type of the value returned.

See "Interfacing With A Spreadsheet" in Chapter 2 of Part B of the *SIMPROCESS User's Manual* for a complete discussion on using spreadsheets with SIMPROCESS and information on differences between a Workbook (**.xls**) and an XML spreadsheet (**.xml**).

The model **SpreadsheetDemo.spm** uses the Input Sources and Export Results features to read from and write to spreadsheets. See Chapter 3 of Part A of the *SIMPROCESS User's Manual* for information on using Input Sources and Chapter 4 of Part B of the *SIMPROCESS User's Manual* for information on exporting results to a spreadsheet.

ConfigureWithDatabase.spm

This model demonstrates the **`CreateResource`**, **`SetResourceDowntime`**, **`SetResourceCost`**, **`SetResourceExpression`**, **`CreateEntityType`**, **`CreateAttribute`**, **`SetActivityExpression`**, **`Attribute`**, **`ElapsedTime`**, and **`DateTime`** System Methods. These methods are used to configure a SIMPROCESS model from a database. Entity Types, Resources and Attributes are created and Expressions are set at the start of the simulation based on information from the **`WidgetManufacturing.mdb`** database and the **`OrderAssembly.txt`** and **`OrderComplete.txt`** files. The System Methods **`OpenDatabase`**, **`ReadFromDatabase`**, **`GetResult`**, **`GetNext`**, **`CloseDatabase`**, **`CreateMap`**, **`PutInMap`**, **`GetFromMap`**, **`ClearMap`**, **`CreateArray`**, **`SetArrayValue`**, **`GetArrayValue`**, **`GenerateEntity`**, **`GetResource`**, **`FreeResource`**, **`UpdateDynamicLabel`**, **`SUBSTR`**, and **`HALT`** are also used but not specifically discussed.

The **Start Run** expression of the **Model Expressions** (**Define/Model Expressions**) reads the database and files to set up the simulation. First, a connection to the database is established, and a Map is created to hold the names of the Resources that are created. The **TechnicianTypes** table of the database contains the information required to create Technicians (Resources). The information in this table is retrieved using **`ReadFromDatabase`** and then a loop cycles through the records to create the Resources.

- Information from the **Employees** table retrieved using **`ReadFromDatabase`** is used to determine the number of units of each Resource.
- The **`CreateResource`** statement actually creates the Resource with the name obtained from the **TechnicianTypes** table and the number of units derived from the **Employees** table.
- After the Resource is created, **`SetResourceDowntime`** applies one of the predefined global Resource Downtimes (**Define/Resource Downtimes**) to the new Resource, and **`SetResourceCost`** sets the hourly cost of the Resource as determined from the **TechnicianTypes** table.
- The **Expressions** table contains some of the Expressions required in the model. The **ExpressionUse** table determines which Resources use the Expressions found in the **Expressions** table. The type of the Expression (**Get Resource** or **Free Resource**) and the Expression code are retrieved from the database and **`SetResourceExpression`** applies the Expression to the Resource.
- The name of the Resource is placed in **Model.TechnicianNameMap** with a default name created with **`TechnicianType`** and the **`id`** as the Map key.

After the loop to create the Resources, a check is made to ensure Resources were defined in the **TechnicianTypes** table. If there are none defined, the simulation is ended. Next, two Maps are created. **Model.TechnicianMap** will hold arrays of the number of each Resource required for

each product, and **Model.AssemblyTimeMap** will hold the time required to assemble each product.

The next section of the **Start Run** Expression creates the Entities for the model. The number of Entities to be created is retrieved from the **Products** table, then an array (**Model.ProductNameArray**) is created to hold the Entity (product) names. The **Products** table is queried for all the product information and, as with the Resources, a loop begins that creates each of the Entities.

- The Entity name is placed in **Model.ProductNameArray**.
- An **`INTEGER`** array is created and populated with the number of each Technician (Resource) required to assemble this product (Entity). The array is placed in **Model.TechnicianMap** with the name of the Entity as the Map key.
- **`CreateEntityType`** creates the Entity with the name and icon retrieved from the **Products** table.
- The time required to assemble the product (Entity) is placed in **Model.AssemblyTimeMap** with the name of the Entity as the Map key.
- **`CreateAttribute`** creates a global Resource Attribute with the same name as the Entity just created. The Attribute is an **`INTEGER`** attribute that collects time-weighted statistics.

Next a global Resource Attribute named **Index** is added to the model with **`CreateAttribute`**. This attribute is used in each Resource's **Get Resource** and **Free Resource** Expressions. Below is the **Get Resource** Expression from the **Expressions** table.

```
attr : ANYOBJ;
attr := Attribute(Entity.Name);
attr := attr + Entity.Quantity[Index];
```

The **`Attribute`** Expression statement is used to retrieve the Attribute with the same name as the Entity. This Expression simply adds the number of units of the Resource required for this Entity. The **Free Resource** Expression is the same except it subtracts the number of units of the Resource required for this Entity. Since time-weighted statistics were set for each of these Attributes when they were created, statistics for the number of each Resource required for each Entity Type can be determined.

`CreateAttribute` is used to create a global Entity Attribute named **Quantity**. This is the same Attribute used above in the **Get Resource** and **Free Resource** Expressions. Note that **Quantity** is an array Attribute. The values for **Quantity** are set in the **Accept Entity** Expression of the **Order Assembly** Activity. This expression is stored in the **`OrderAssembly.txt`** file and is added to the Activity at the end of the **Start Run** Expression.

The final loop in the **Start Run** Expression schedules Entity generation. The order records are retrieved from the **Orders** table. **`ElapsedTime`** is used to determine each order's arrival time in the simulation time unit set in the **Run Settings**. **`DateTime`** returns the current date and time of the simulation. Since this is the **Start Run** Expression, **`DateTime`** will return the starting

date and time set in the **Run Settings**. Thus, in this example, **`ElapsedTime`** is returning the time interval between the starting date and time of the simulation and the date and time of each order. The quantity of each product (Entity) is retrieved and **`GenerateEntity`** schedules the Entity generation.

Finally, the **Accept Entity** Expression for the **Order Complete** activity is retrieved from **`OrderComplete.txt`**, and the database connection is closed.

The **End Run** Expression clears each Map.

The **Accept Entity** Expression for **Order Assembly** gets the appropriate array from **Model.TechnicianMap** and transfers the values from the array to the Entity Attribute **Quantity**. Then **`GetResource`** and **`FreeResource`** statements are executed for each Resource. Finally, the delay time for the Entity is retrieved from **Model.AssemblyTimeMap**.

The **Accept Entity** Expression of **Order Complete** records the ending time of the simulation and displays the date in the dynamic label.

InterruptDemo.spm

This model demonstrates the **`InterruptActivity`** and **`InterruptEntity`** System Methods. The model represents a car repair business. The business is open from 7:00 to 19:00 Monday through Friday. A **Car** Entity arrives on average every 10 minutes based on an Exponential distribution. From 1 to 3 items on each car need repair. If the estimate is accepted the **Car** enters the **Repair Car** process. Immediately upon entering **Repair Car** a clone Entity **Customer** is created. This Entity represents the customer that brought the car and it delays the amount of time the customer is willing to wait before canceling the repairs. The amount of time the **Customer** is willing to wait depends on the number of items needing repair (see the **Accept Entity** Expression of **Customer Wait**). At the end of this time the **Customer** Entity enters **Interrupt Repairs**. If the original **Car** Entity still exists and is in **Make Repairs**, **`InterruptEntity`** is called to stop the repairs and send the **Car** to the Transfer Activity **Customer Canceled Repairs**.

There are 10 **Mechanic** Resources so up to 10 cars can be in service at a time. At the end of each day (19:00) the Resources go down and interrupt any repairs in process. The repairs continue at the beginning of the next work day. At this point it is assumed the customer will not cancel the repairs.

The **End of Day** Process controls what happens to **Car** Entities that have not started service before closing. An **End of Day** Entity is generated at 19:00 each day. The Accept Entity Expression of **Release Waiting Cars** interrupts the Activities **Get Mechanic**, **Make Estimate**, **Review Estimate**, and **Customer Wait**. This causes these Activities to empty any Entities still remaining at the end of the business day.

In this particular example the optional **`Entity State`** parameters for **`InterruptActivity`** were not used. This is because all Entities not in **Make Repairs**, no matter what state, needed to be interrupted at the end of the day. Up to two **`Entity State`** parameters can be included. The permissible values are **`All`** (the default if no **`Entity State`** parameter is specified), **`Processing`**, **`WaitingForResource`**, or **`HoldingForCondition`**.