Simio

Forward Thinking

Simio API Note: Emulating World-Time

June2018 (Dhouck)

Contents

Simio
Forward Thinking

Overview

This API Note describes how Simio can be configured to accomplish these two tasks:

1. Emulating World-Time; that is, running the Simulation so that each second of simulation time (Sim-Time) corresponds to 1 second of World-Time (or, if you prefer, Real-Time or WallClock-Time).
2. Communicating with an external device (external to Simio), which is also in World-Time.

This project also uses the singleton Entity Data technique (refer to Entity Data Handling document) for keeping track of Entity data.

This Note describes some complex programming topics. It assumes that the reader is familiar with C# and .NET technologies such as threading, asynchronous operations, and the Singleton pattern of programming.

Some Background Information on the Simio Engine

The Simio engine (which we'll just call the Engine) is the logic that implements the simulation and planning logic of Simio. When used as a desktop application, it usually includes a UI.

Generally speaking, a simulation run within the Engine is single threaded, meaning that any delays incurred by an Entity affect the whole run. For example, if an Entity decided to do a long-running operation (such as a large synchronous read or write to an external device), the Simio Engine (including the UI) will "hang" for the duration of that operation.

Note: We are here talking about a single "run". When Simio is doing things such as Experiments, it scales by using a separate thread for each Experiment.

Simio provides the run-time API which allows users to create custom Steps (User Extensions). These are implemented as calls to .NET methods that the user can write and are implement by conforming to the published Interfaces. As such, anything that can be done in .NET can be done in these steps. When a Step is executed it is done by a Token, which is a proxy for an Entity (i.e. the green triangles that are seen flowing through the Simio model). These Entities are born, they travel through the simulation, and they are destroyed.

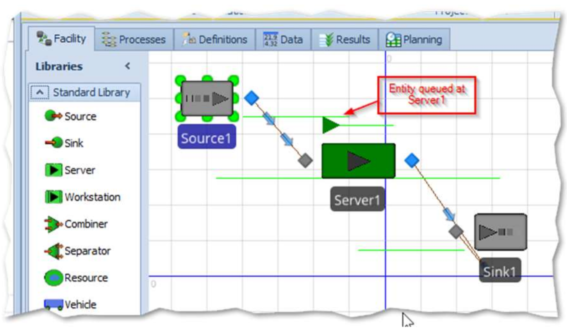


Figure 1 - The Source-Server-Sink Model

Entities are instances of Simio Intelligent Objects, and can share data using the normal Simio mechanism of Properties, State variables, etc. They can also execute logic as the travel through the system using Simio expressions.

In this document we'll also show how the User Extension (User Step) can employ the C# singleton as another way that Entities can share data.

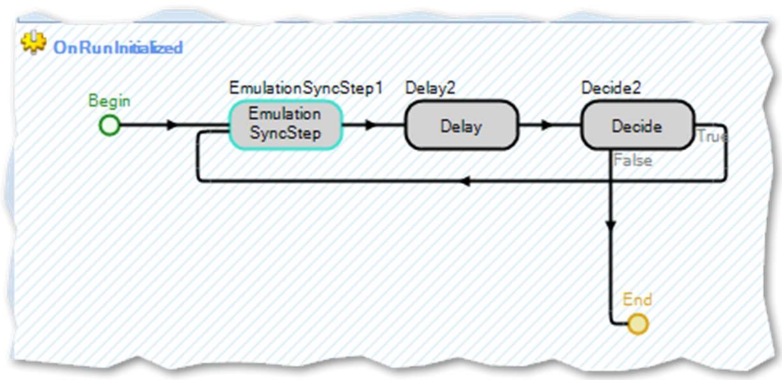
Simio
Forward Thinking

Extending Entities with the API

At the time of this writing, Simio does not have a built-in way of running in World-Time. However, it can be easily accomplished with a few custom User Steps.

World-Time Emulation

The World-Time emulation is achieved using the EmulationSyncStep, which is run at the OnRunInitialized event in conjunction with a Delay and Decide step.

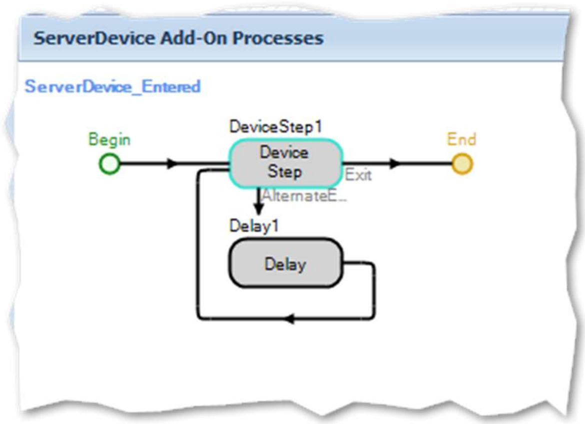


The logic for the step is quite simple: At each increment of Sim-Time (determined by the small Delay step; for example 1 second) and if the Emulation SyncStep determines that the Sim-Time now exceeds World-Time (which is usually the case), then the simulation is paused for the difference. This loop runs for the duration of the simulation.

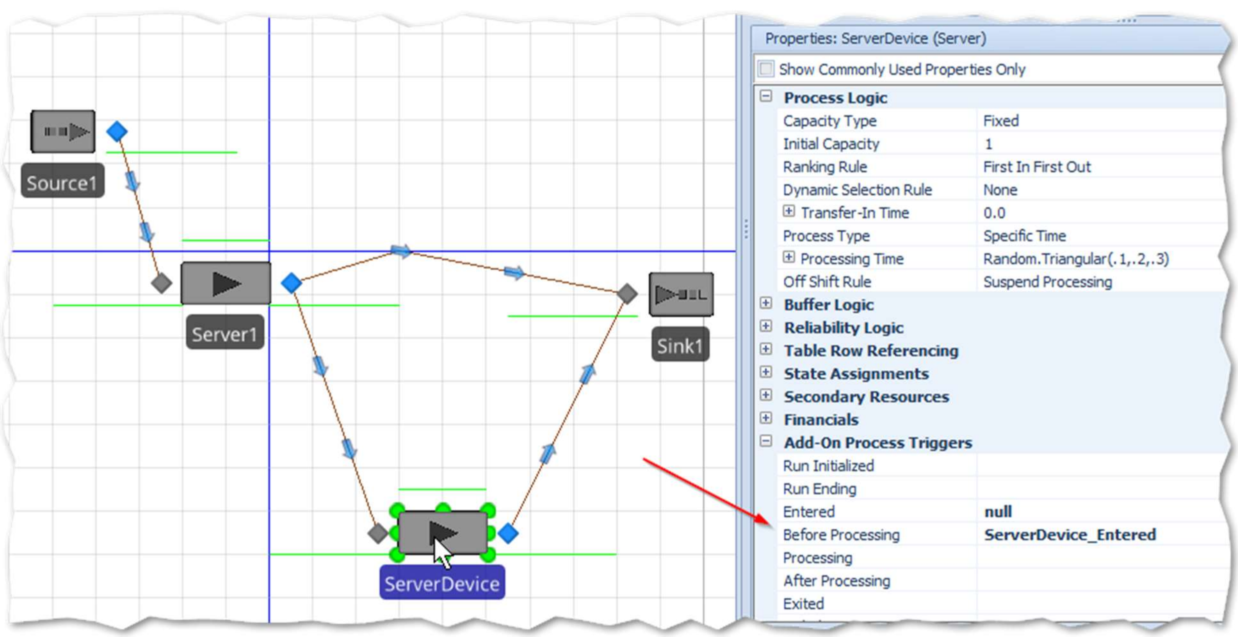
Simio
Forward Thinking

External Device Communications in World-Time

The logic for communicating with a device in World-Time involves a User "Device Step" that upon first entry requests information from the device. It then checks whether the device has responded. If there is no response, then a small Simulation Delay is called and the Response is looked for again.



To illustrate these techniques, a very basic SourceServerSink (SSS) model was created, and modified to have an extra "ServerDevice".

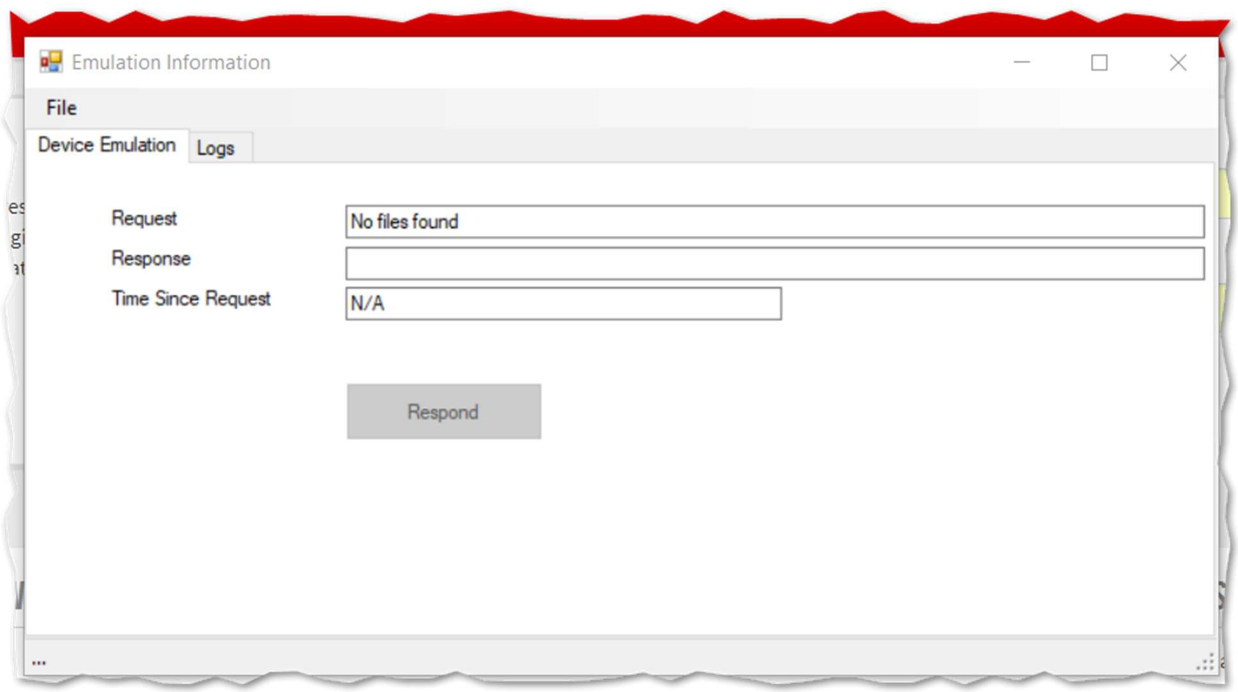


Since the example “Device” employs files for communications, a Property FolderPath exists on the Device Step, and defaults to “C:\(test)”

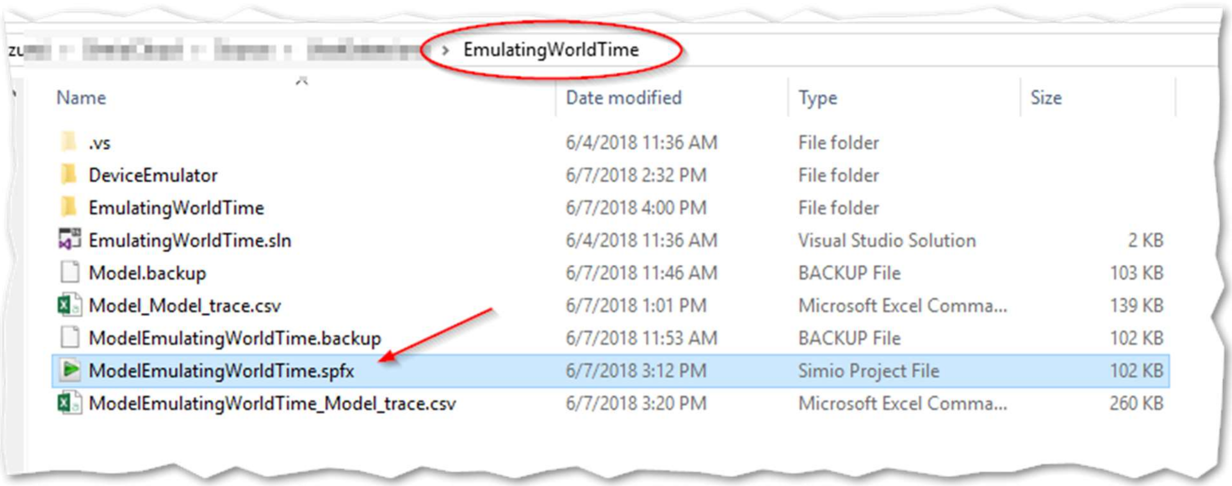
Keep in mind that at this point this example project was built for demonstration purposes. For production purposes it would need to be ruggedized (e.g. better error handling, startup, and shutdown).

For this project there is also a separate DeviceEmulator program. It is a simple WinForms application as included in the project folder.

There is also an animated Gif (SimioEmulation1.gif) which demonstrates what you should see when you run the project and also an example use of the DeviceEmulator.



The Simio Model (ModelEmulatingWorldTime.Spfx) resides directly under the EmulatingWorldTime folder.


EmulatingWorldTime
Name
Date modified
Type
Size
.vs
6/4/2018 11:36 AM
File folder
DeviceEmulator
6/7/2018 2:32 PM
File folder
EmulatingWorldTime
6/7/2018 4:00 PM
File folder
EmulatingWorldTime.sln
6/4/2018 11:36 AM
Visual Studio Solution
2 KB
Model.backup
6/7/2018 11:46 AM
BACKUP File
103 KB
Model_Model_trace.csv
6/7/2018 1:01 PM
Microsoft Excel Comma...
139 KB
ModelEmulatingWorldTime.backup
6/7/2018 11:53 AM
BACKUP File
102 KB
ModelEmulatingWorldTime.spfx
6/7/2018 3:12 PM
Simio Project File
102 KB
ModelEmulatingWorldTime_Model_trace.csv
6/7/2018 3:20 PM
Microsoft Excel Comma...
260 KB

User Step Code

Again, there are two User Steps:

1. EmulationSync Step, and
2. Device Step

EmulationSync Step

The EmulationSync Step uses a singleton to store Sim-Time and World-Time data. This could be done with Simio objects (Properties and States) if you preferred.

On start, there is a one-time store of the starting Sim-Time and World-Time.

At each run, the difference (delta) between the Sim-Time and World-Time is performed, and a “Thread.Sleep” is called for that duration to suspend the thread (which is the simulation thread).

```csharp
/// <summary>
/// Method called when a process token executes the step.
/// </summary>
public ExitType Execute(IStepExecutionContext context)
{
    EmulatingWorldTimeSingleton eContext = EmulatingWorldTimeSingleton.Instance;

    // One time initialization
    if ( eContext.Ticks == 0 )
    {
        eContext.StartSimHour = context.Calendar.TimeNow;
        eContext.StartWorldTime = DateTime.UtcNow;
    }

    double simNow = context.Calendar.TimeNow; // Get Sim-Time

    DateTime worldTimeNow = DateTime.UtcNow;
    double deltaWorldSeconds = worldTimeNow.Subtract(eContext.StartWorldTime).TotalSeconds;

    // Usually, simulation time will be ahead of (faster than) world time, so we'll slow it down.
    double deltaSimSeconds = (simNow - eContext.StartSimHour) * 3600.00;

    double adjustmentSeconds = deltaSimSeconds - deltaWorldSeconds;

    if (adjustmentSeconds > 0)
    {
        // This will suspend the thread for the given number of milliseconds, which
        // allows the simulation time to sync with world time.
        System.Threading.Thread.Sleep((int)(1000.0 * (adjustmentSeconds)));
    }
    else // This would likely only happen during debugging
    { }

    eContext.BumpSimTime();

    return ExitType.FirstExit;
}
```

Note that if you are holding up time by stopping the code in the debugger, when you finally run without breakpoints, the simulation will proceed at its full rate until it catches-up to the world time. At which point it will continue to be synced with World-Time.

Device Step

The emulation of a device uses an external program (Device Emulator) and the file system to demonstrate a Request-Response type of device.

This step also uses a singleton structure to store data, but it was added mostly as a demonstration of how data (even large amounts of data) could be associated with an Entity. Note that the unique name of Entities makes it a rather convenient “key”.

It begins with an Entity entering the special “DeviceServer”

We need to differentiate whether this is our first time into the step, and we’ll do this by checking if we already have built a Request file.

```
public ExitType Execute(IStepExecutionContext context)
{
    // Get the reference to our singleton.
    EntityDataSingleton eds = EntityDataSingleton.Instance;

    // Just a 'one shot' to store our folder path
    if ( string.IsNullOrEmpty(eds.FolderPath))
    {
        IPropertyReader reader = _properties.GetProperty("FolderPath") as IPropertyReader;
        string folder = reader.GetStringValue(context);
        eds.FolderPath = folder;
    }

    var simioEntity = context.AssociatedObject;
    string name = simioEntity.HierarchicalDisplayName;

    string key = "";
    if (!name.StartsWith("DefaultEntity"))
        return ExitType.FirstExit;

    string[] tokens = name.Split('.');
    key = name.Replace('.', '$'); // Makes it more convenient for parsing later.
```

If the Request file does not exist then we’ll build one.

Regardless, we’ll do a check for a Response file. If it exists we have our answer from the Device, so we can take FirstExit.

If it doesn’t exist (the Device has not yet responded), we take door #2 (AlternateExit) which (in our Process) leads us to a short Simio-Delay step and then back again to this Device Step.

Simio

Forward Thinking

mulatingWorldTime | EmulatingWorldTime.DeviceStep | Exe

```
string[] tokens = name.Split('.');
key = name.Replace('.', '$'); // Makes it more convenient for parsing later.

// Fetch or create our entity data that will accompany the entity.
// This data is stored in the singleton's dictionary, and fetched by key.
EntityData eData = null;
if (!eds.EntityDataDict.TryGetValue(key, out eData))
{
    eData = new EntityData(key);
    eds.EntityDataDict.TryAdd(eData.Key, eData);
}

eData.TimeRequestMade = DateTime.UtcNow;

string folderPath = eds.FolderPath;
string requestFilePath = Path.Combine(folderPath, $"Request-{key}.txt");

// If the request file is there, then we are waiting for a response, so
// don't generate another request
if (!File.Exists(requestFilePath))
{
    logit(EnumLogFlags.Information, $"Request file written to={requestFilePath}");

    // Put a request file, and then we'll poll for the response.
    File.WriteAllText(requestFilePath, "(data body: info the device might want)");
}

// Look for the Response file. If found, then Exit First, if not, then exit Alternate
string responseFilePath = Path.Combine(folderPath, $"Response-{key}.txt");

if (File.Exists(responseFilePath))
{
    string contents = File.ReadAllText(responseFilePath);
    File.Delete(responseFilePath);

    logit(EnumLogFlags.Information, $"Found file={responseFilePath} Contents={contents}");
    return ExitType.FirstExit;
}
else
{
    return ExitType.AlternateExit;
}
```