



Enterprise Architect

User Guide Series

Executable State Machines

Author: Sparx Systems

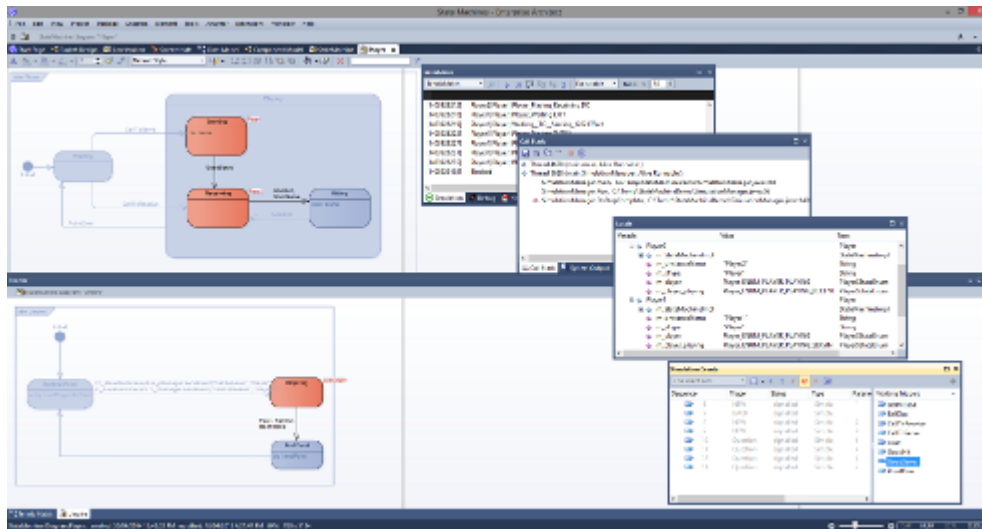
Date: 15/07/2016

Version: 1.0

Table of Contents

Executable State Machines	3
Modeling Executable Statemachines	5
Executable StateMachine Artifact	10
Code Generation for Executable State Machines	12
Debugging Execution of Executable State Machines	15
Execution and Simulation of Executable State Machines	17
Example Executable State Machine	18
Deferred Event Pattern	22
States With Multiple State Regions	28
Entry and Exit Points (Connection Point References)	29

Executable State Machines



Executable State Machines provide a powerful means of rapidly generating, executing and simulating complex state models. In contrast to dynamic simulation of State Charts using Enterprise Architect's Simulation engine, Executable State Machines provide a complete language-specific implementation that can form the behavioral 'engine' for multiple software products on multiple platforms. Visualization of the execution uses and integrates seamlessly with the Simulation capability. Evolution of the model now presents fewer coding challenges. The code generation, compilation and execution is taken care of by Enterprise Architect. For those having particular requirements, each language is provided with a set of code templates. Templates can be customized by you to tailor the generated code in any ways you see fit.

These topics will introduce you to the basics of modeling Executable State Machines and tell you how to generate and simulate them.

Overview of Building and Executing State Machines

Building and using Executable State Machines is quite straight forward, but does require a little planning and some knowledge of how to link the different components up to build an effective executing model. Luckily you do not have to spend hours getting the model right and fixing compilation errors before you can begin visualizing your design.

Having sketched out the broad mechanics of your model, you can generate the code to drive it, compile, execute and visualize it in a matter minutes. These points summarize what is required to start executing and simulating State Machines.

Facility	Description
Build Class and State models	The first task is to build the standard UML Class and State models that describe the entities and behavior to construct. Each Class of interest in your model should have its own State Machine that describes the various states and transitions that govern its overall behavior.
Create an Executable Statemachine Artifact	Once you have modeled your Classes and State models, its time to design the Executable Statemachine Artifact. This will describe the Classes and objects involved, and their initial properties and relationships. It is the binding script that links multiple objects together and it determines how these will communicate at runtime. Note that it is possible to have two or more objects in an Executable Statemachine Artifact as instances of a single Class. These will have their own state and behavior at run-time and can interact if necessary.
Generate Code and	Whether it is Javascript, C++, Java or C# that you need, EA's engineering

Compile	capabilities provide you with a powerful tool, allowing you to regenerate the executable at any time, and without the loss of any customized code you may have made. This is a major advantage over a project's lifetime. It is probably also worth noting that the entire code base generated is independent and portable. In no way is the code coupled with any infrastructure used by the simulation engine.
Execute State Machines	So how do we see how these state machines behave. One method is to build the code base for each platform, integrate it in one or more systems, examining the behaviors, 'in-situ', in perhaps several deployment scenarios. Or we can execute it with Enterprise Architect. Whether it is Java, Javascript, C, C++ or C#, EA will take care of creating the runtime, the hosting of your model, the execution of it's behaviors and the rendition of all state machines.
Visualize State Machines	Executable Statemachine visualization integrates with Enterprise Architect's Simulation tools. Watch state transitions as they occur on your diagram and for which object(s). Easily identify objects sharing the same state. Importantly, these behaviors remain consistent across multiple platforms. You can also control the speed at which the machines operate to better understand the timeline of events.
Debug State Machines	When states should change but do not, when a transition should not be enabled but is, when the behavior is in short undesirable and not immediately apparent from the model, we can turn to debugging. Enterprise Architect's Visual Execution Analyzer comes with debuggers for all the languages supported by Executable State Machine code generation. Debugging provides many benefits, one of which might be to verify / corroborate the code attached to behaviors in a State Machine to ensure it is actually reflected in the executing process.

Modeling Executable Statemachines

Most of the work required to model an Executable State Machine is standard UML based modeling of Classes and State models. There are a couple of conventions that must be observed to ensure a well formed code base. The only novel construct is the use of a stereotyped Artifact element to form the configuration of an Executable State Machine instance or scenario. The Artifact is used to specify details such as:

- The code language (Javascript, C#, Java, C++ including C)
- The Classes and State Machines involved in the scenario
- The instance specifications including run-state; note that this could include multiple instances of the same State Machine, for example where a 'Player' Class is used twice in a Tennis Match simulation.

Basic Modeling Tools and Objects for Executable State Machines

This table details the primary modeling elements used when building Executable State Machines.

Object	Details
Classes and Class Diagrams	Classes define the object types that are relevant to the State Machine(s) being modeled. For example, in a simple Tennis Match scenario you might define a Player, a Match, a Hit and an Umpire Class. Each will have its own State Machine(s) and at runtime will be represented by object instances for each involved entity. See the <i>UML modeling guide</i> for more information on Classes and Class diagrams.
State Machines	For each Class you define that will have dynamic behavior within a scenario, you will typically define one or more UML State Machines. Each State Machine will determine the legal state-based behavior appropriate for one aspect of the owning Class. For example, it is possible to have a State Machine that represents a Player's emotional state, one that tracks his current fitness and energy levels, and one that represents his winning or losing state. All these State Machines will be initialized and started when the State Machine scenario begins execution.
Executable StateMachine Artifact	<p>This stereotyped Artifact is the core element used to specify the participants, configuration and starting conditions for an Executable State Machine. From the scenario point of view it is used to determine which Instances (of Classes) are involved, what events they might Trigger and send to each other, and what starting conditions they operate under.</p> <p>From the configuration aspect, the Artifact is used to set up the link to an analyzer script that will determine output directory, code language, compilation script and similar. Right clicking on the Artifact will allow you to generate, build, compile and visualize the real time execution of your State Machines.</p>

State Machine Constructs Supported

This table details the State Machine constructs supported and any limitations or general constraints relevant to each type.

Construct	Description
State Machines	<p>Supported</p> <p>Simple State Machine: State Machine has one region.</p>

	<p>Orthogonal State Machine: State Machine contains multiple regions.</p> <p>Top level region (owned by State Machine) activation semantics:</p> <p>Default Activation: When the State Machine starts executing.</p> <p>Entry Point Entry: Transitions from Entry Point to vertices in the contained regions.</p> <ul style="list-style-type: none"> <i>Note 1: In each Region of the State Machine owning the Entry Point, there is at most a single Transition from the entry point to a Vertex within that Region.</i> <i>Note 2: This State Machine can be referenced by a Submachine State. Connection point reference should be defined in the Submachine State as sources/targets of transitions. The Connection point reference represents a usage of an Entry/Exit Point defined in the State Machine and referenced by the Submachine State.</i> <p>Not Supported</p> <p>Protocol State Machine</p> <p>State Machine Redefinition</p>
States	<p>These State types are supported:</p> <ul style="list-style-type: none"> Simple State: has no internal Vertices or Transitions. Composite State: contains exactly one Region. Orthogonal State: contains multiple Regions. Submachine State: refers to an entire State Machine
Composite State Entry	<p>Supported.</p> <ul style="list-style-type: none"> Default Entry Explicit Entry Shallow History Entry Deep History Entry Entry Point Entry
Substates	<p>Supported.</p> <p>Sub States and Nested Sub States.</p> <p>Entry and Exit semantics where transition covers multiple nested levels of states will obey correct execution of nested behaviors (such as OnEntry and OnExit).</p>
Transitions support	<p>Supported:</p> <ul style="list-style-type: none"> External Transition Local Transition Internal Transition (draw a self Transition and change Transition kind to Internal) Completion Transition and Completion Events Transition Guards Compound Transitions Firing priorities and selection algorithm <p>For further details, refer to the <i>UML Specification</i>.</p>

Trigger and Events	<p>An Executable State Machine supports event handling only for Signals.</p> <p>To use Call, Timing or Change Event types you need to define an outside mechanism to generate signals based on these events.</p>
Signal	<p>Supported.</p> <p>Attributes can be defined in Signals; the value of the attributes can be used as event arguments in Transition Guards and Effects.</p> <p>For example, this is the code set in the effect of a transition in C++:</p> <pre>if(signal->signalEnum == ENUM_SIGNAL2) { int xVal = ((Signal2*)signal)->myVal; }</pre> <p>Signal2 is generated as this code:</p> <pre>class Signal2 : public Signal{ public: Signal2(){}; Signal2(std::vector<String>& lstArguments); int myVal; };</pre> <p>Note: Further details can be found by generating an Executable State Machine and referring to the generated 'EventProxy' file.</p>
Initial	<p>Supported.</p> <p>An initial Pseudostate represents a starting point for a Region; It is the source for at most one Transition; There can be at most one initial Vertex in a Region.</p>
Regions	<p>Supported.</p> <p>Default Activation & Explicit Activation:</p> <p>Transitions terminate on the containing state:</p> <ul style="list-style-type: none"> • If initial Pseudostate is defined in the Region: Default activation; • If no initial Pseudostate is defined, region will remain inactive and the containing state is treated as a Simple state. • Transition terminate on one of the region's contained vertices: Explicit activation. This will result in the default activation of all of its orthogonal Regions, unless those Regions are also entered explicitly (multiple orthogonal Regions can be entered explicitly in parallel through Transitions originating from the same fork Pseudostate). <p>For example, if there are 3 regions defined for a Orthogonal State, if RegionA and RegionB have initial Pseudostate, then RegionC is explicitly activated, default Activation applies to RegionA and RegionB; the containing state will have 3 active regions.</p>
Choice	<p>Supported.</p> <p>Guard Constraints on all outgoing Transitions are evaluated dynamically, when the compound transition traversal reaches this Pseudostate.</p>
Junction	<p>Supported.</p> <p>Static conditional branch: guard constraints are evaluated before any compound transition is executed.</p>

Fork / Join	Supported. Non-threaded, each active region move one step alternatively based on completion event pool mechanism.
EntryPoint / ExitPoint Nodes	Supported. Non-threaded for orthogonal state or orthogonal statemachine; each active region move one step alternatively based on completion event pool mechanism.
History Nodes	Supported. DeepHistory: represents the most recent active state configuration of its owning State. ShallowHistory: represents the most recent active substate of its containing State, but not the substates of that substate.
Deferred Events	Supported. Draw a self Transition and change Transition kind to Internal. Type 'defer();' in the 'Effect' field for the transition.
Connection Point References	Supported. A connection point reference represents a usage (as part of a submachine State) of an Entry/Exit Point defined in the StateMachine referenced by the Submachine State. Connection point references of a submachine State can be used as sources/targets of Transitions. They represent entries into or exits out of the StateMachine referenced by the Submachine State.
State behaviors	Supported. State entry, doActivity and exit behavior can be defined as Operations on a state. The code that will be used for each behavior is entered into the 'initial code' field by default. Note that this could be changed to 'Behavior' field via customization of the generation template. The doActivity behavior generated will be run to completion before proceeding. The code is not concurrent as to other entry behavior; the doActivity behavior is implemented as execute in sequence after entry behavior.

References to Behaviors within other Contexts/Classes

If the Submachine State references behavior elements outside the current context or Class, you must add an `<<import>>` connector from the current context Class to the container context Class. For example:

Submachine State S1 in Class1 refers to StateMachine ST2 in Class2

Therefore, we add an `<<import>>` connector from Class1 to Class2 in order for Executable StateMachine code generation to generate code correctly for Submachine State S1. (On Class 1, click on the Quick Linker arrow and drag to Class 2, then select 'Import' from the menu of connector types.)

Reusing Executable Statemachine Artifacts

You can create multiple models or versions of a component using a single executable Artifact. An Artifact representing a resistor, for example, could be re-used to create both a foil resistor and a wire wound resistor. This is likely the case for similar objects who although represented by the same classifier, typically exhibit different run states. A property named 'resistorType' taking the value 'wire' rather than 'foil' might be all that is required from a modeling point of view. The

same State Machines can then be re-used to test behavioral changes that might result due to variance in run-state. This is the procedure:

Step	Action
Create or open component diagram	Open a component diagram to work on. This might be the diagram that contains your original artifact.
Select the Executable State Machine to copy	Now find the original Executable Statemachine Artifact in the Project Browser .
Create the New Component	<p>Whilst holding the Ctrl key, drag the original artifact on to your diagram. You will be prompted with two questions.</p> <p>The answer to the first is Object and to the second All. Rename the artifact to differentiate it from the original and then proceed to alter its property values.</p>

Executable StateMachine Artifact

An Executable StateMachine Artifact is key to generating StateMachines that can interact with each other. It specifies the objects that will be involved in a simulation, their state and how they connect.

Creating the Properties of an Executable StateMachine

Each Executable StateMachine scenario involves one or more StateMachines. The StateMachines included are specified by UML Property elements; each Property will have a UML Classifier (Class) that determines the StateMachine(s) included for that type. Multiple types included as multiple Properties can end up including many StateMachines, which are all created in code and initialized on execution.

Action	Description
Drop a Class from the Project Browser on to the <<Executable Statemachine>> Artifact	<p>The easiest way to define properties on an Executable StateMachine is to drop the Class onto the Executable StateMachine from the Project Browser. On the dialog that is shown, select the option to create a Property. You can then specify a name describing how the Executable StateMachine will refer to this property.</p> <p>Note: Depending on your options, you might have to hold down the Ctrl key to choose to create a property. This behavior can be changed at any time using the 'Hold Ctrl to Show this dialog' checkbox.</p>
Use and Connect Multiple UML Properties	<p>An Executable StateMachine describes the interaction of multiple StateMachines. These can be different instances of the same StateMachine, different StateMachines for the same instance, or completely different StateMachines from different base types. To create multiple properties that will use the same StateMachine, drop the same Class onto the Artifact multiple times. To use different types, drop different Classes from the Project Browser as required.</p>

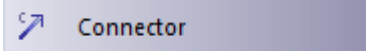
Defining the initial state for properties

The StateMachines run by an Executable StateMachine will all run in the context of their own Class instance. An Executable StateMachine allows you to define the initial state of each instance by assigning property values to various Class attributes. For example you might specify a Player's age, height, weight or similar if these properties have relevance to the scenario being run. By doing this it is possible to set up detailed initial conditions that will influence how the scenario plays out.

Action	Description
Set Property Values dialog	<p>The dialog for assigning property values can be opened by right-clicking on a Property and selecting 'Features & Properties Set Property Values', or by using the keyboard shortcut Ctrl+Shift+R.</p>
Assign a value	<p>The 'Set Property Values' dialog allows you to define values for any attribute defined in the original Class. To do this, select the variable, set the operator to '=' and enter the required value.</p>

Defining relationships between properties

In addition to describing the values to assign to variables owned by each property, an Executable StateMachine allows you to define how each property can reference others based on the Class model that they are instances of.

Action	Description
Create a connector	<p>Connect multiple properties using the Connector relationship from the Composite toolbox.</p>  <p>Alternatively, use the Quicklinker to create a relationship between two Properties and select 'Connector' as the relationship type.</p>
Map to Class model	<p>Once a connector exists between two properties, you can map it back to the Association it represents in the Class model. To do this, select the connector and use the keyboard shortcut Ctrl+L. This shows the 'Choose an Association' dialog. This allows the generated State Machine to send signals to the instance filling the role specified in the relationship during execution.</p>

Code Generation for Executable State Machines

The code generated for an Executable State Machine is based on its language property. This might be Java, C, C++, C# or Javascript. Regardless, Enterprise Architect generates the appropriate code, which is immediately ready to compile and run. There are no manual interventions necessary before you run it. In fact after the initial generation, any Executable State Machine can be generated, built and executed at the click of a button.

Generating Code

The context menu of an Executable State Machine provides special code generation commands for generating the State Machine. The first option is to generate the code, which will display this dialog.

Executable State Machine Code Generation

Artifact: PairedTurbineTest Language: C#

Project output directory: C:\turbine\PairedTurbines













Important: Please note, all files in this folder will be deleted.

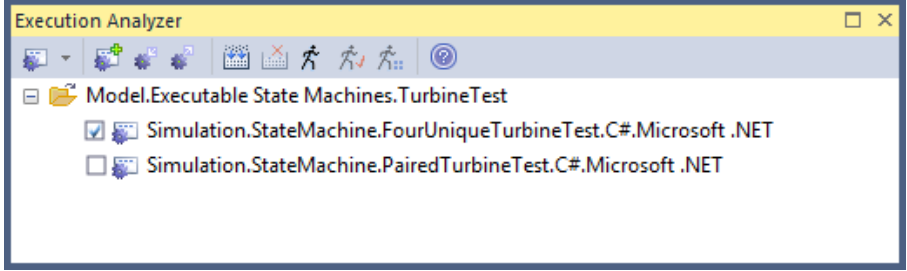
Project build environment

Local Path: Location of Microsoft .NET Framework directory.
 CS_HOME C:\Windows\Microsoft.NET\Framework\v3.5

Compilation Options: ☒ 32bit ☐ 64bit

Generate Cancel Help

Feature	Description																				
Set Generation Path	The generation dialog allows you to enter a target directory to which all source files will be generated.																				
Set Framework Path	Each supported language provides an option to define the path to the target frameworks that are required to compile and run the generated code. For more information, see the <i>Languages Supported</i> section.																				
Files Created	<p>Regardless of the selected language, the code generation of an execution analyzer will create a similar set of files as shown here for C#.</p> <table><tr><td></td><td>ConsoleManager.cs</td><td>12/06/2015 10:56 ...</td><td>C# Language File</td><td>2 KB</td></tr><tr><td></td><td>ContextManager.cs</td><td>12/06/2015 10:56 ...</td><td>C# Language File</td><td>24 KB</td></tr><tr><td></td><td>EventProxy.cs</td><td>12/06/2015 10:56 ...</td><td>C# Language File</td><td>2 KB</td></tr><tr><td></td><td>SimulationManager.cs</td><td>12/06/2015 10:56 ...</td><td>C# Language File</td><td>4 KB</td></tr></table>		ConsoleManager.cs	12/06/2015 10:56 ...	C# Language File	2 KB		ContextManager.cs	12/06/2015 10:56 ...	C# Language File	24 KB		EventProxy.cs	12/06/2015 10:56 ...	C# Language File	2 KB		SimulationManager.cs	12/06/2015 10:56 ...	C# Language File	4 KB
	ConsoleManager.cs	12/06/2015 10:56 ...	C# Language File	2 KB																	
	ContextManager.cs	12/06/2015 10:56 ...	C# Language File	24 KB																	
	EventProxy.cs	12/06/2015 10:56 ...	C# Language File	2 KB																	
	SimulationManager.cs	12/06/2015 10:56 ...	C# Language File	4 KB																	

	In addition, there are also files generated for each Class used by the Executable State Machine.
Execution Analyzer Scripts	<p>Each Executable State Machine that is generated will also generate an Execution Analyzer script. This allows the generated code to be compiled either from the Execution Analyzer window or using the Executable State Machine context menu.</p> 
Generation Output	Where generating progress messages, warnings or errors are displayed in the 'Executable State Machine Output' page of the System Output window.

Compiling Code

The code generated by an Executable State Machine can be compiled by Enterprise Architect in one of three ways.

Method	Description
Execution Analyzer Script	The generated execution analyzer script includes a command to build the source code. This means that when it is active you can compile directly using the execution analyzer script, including using the built-in shortcut key Ctrl+Shift+F12 .
Compile Context Menu Command	The 'Code Generation' context menu of an Executable State Machine provides a 'Compile' command, allowing you to compile the code that already exists in the target directory. This can be used directly after calling 'Generate', or if you have modified the generated code in any way.
Generate, Build and Run Command	The 'Code Generation' context menu of an Executable State Machine provides a shortcut to allow changes made to a State Machine to be quickly tested. This command updates the code before compiling it, and then starts the simulation as discussed in detail in a later topic.
Compiler Output	When compiling, all output from the compiler is shown on the Build page of the System Output window. This also allows you to double-click on any compiler errors to open a source editor to the appropriate line.

Leveraging existing code

Executable State Machines generated and executed by Enterprise Architect can leverage existing code for which no Class model exists. To do this you would create an abstract Class element naming only the operations to call in the external codebase. You would then create a generalization between this interface and the State Machine Class, adding the required linkages manually in the Analyzer Script. For Java you might add .jar files to the Class path. For native code you might add a .dll to the linkage.

Languages Supported

An Executable Statemachine supports code generation for these platform languages:

Language Platform	Languages
Microsoft Native	<ul style="list-style-type: none">• C• C++
Microsoft .NET	<ul style="list-style-type: none">• C#
Scripting	<ul style="list-style-type: none">• JavaScript
Oracle Java	<ul style="list-style-type: none">• Java

Debugging Execution of Executable State Machines

Creation of Executable State Machines provides benefits even after the generation of code. By using the execution analyzer, Enterprise Architect is able to connect to the generated code. As a result you are able to visually debug and verify the correct behavior of the code; the exact same code generated from your State Machines, demonstrated by the simulation and ultimately incorporated in a real world system.

Debugging a State Machine

Being able to debug an Executable State Machine gives additional benefits. These benefits allow you to:

- Interrupt the execution of the simulation and all executing State Machines.
- View the raw state of each State Machine instance involved in the simulation.
- View the source code and **Call Stack** at any point in time.
- Trace additional information about the execution state through the placement of tracepoints on lines of source code.
- Control the execution through use of actionpoints and breakpoints (break on error for example).
- Diagnose changes in behavior, due to either code or modeling changes.

If you have generated, built and run an Executable State Machine successfully, you can debug it! The Analyzer Script created during the generation process is already configured to provide debugging. To start debugging, simply start running the Executable State Machine using the **Simulation Control**. Depending on the nature of the behavior being debugged, we would probably set some breakpoints first.

Breaking execution at a state transition

Like any debugger we can use breakpoints to examine the executing statemachine at a point in code. Locate a class of interest in either the diagram or project browser and press **F12** to view the source code. It is easy to locate the code for state transitions from the naming conventions used during generation. If you wish to break at a particular transition, locate the transition function in the editor and place a breakpoint marker by clicking in the left margin at a line within the function. When you run the executable statemachine, the debugger will halt at this transition and you will be able to view the raw state of variables for any state machines involved.

Breaking execution conditionally

Each breakpoint can take a condition and a trace statement. When the breakpoint is encountered and the condition evaluates to **True** the execution will halt. Otherwise the execution will continue as normal. You compose the condition using the names of the raw variables and comparing them using the standard equality, operands: `<` `>` `=` `>=` `<=`. eg: `(this.m_nCount > 100)` and `(this.m_ntype == 1)`

To add a condition to a breakpoint you have set, right-click the breakpoint and select properties. By clicking the breakpoint while holding the CTRL key, the properties can be quickly edited.

Tracing auxiliary information

It is possible to trace information from within the statemachine itself using the TRACE clause; in an *effect* for example. Debugging also provides trace features known as Tracepoints. These are simply breakpoints that, instead of breaking, print trace statements when they are encountered. The output is displayed in the **Simulation Control** window. They can be used as a diagnostic aid to show / prove the sequence of events and the order in which instances change state.

Viewing the callstack

Whenever a breakpoint is encountered, the callstack is available. This is available from the Analyzer Menu. Use this to determine the order in which the execution is taking place.

Execution and Simulation of Executable State Machines

One of the many features of Enterprise Architect is its ability to perform simulations. An Executable State Machine generated and built in Enterprise Architect can hook into the Simulation feature to visually demonstrate the live execution of the State Machine Artifact.

Starting a simulation

The **Simulation Control** toolbar provides a **Search button** that allows you to select the Executable Statemachine Artifact to run. The control also maintains a drop-down list of the most recent Executable State Machines for you to choose from. You can also use the context menu on an Executable Statemachine Artifact itself to initiate the simulation.

Controlling speed

The **Simulation Control** provides a speed setting. You can use this to adjust the rate at which the simulation executes. The speed is represented as a value between 0 and 100. (*Higher value is faster*). A value of zero will cause the simulation to halt after every step (*requires using the toolbar controls to manually step the simulation*).

Notation for active states

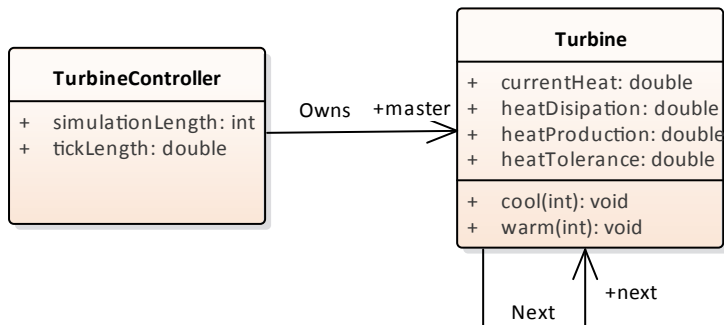
As the executable statemachine executes, the relevant statemachine diagrams are displayed. The display is updated at the end of every step-to-completion cycle. You will notice that only the active state for the instance completing a step is highlighted. The other states remain dimmed.

It is easy to identify which instance is in which state as the states will be labeled with the name of any instance currently in that particular state. If more than one artifact property of the same type share the same state, the state will have two labels naming each property.

Example Executable State Machine

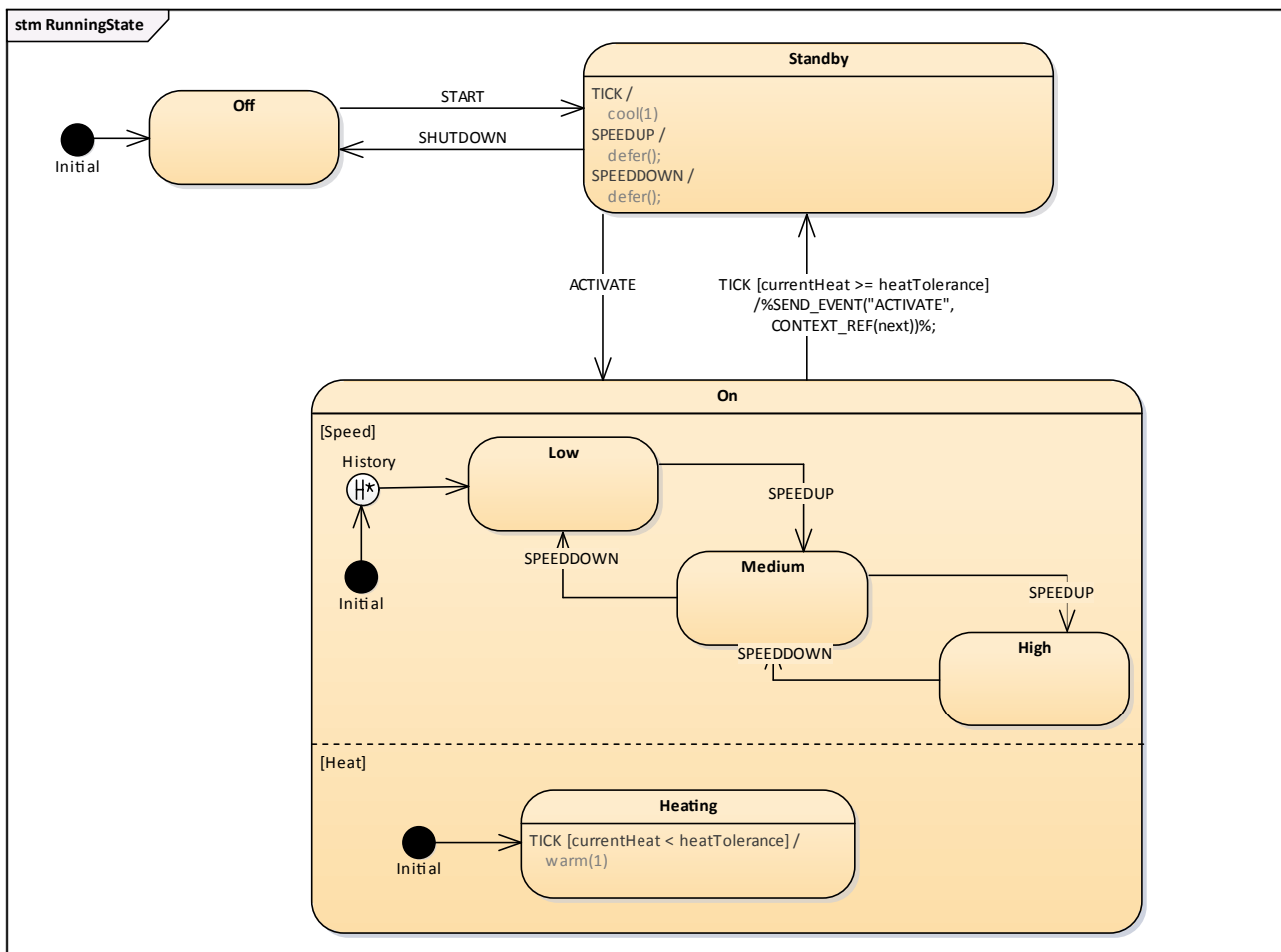
Example Class Model

This image shows a sample Class model that is used by the State Machines to follow.

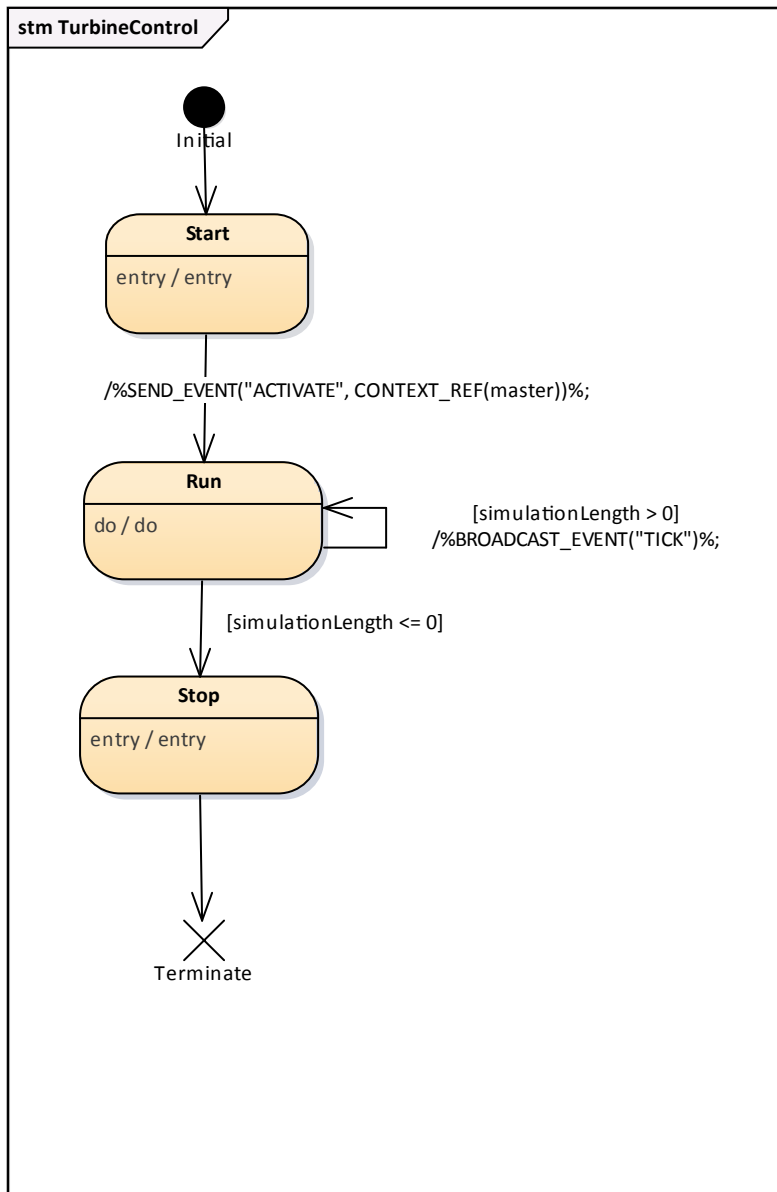


Example State Machines

These two diagrams show the definitions of two State Machines. The first references another State Machine of the same type, while the second drives any instances of the first that exist.

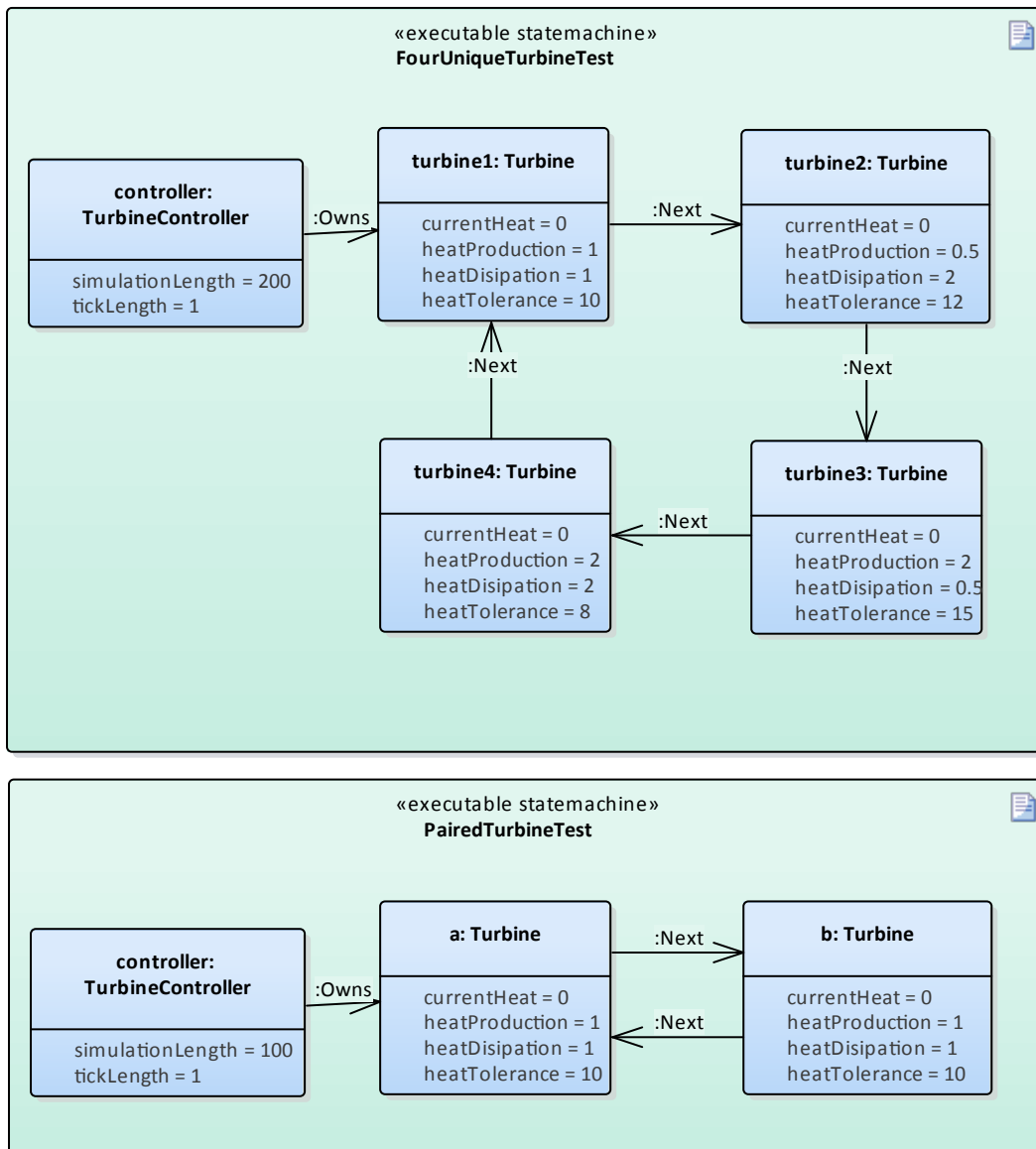


And the top level controller.



Example Artifacts

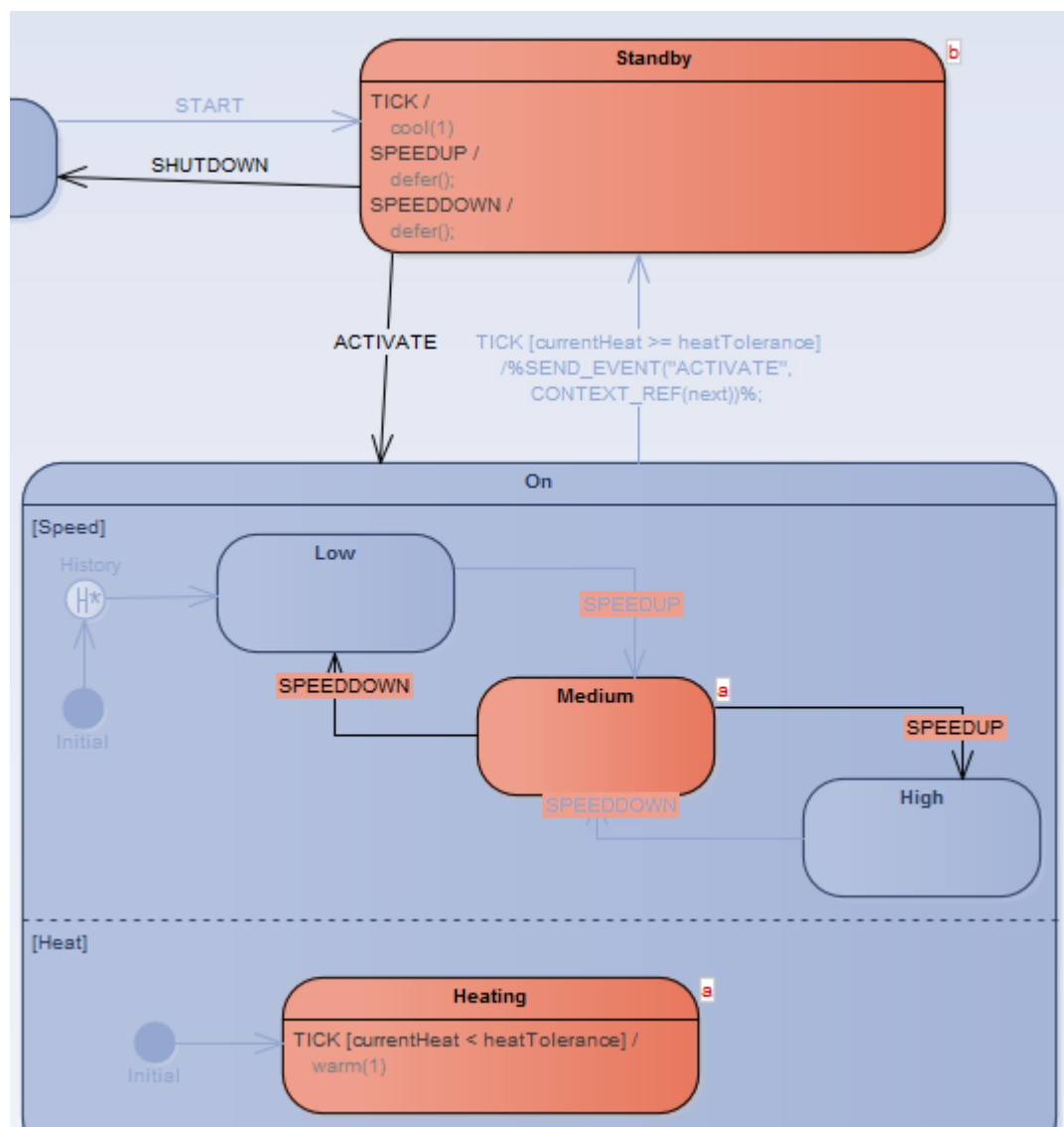
From the example diagrams above, we can create Executable State Machines as shown here.



Note how property values have been set for each property, and the links between them identify the relationships that exist in the Class model.

Simulation Results

When running, Enterprise Architect will highlight the currently active states in any statemachines. Where multiple instances of a statemachine exist, it will also show the names of each instance in that state.



Deferred Event Pattern

Enterprise Architect supports the Deferred Event pattern.

To create a Deferred Event in a State:

1. Create a self transition for the State.
2. Change the 'kind' of the transition to 'internal'.
3. Specify the Trigger to be the event you want to defer.
4. In the 'Effect' field, type 'defer();'.

To Simulate:

From the main menu, select 'Analyzer | Simulator | Open Simulator and Simulator Event Window'.

The **Simulator Events window** allows you to trigger events by double clicking on a trigger in the 'Waiting Triggers' column.

The **Simulation window** shows the execution in text. You can type 'dump' in the Simulator command line to show how many events are deferred in the queue; it might resemble this:

```
[24850060]   Event Pool: [NEW,NEW,NEW,NEW,NEW,]
```

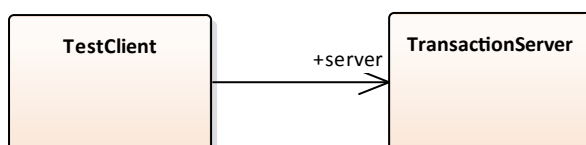
Deferred Event Example

This example shows a model using deferred events, along with the **Simulation Events window** showing all available Events.

We firstly setup the contexts (the class elements containing the state machines), simulate in a simple context and raise the event from outside of it; then simulate in a client-server contexts with the send event mechanism.

Create Context and Statemachine

Create the server context



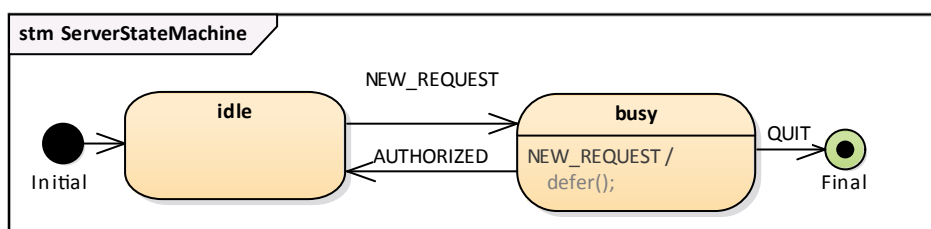
Create a class diagram,

a class element *TransactionServer*, add to which a statemachine *ServerStateMachine*.

a class element *TestClient*, add to which a statemachine *ClientStateMachine*.

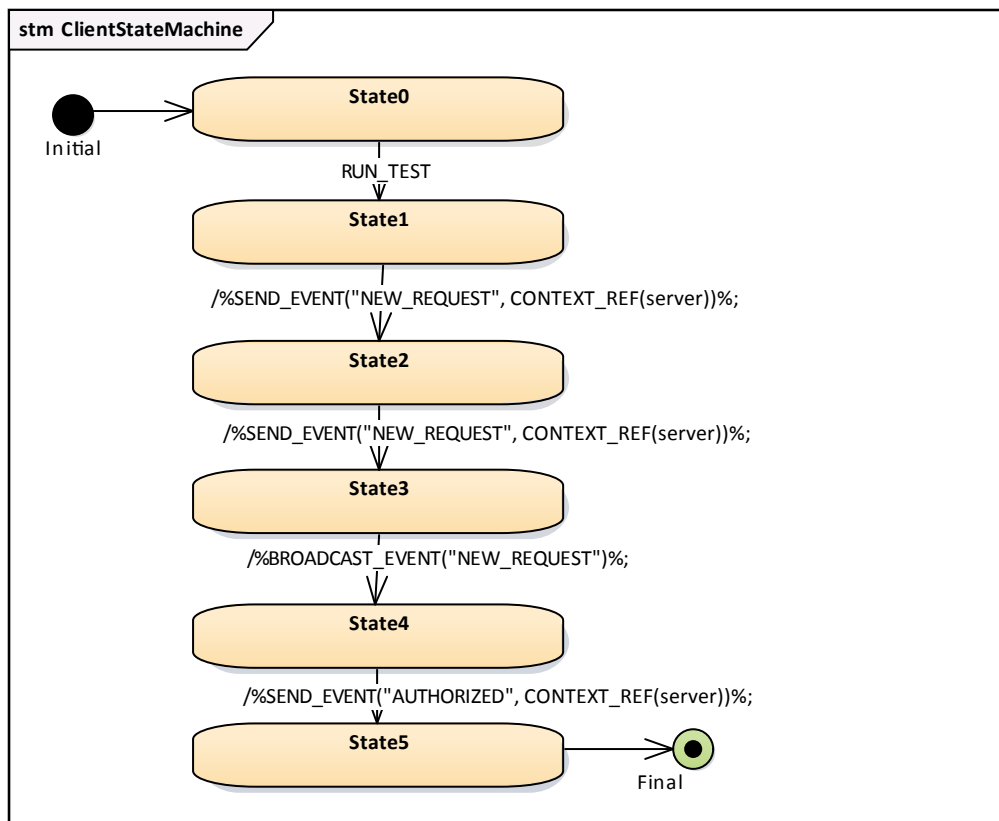
an association from *TestClient* to *TransactionServer*, target role named *server*

Modeling for *ServerStateMachine*



- Put an Initial Node *Initial* onto the statemachine diagram, transaction to
- a state *idle*, transition(with event NEW_REQUEST as trigger) to
- a state *busy*,
 transition (with event QUIT as trigger) to a Final State *Final*
 transition (with event AUTHORIZED as trigger) to *idle*
 transition (with event NEW_REQUEST as trigger, defer(); as effect) to *busy*

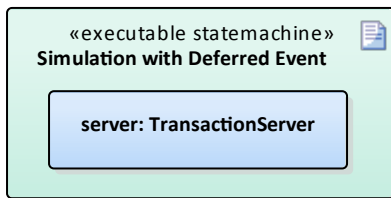
Modeling for *ClientStateMachine*



- Put an Initial Node *Initial* onto the statemachine diagram, transaction to
- a state *State0*, transition(with event RUN_TEST as trigger) to
- a state *State1*, transition(effect: %SEND_EVENT("NEW_REQUEST", CONTEXT_REF(server));) to
- a state *State2*, transition(effect: %SEND_EVENT("NEW_REQUEST", CONTEXT_REF(server));) to
- a state *State3*, transition(effect: %BROADCAST_EVENT("NEW_REQUEST");) to
- a state *State4*, transition(effect: %SEND_EVENT("AUTHORIZED", CONTEXT_REF(server));) to
- a state *State5*, transition to
- a Final State *Final*

Simulation in a simple context

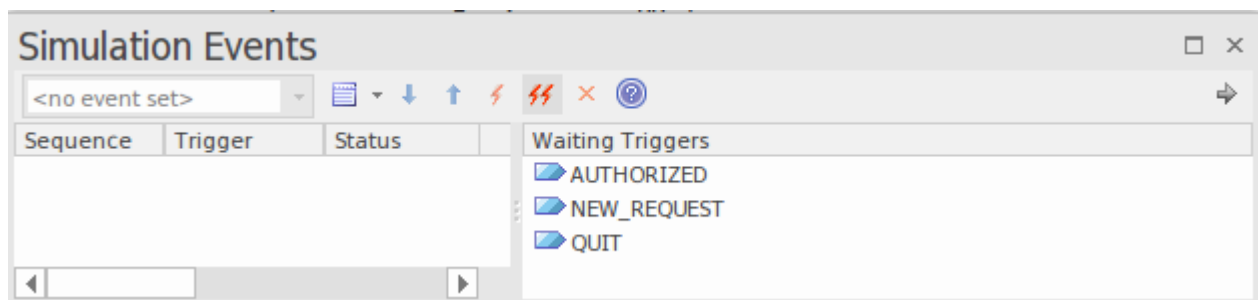
Create the Simulation Artifact



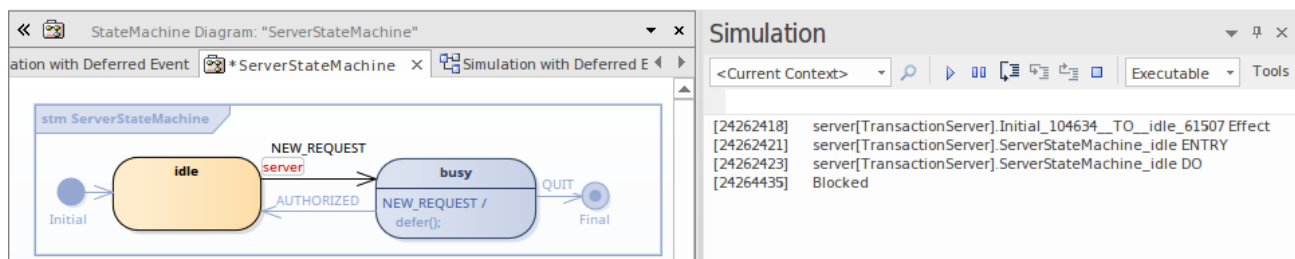
- Create an Executable StateMachine artifact with name *Simulation with Deferred Event* and Language set to *JavaScript*
- Enlarge it; **Ctrl** + Drag TransactionServer, paste as a property with name *server*

Run Simulation

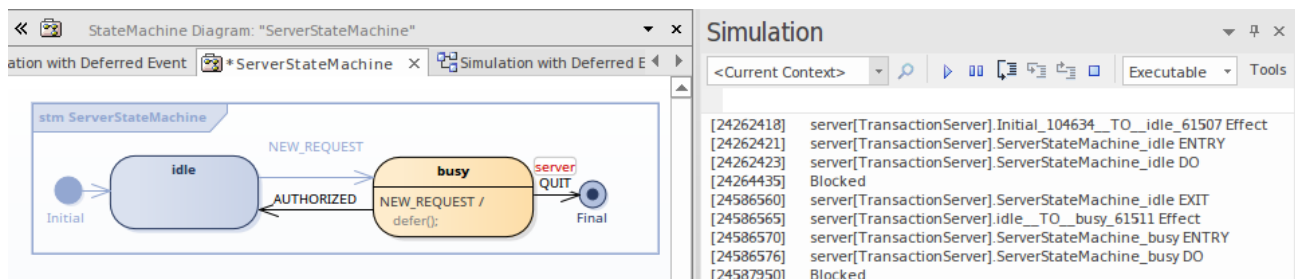
- Select the artifact | select Ribbon Simulate | Build & Run | Specify a directory for you code (NOTE: all the files in the directory will be deleted before simulation started) | Generate
- Open Simulation Event window by Ribbon Simulate | Triggers



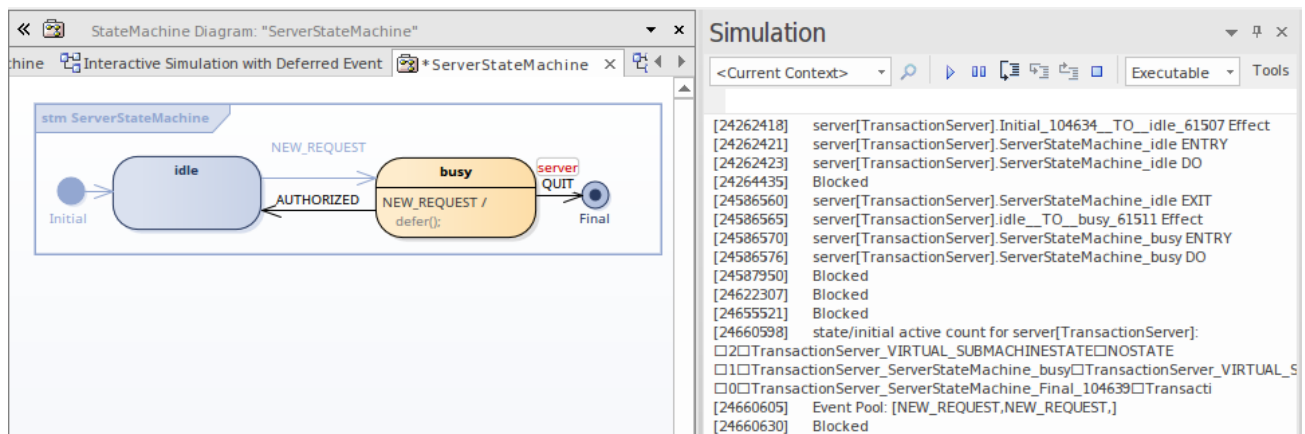
Once simulation started, *idle* will be the active state.



- Trigger NEW_REQUEST by double click it in the event window; *idle* will be exited and *busy* is activated.



- Trigger NEW_REQUEST by double click it in the event window again; *busy* stay as activated, an instance of NEW_REQUEST is appended in the event pool.
- Trigger NEW_REQUEST by double click it in the event window again; *busy* stay as activated, an instance of NEW_REQUEST is appended in the event pool.
- Type dump in the Simulation command line, you can see that the event pool has two instances of NEW_REQUEST.

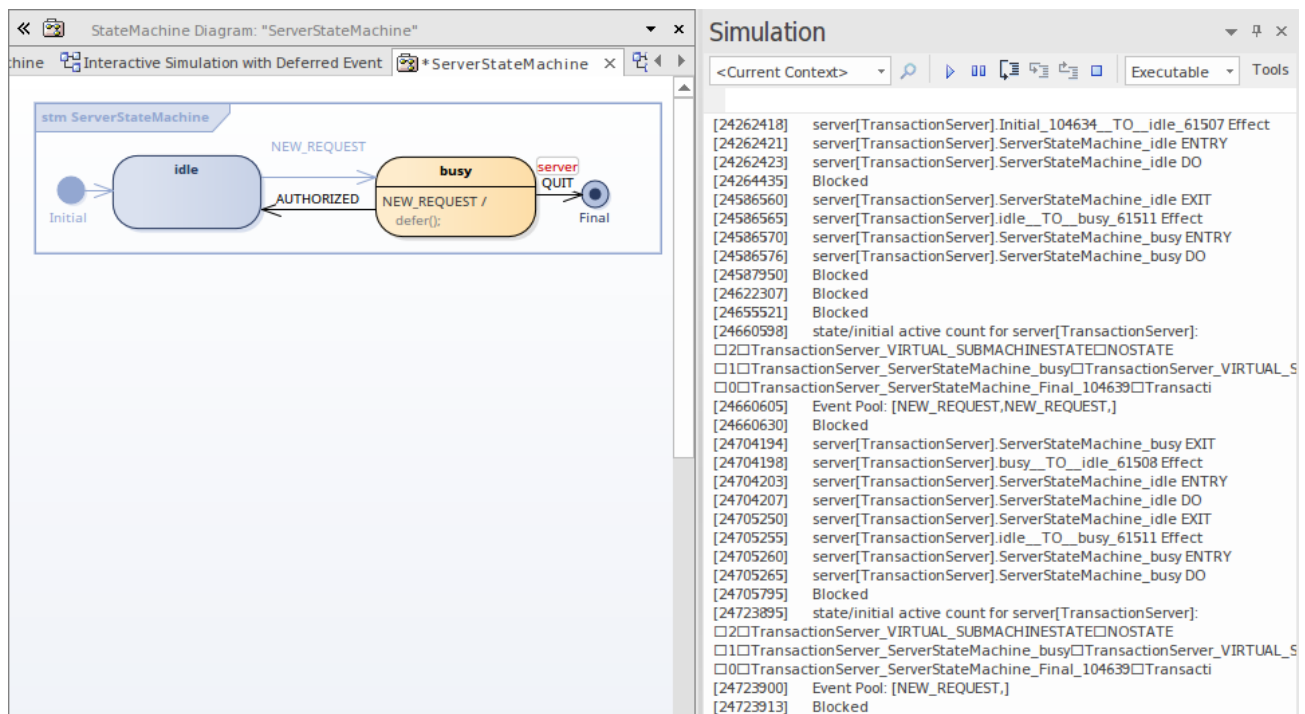


- Trigger AUTHORIZED by double click it in the event window. Then following things will happen:

1. *busy* was exited and *idle* became active
2. a NEW_REQUEST event is retrieved from the pool and *idle* was exited and *busy* became active

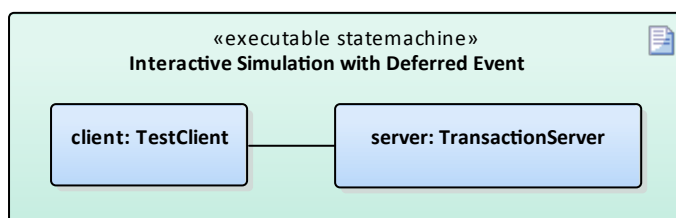
Type "dump" in the simulation command line,

- There is only one instance of NEW_REQUEST in the event pool



Interactive simulation via Send/Broadcast Event

Create the Simulation Artifact

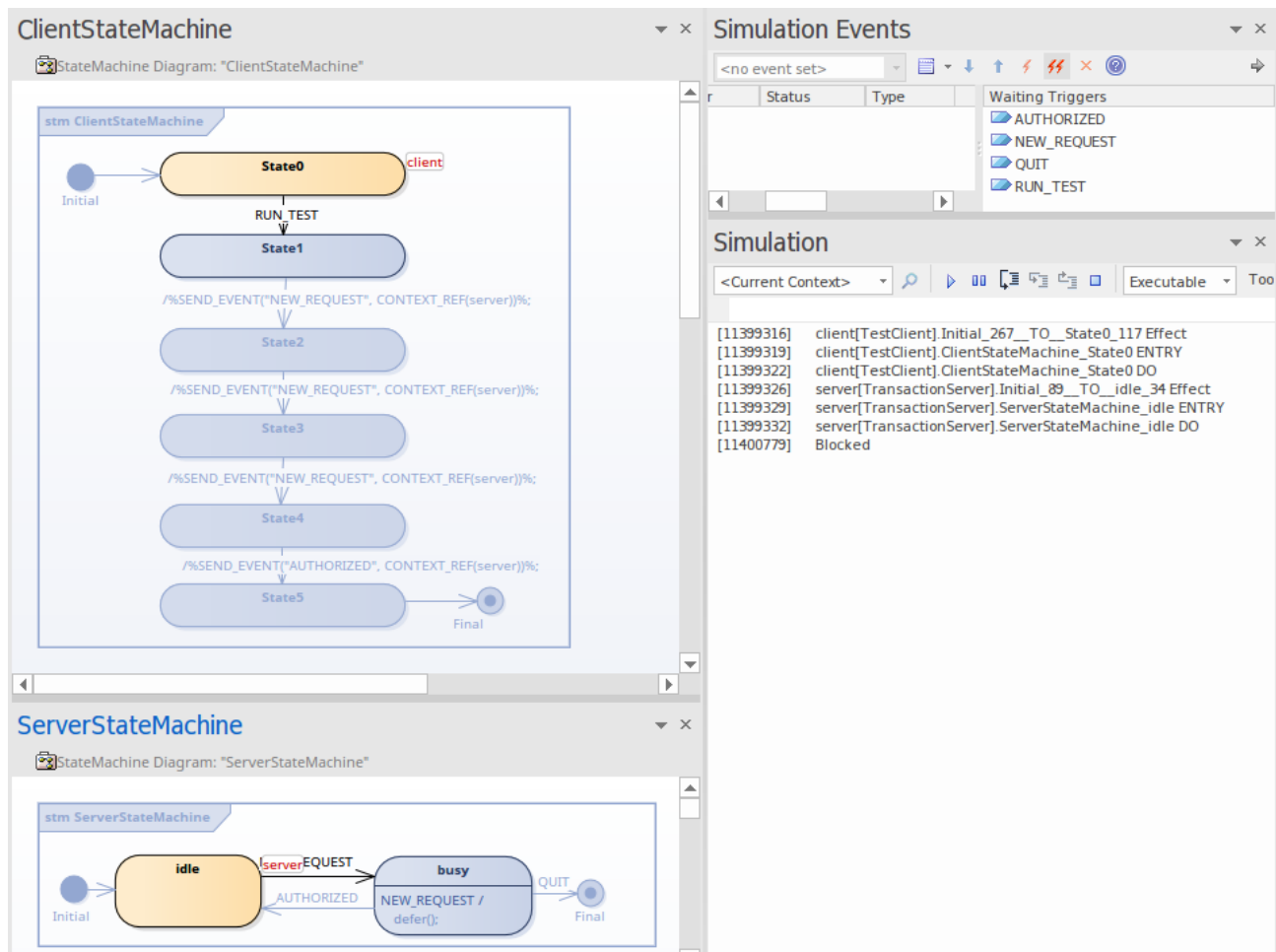


- Create an Executable StateMachine artifact with name *Interactive Simulation with Deferred Event* and Language set to *JavaScript*. Enlarge it
- **Ctrl** + Drag TransactionServer, paste as a property with name *server*
- **Ctrl** + Drag TestClient, paste as a property with name *client*
- Create a connector from *client* to *server*

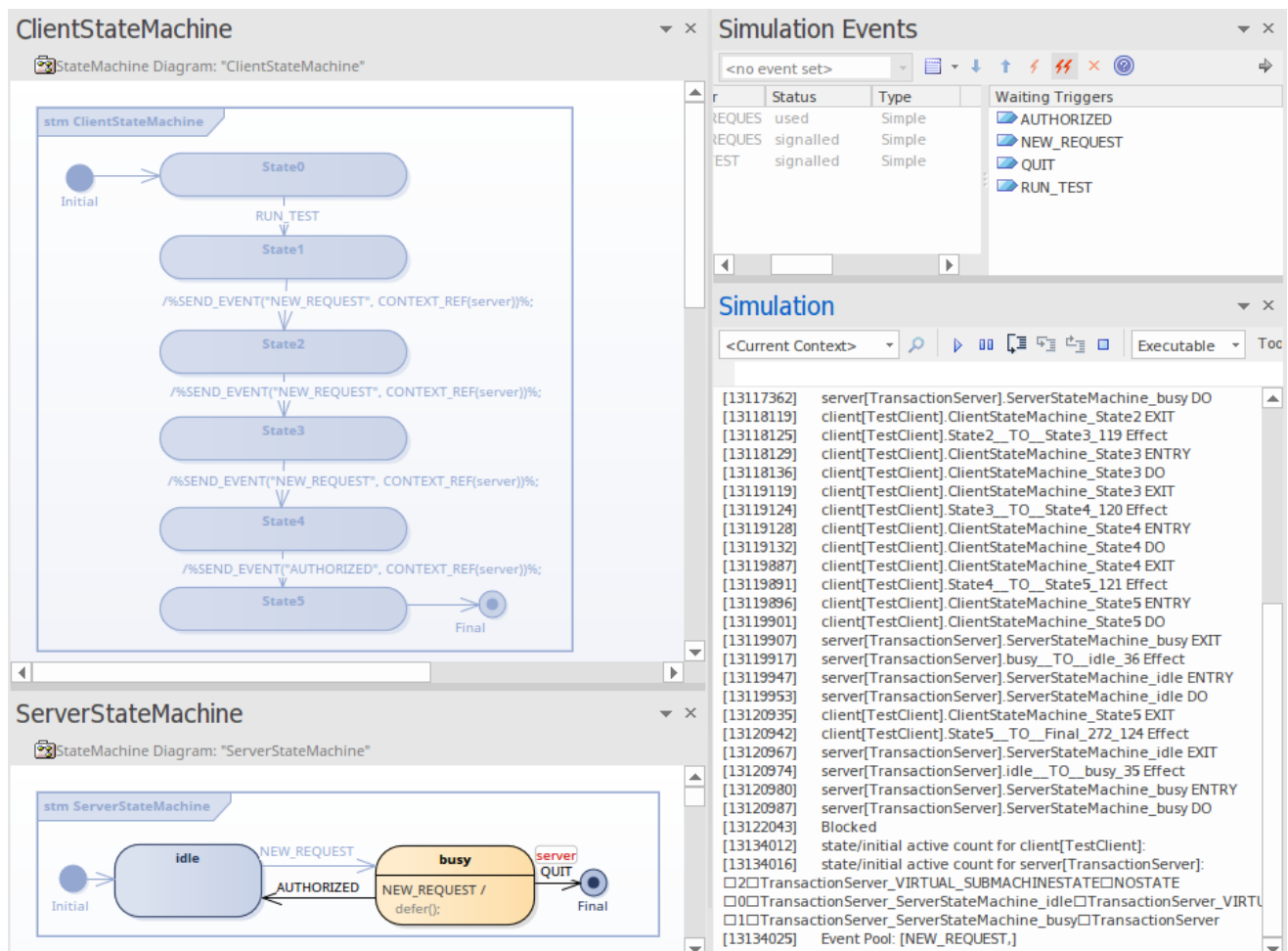
Run Interactive Simulation

Launch the simulation in the same way as the simple context;

Once the simulation started, the *client* is staying at State0, the *server* is staying at *idle*.



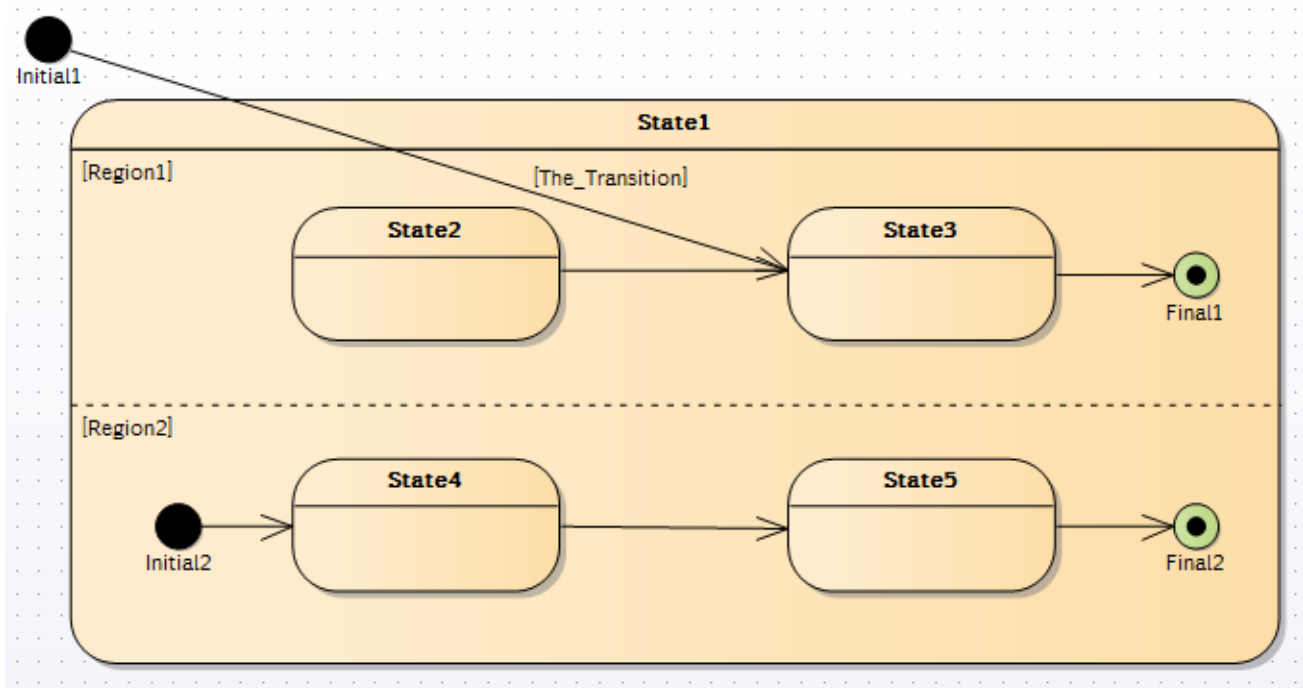
Trigger RUN_TEST by double click it in the event window. Then event NEW_REQUEST will be triggered 3 times (by SEND_EVENT / BROADCAST_EVENT), and AUTHORIZED will be triggered once by SEND_EVENT.



Type "dump" in the simulation command line, there are one instance of NEW_REQUEST left in the event pool. The result match our manually triggering test.

States With Multiple State Regions

A State is activated when there is a transition to the State, or a transition to a State within the State. See the following example:



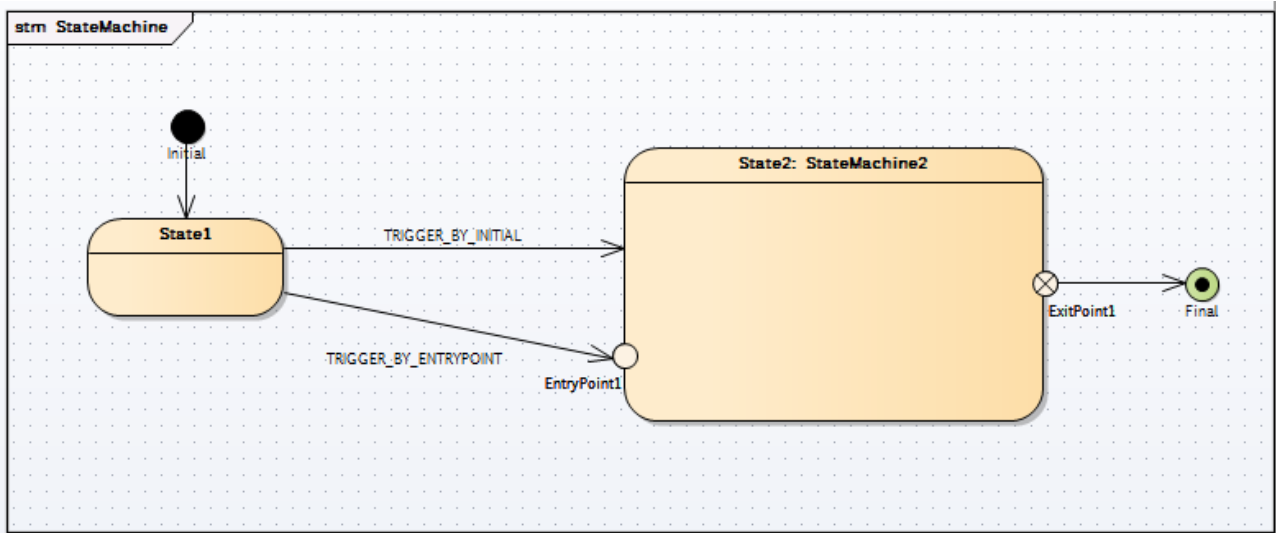
- **State1** is activated because of **The_Transition** going to **State3**.

When a State is activated, all regions within the State which can be activated, will be. A region can be activated if there is an Initial element, or if there is a transition from outside the State into one of the region's States. All activating regions run in no particular order.

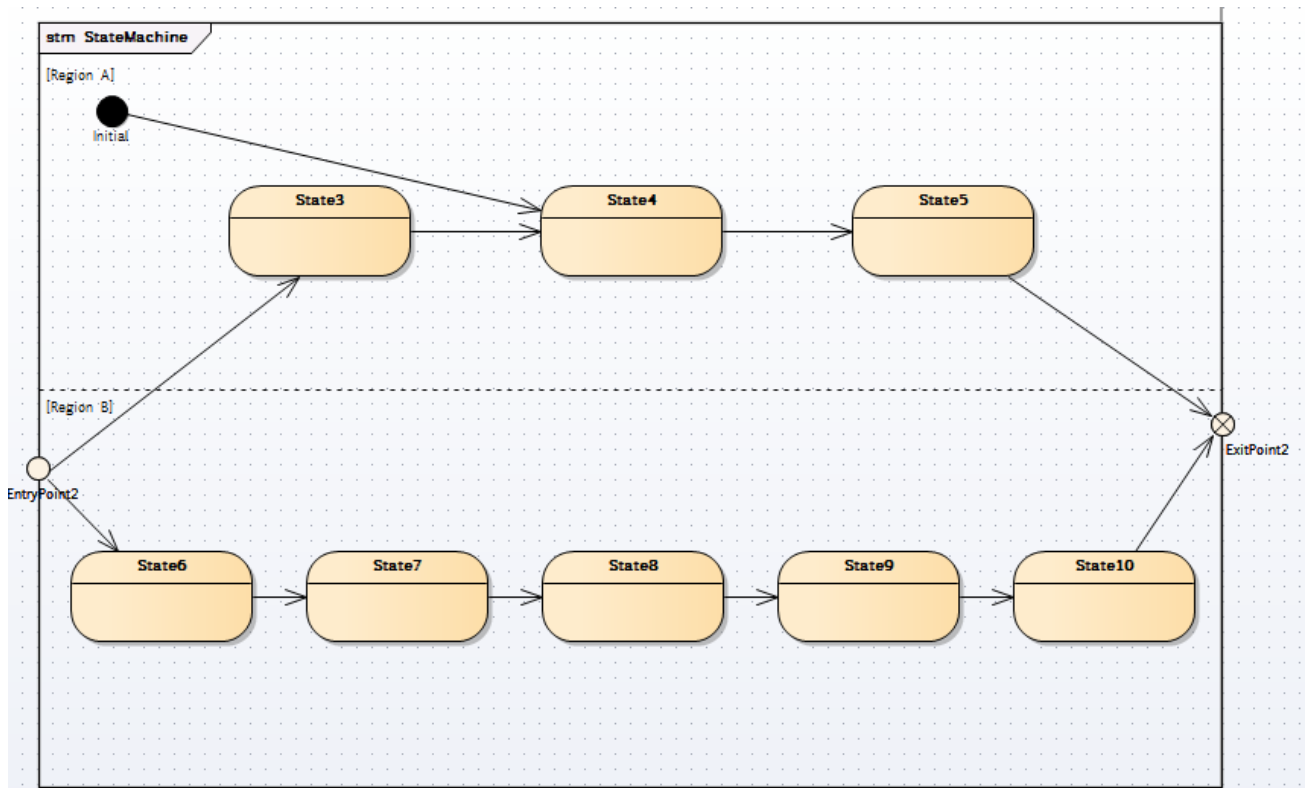
- **Region1** is activated because of **The_Transition** going to **State3**.
- **Region2** is activated because there is an Initial element.

Entry and Exit Points (Connection Point References)

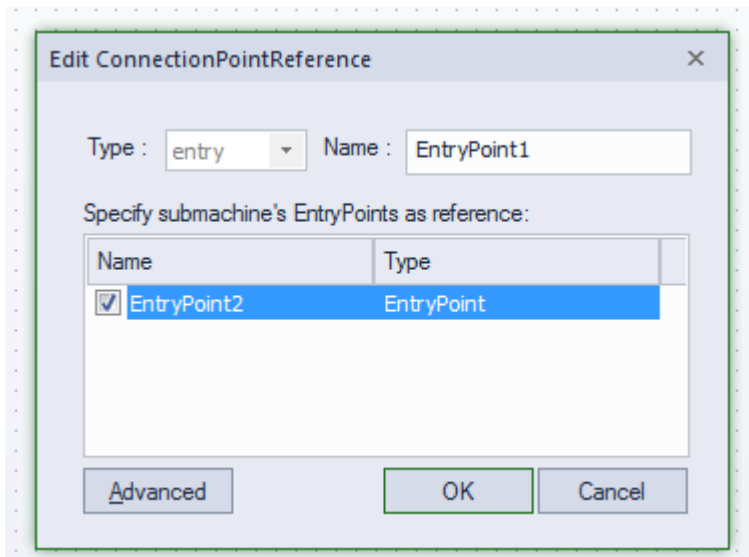
Enterprise Architect provides support for Entry/Exit points, and connection point references. In the below example, State1 has two triggers, **TRIGGER_BY_INITIAL** and **TRIGGER_BY_ENTRYPOINT** leading to State2 and its EntryPoint1.



State2 is typed by another StateMachine (StateMachine2) using **Ctrl+L** when State2 is selected. StateMachine2 is as follows:



To link **EntryPoint1** with **EntryPoint2**, and **ExitPoint1** with **ExitPoint2**, double click on **EntryPoint1** and tick **EntryPoint2** and the same for the exit point:



If we ran this example and triggered **TRIGGER_BY_INITIAL**, Region A will activate and hit Initial -> State4 -> State5 -> ExitPoint2.

If we ran this example and triggered **TRIGGER_BY_ENTRYPOINT**, both Region A and B will activate and hit all States (3 to 10).

