ENTERPRISE ARCHITECT

User Guide Series

Executable StateMachines

Author: Sparx Systems

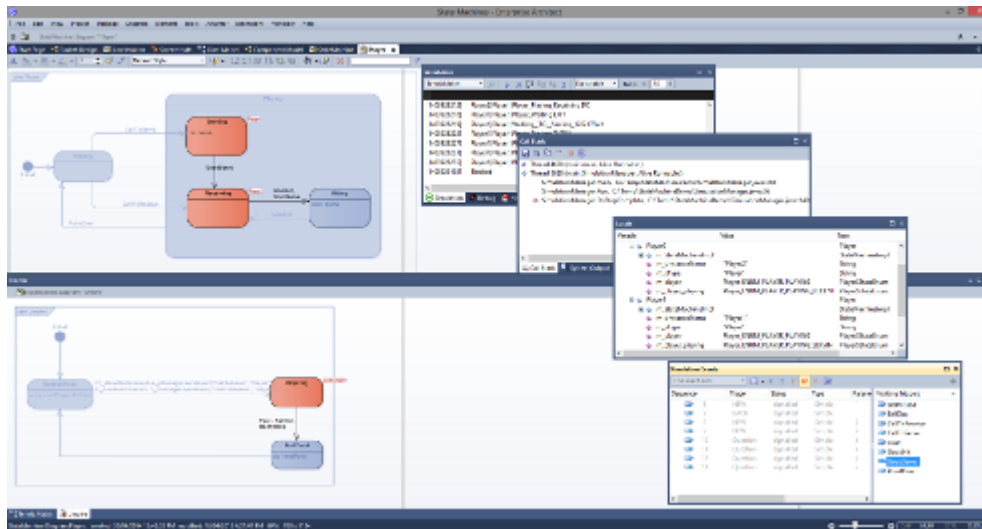Date: 2022-10-03

Version: 16.0

CREATED WITH ENTERPRISE ARCHITECT

Table of Contents

Executable StateMachines

Executable StateMachines provide a 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 StateMachines provide a complete language-specific implementation that can form the behavioral 'engine' for multiple software products on multiple platforms. Visualization of the execution is based on a seamless integration 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 introduce you to the basics of modeling Executable StateMachines and help you to understand how to generate and simulate them.

The creation and use of Executable StateMachines, and generating code from them, are supported by the Unified and Ultimate Editions of Enterprise Architect.

Overview of Building and Executing StateMachines

Building and using Executable StateMachines is quite straightforward, 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 StateMachines.

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 StateMachine 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, it is 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 Compile	Whether you use JavaScript, C++, Java or C#, Enterprise Architect's engineering capabilities provide you with an effective tool, allowing you to regenerate the executable at any time, and without the loss of any customized code you might 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 StateMachines	So how do we see how these StateMachines 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#, Enterprise Architect will take care of creating the runtime, the hosting of your model, the execution of its behaviors and the rendition of all StateMachines.
Visualize StateMachines	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 StateMachines	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 ExecutableStateMachine code generation. Debugging provides many benefits, one of which might be to verify / corroborate the code attached to behaviors in a StateMachine to ensure it is actually reflected in the executing process.

Modeling Executable StateMachines

Most of the work required to model an Executable StateMachine is standard UML-based modeling of Classes and State models, although 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 StateMachine instance or scenario. The Artifact is used to specify details such as:

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

Basic Modeling Tools and Objects for Executable StateMachines

These are the primary modeling elements used when building Executable StateMachines.

Element type	Description
Classes and Class Diagrams	Classes define the object types that are relevant to the StateMachine(s) being modeled. For example, in a simple Tennis Match scenario you might define a Class for each of a Player, a Match, a Hit and an Umpire. Each will have its own StateMachine(s) and at runtime will be represented by object instances for each involved entity. See the *UML Modeling Guide* for more information on Classes and Class diagrams.
StateMachines	For each Class you define that will have dynamic behavior within a scenario, you will typically define one or more UML StateMachines. Each StateMachine will determine the legal state-based behavior appropriate for one aspect of the owning Class. For example, it is possible to have a StateMachine 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 StateMachines will be initialized and started when the StateMachine scenario begins execution.
Executable StateMachine Artifact	This stereotyped Artifact is the core element used to specify the participants, configuration and starting conditions for an Executable StateMachine. 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. 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 StateMachines.

StateMachine Constructs Supported

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

Construct	Description
StateMachines	- Simple StateMachine: the StateMachine has one Region - Orthogonal StateMachine: the StateMachine contains multiple Regions

	Top level Region (owned by StateMachine) activation semantics: **Default Activation:** When the StateMachine starts executing. **Entry Point Entry:** Transitions from Entry Point to vertices in the contained Regions. • *Note 1: In each Region of the StateMachine owning the Entry Point, there is at most a single Transition from the entry point to a Vertex within that Region* • *Note 2: This StateMachine can be referenced by a Submachine State - the connection point references 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 StateMachine and referenced by the Submachine State* **Multiple StateMachines:** The order of listing in the Browser window determines the execution order. • When a Submachine State is involved, there might be multiple StateMachines under the Class • Use the Move Up or Move Down arrows in the Browser window toolbar to adjust the order of StateMachines; the top one will be set as the main StateMachine **Not Supported** • Protocol StateMachine • StateMachine Redefinition
States	• 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 StateMachine
Composite State Entry	• Default Entry • Explicit Entry • Shallow History Entry • Deep History Entry • Entry Point Entry
Substates	• Substates and Nested Substates Entry and Exit semantics, where the transition includes multiple nested levels of states, will obey the correct execution of nested behaviors (such as OnEntry and OnExit).
Transitions Support	• 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 For further details, refer to the *OMG UML Specification*.
Trigger and Events	An Executable StateMachine supports event handling for Signals only.

	To use Call, Timing or Change Event types you must define an outside mechanism to generate signals based on these events.
Signal	Attributes can be defined in Signals; the value of the attributes can be used as event arguments in Transition Guards and Effects.
	For example, this is the code set in the effect of a transition in C++:
	<pre> if(signal->signalEnum == ENUM_SIGNAL2)

    {

        int xVal = ((Signal2*)signal)->myVal;

    }</pre> |
| | Signal2 is generated as this code: |
| | <pre>   class Signal2 : public Signal{

    public:

        Signal2(){};

        Signal2(std::vector<String>& lstArguments);

        int myVal;

    };</pre> |
	Note: Further details can be found by generating an Executable StateMachine and referring to the generated 'EventProxy' file.
Initial	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.
Regions	**Default Activation & Explicit Activation:**
	Transitions terminate on the containing State:
	• If an initial Pseudostate is defined in the Region: **Default activation**
	• If no initial Pseudostate is defined, the Region will remain inactive and the containing State is treated as a Simple State
	• If the transition terminates on one of the Region's contained vertices: **Explicit activation**, resulting 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)
	For example, if there are three Regions defined for an Orthogonal State, and *RegionA* and *RegionB* have an Initial Pseudostate, then *RegionC* is explicitly activated. Default Activation applies to *RegionA* and *RegionB*; the containing State will have three active Regions.
Choice	Guard Constraints on all outgoing Transitions are evaluated dynamically, when the compound transition traversal reaches this Pseudostate.
Junction	Static conditional branch: guard constraints are evaluated before any compound transition is executed.
Fork / Join	Non-threaded, each active Region moves one step alternately, based on a completion event pool mechanism.
EntryPoint / ExitPoint Nodes	Non-threaded for orthogonal State or orthogonal StateMachine; each active Region moves one step alternately, based on a completion event pool mechanism.
History Nodes	• 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	Draw a self Transition, and change the Transition *kind* to Internal. Type 'defer();' in the 'Effect' field for the transition.
Connection Point References	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 and targets of Transitions. They represent entries into or exits out of the StateMachine referenced by the Submachine State.
State behaviors	State 'entry', 'doActivity' and 'exit' behaviors are defined as operations on a State. By default, you type the code that will be used for each behavior into the 'Code' panel of the Properties window for the Behavior operation. Note that you can change this to type the code into the 'Behavior' panel, by customizing the generation template. The 'doActivity' behavior generated will be run to completion before proceeding. The code is not concurrent with 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 to be the case for similar objects that, 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 StateMachines 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 StateMachine to copy	Now find the original Executable StateMachine Artifact in the Browser window.
Create the New Component	Whilst holding the Ctrl key, drag the original Artifact on to your diagram. You will be prompted with two questions. 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.

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. A big advantage in using Executable StateMachine Artifacts is that each of several parts in an Artifact can represent an instance of a StateMachine, so you can set up simulations using multiple instances of each StateMachine and observe how they interact. An example is provided in the *Example Executable StateMachine* Help topic.

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.

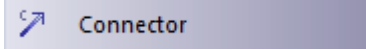| Action	Description
Drop a Class from the Browser window on to the <<Executable StateMachine>> Artifact	The easiest way to define properties on an Executable StateMachine is to drop the Class onto the Executable StateMachine from the Browser window. On the dialog that is shown, select the option to create a Property. You can then specify a name describing how the Executable StateMachine will refer to this property. 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.
Use and Connect Multiple UML Properties	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 Browser window as required.

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	The dialog for assigning property values can be opened by right-clicking on a Property and selecting 'Features	Set Property Values', or by using the keyboard shortcut Ctrl+Shift+R.
Assign a Value	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.

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	Connect multiple properties using the Connector relationship from the Composite toolbox. ⤴ Connector Alternatively, use the Quick Linker to create a relationship between two Properties and select 'Connector' as the relationship type.
Map to Class Model	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. The 'Choose an Association' dialog displays, which allows the generated StateMachine to send signals to the instance filling the role specified in the relationship during execution.

Code Generation for Executable StateMachines

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

Language Supported

An Executable StateMachine supports code generation for these platform languages:

- Microsoft Native C/C++
- Microsoft .NET (C#)
- Scripting (JavaScript)
- Oracle Java (Java)

From Enterprise Architect Release 14.1, code generation is supported without dependency on the simulation environment (compilers). For example, if you don't have Visual Studio installed, you can still generate code from the model and use it in your own project. The compilers are still needed if you want to simulate models in Enterprise Architect.

Simulation Environment (Compiler Settings)

If you want to simulate the Executable StateMachine model in Enterprise Architect, these platforms or compilers are required for the languages:

Language Platform	Example of Framework Path
Microsoft Native (C/C++)	C:\Program Files (x86)\Microsoft Visual Studio 12.0 C:\Program Files (x86)\Microsoft Visual Studio\2017\Professional (or other editions)
Microsoft .NET (C#)	C:\Windows\Microsoft.NET\Framework\v3.5 (or higher)
Scripting (JavaScript)	N/A
Oracle Java (Java)	C:\Program Files (x86)\Java\jdk1.7.0_17 (or higher)

Access

Ribbon	Simulate > Executable States > Statemachine > Generate, Build and Run or Simulate > Executable States > Statemachine > Generate

Generating Code

The 'Simulate > Executable States > Statemachine' ribbon options provide commands for generating code for the StateMachine. Select the Executable StateMachine Artifact first, then use the ribbon option to generate the code. The 'Executable Statemachine Code Generation' dialog displayed depends on the code language.

Generating Code (Java on Windows)

Project output directory	Displays the directory in which the generated code files will be stored. If necessary, click on the button at the right of the field to browse for and select a different directory. The names of the generated classes and their source file paths are displayed after this.
Executable Statemachine Target Machine	Select the 'Local' option.
Java JDK	Enter the installation directory of the Java JDK to be used.

Generating Code (Java on Linux)

Project output directory:	Displays the directory in which the generated code files will be stored. If necessary, click on the button at the right of the field to browse for and select a different directory. The names of the generated classes and their source file paths are displayed when the path is changed.
Executable Statemachine Target Machine	Select the 'Remote' option.
Operating System	Select Linux.
Port	This is the debugger Port to be used. You will find references to this Port number in the 'Debug' and 'DebugRun' sections of the Analyzer Script generated.

Generating Code (Other Languages)

At the same time the System Output window opens at the 'Executable StateMachine Output' page, on which progress messages, warnings or errors are displayed during code generation.

On the 'Executable StateMachine Code Generation' dialog, the 'Artifact' field and 'Language' field display the element name and coding language as defined in the element's 'Properties' dialog.

Field/Option	Description
Project output directory	Displays the directory in which the generated code files will be stored. If necessary, click on the button at the right of the field to browse for and select a different directory.
Project build environment	The fields and information in this panel vary depending on the language defined in the Artifact element and in the script. However, each supported language provides an option to define the path to the target frameworks that are required to build and run the generated code; examples are shown in the *Languages Supported* section of this topic.

This path, and its Local Paths ID, are defined in the 'Local Paths' dialog and shown here on the 'Executable StateMachine Code Generation' dialog. |

Generate

Click on this button to generate the StateMachine code. The code generation will overwrite any existing files in the project output directory. The set of files will include all required files including those for each Class referenced by the StateMachine.

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

Each Executable StateMachine that is generated will also generate an Execution Analyzer script, which is the configuration script for building, running and debugging the Executable StateMachine.
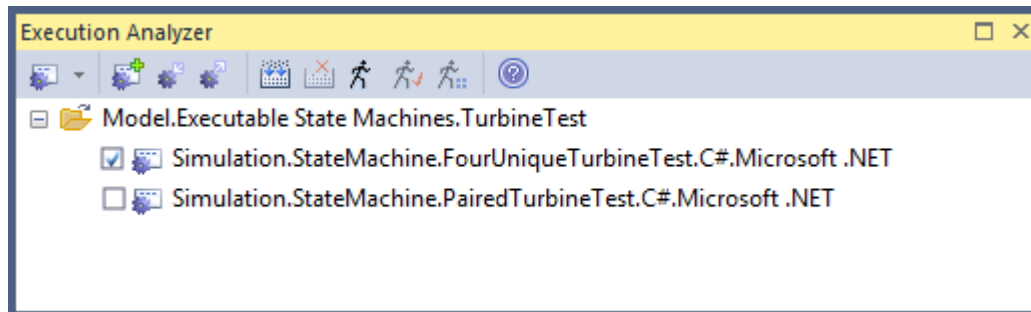
Building Code

The code generated by an Executable StateMachine can be built by Enterprise Architect in one of three ways.

Method	Description
Ribbon Generate, Build and Run Command	For the selected Executable StateMachine, generates the entire code base again. The source code is then compiled and the simulation started.
Ribbon Build Command	Compiles the code that has been generated. This can be used directly after generating the code, if you have made changes to the build procedure (the Analyzer Script) or modified the generated code in some way.
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 build directly using the built-in shortcut Ctrl+Shift+F12.
Build Output	When building, all output is shown on the 'Build' page of the System Output window. You can double-click on any compiler errors to open a source editor at the appropriate line.

Leveraging existing code

Executable StateMachines 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 StateMachine 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.

Debugging Execution of Executable StateMachines

Creation of Executable StateMachines provides benefits even after the generation of code. 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 StateMachines, demonstrated by the simulation and ultimately incorporated in a real world system.

Debugging a StateMachine

Being able to debug an Executable StateMachine gives additional benefits, such as being able to:

- Interrupt the execution of the simulation and all executing StateMachines
- View the raw state of each StateMachine 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 StateMachine 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 StateMachine using the Simulation Control. Depending on the nature of the behavior being debugged, however, 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 Browser window 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 want 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 StateMachines 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: < > = >= <=. For example:

(this.m_nCount > 100) and (this.m_ntype == 1)

To add a condition to a breakpoint you have set, right-click on the breakpoint and select 'Properties'. By clicking on the breakpoint while pressing 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, for example, an *effect*. 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 and prove the sequence of events and the order in which instances change state.

Viewing the Call Stack

Whenever a breakpoint is encountered, the Call Stack is available from the Analyzer menu. Use this to determine the sequence in which the execution is taking place.

Execution and Simulation of Executable StateMachines

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

Starting a Simulation

The Simulation Control toolbar provides a Search button that you use to select the Executable StateMachine Artifact to run. The control maintains a drop-down list of the most recent Executable StateMachines 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 (a higher value is faster). A value of zero will cause the simulation to halt after every step; this requires using the toolbar controls to manually step through 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 are labeled with the name of any instance currently in that particular state. If two or more Artifact properties of the same type share the same State, the State will have a separate label for each property name.

Generate Timing Diagram

After completing the simulation of an Executable StateMachine, you can generate a Timing diagram from the output. To do this:

In the Simulation window toolbar, click on 'Tools | Generate Timing Diagram'.
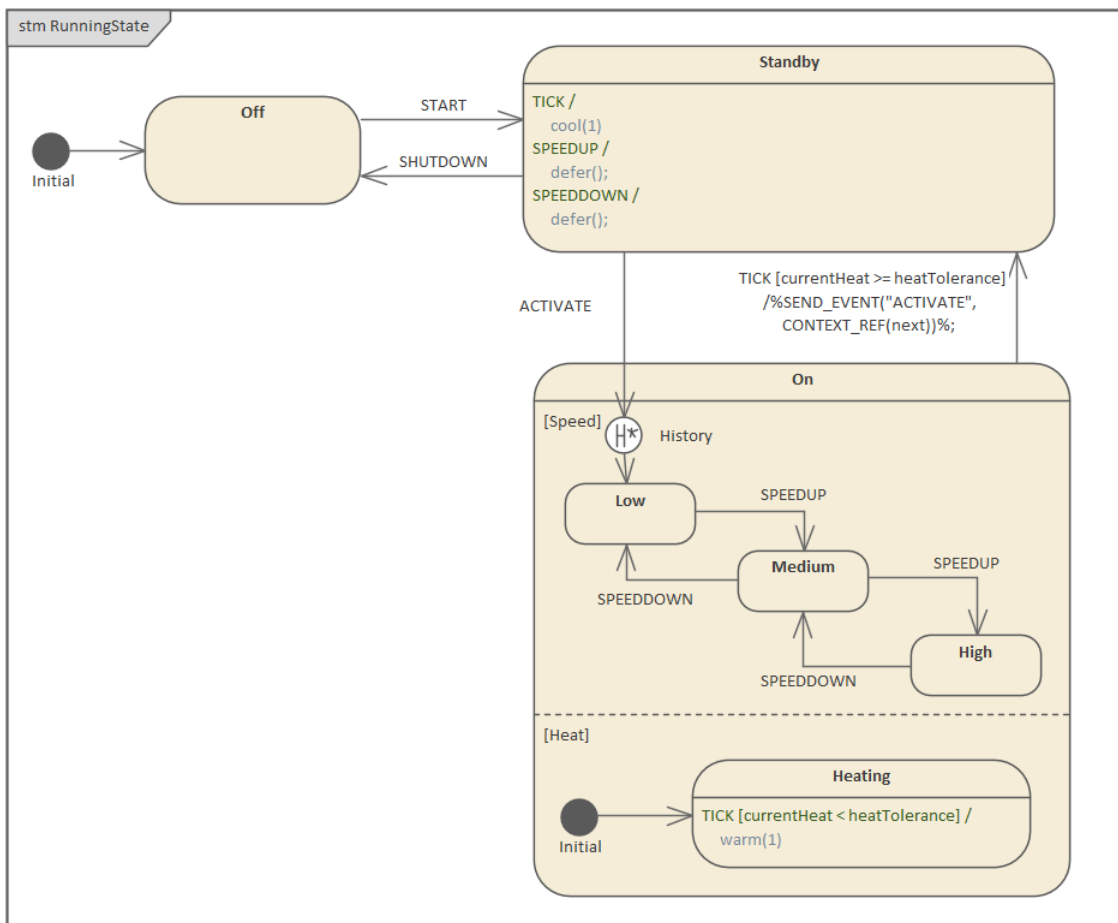
Example: Executable StateMachine

Example Class Model

This image shows a sample Class model that is used by the StateMachines described in this topic.
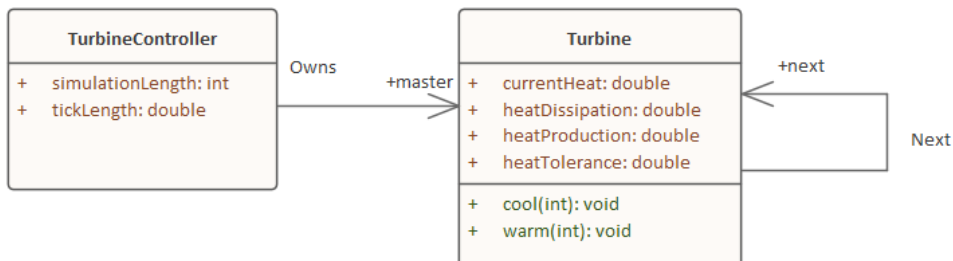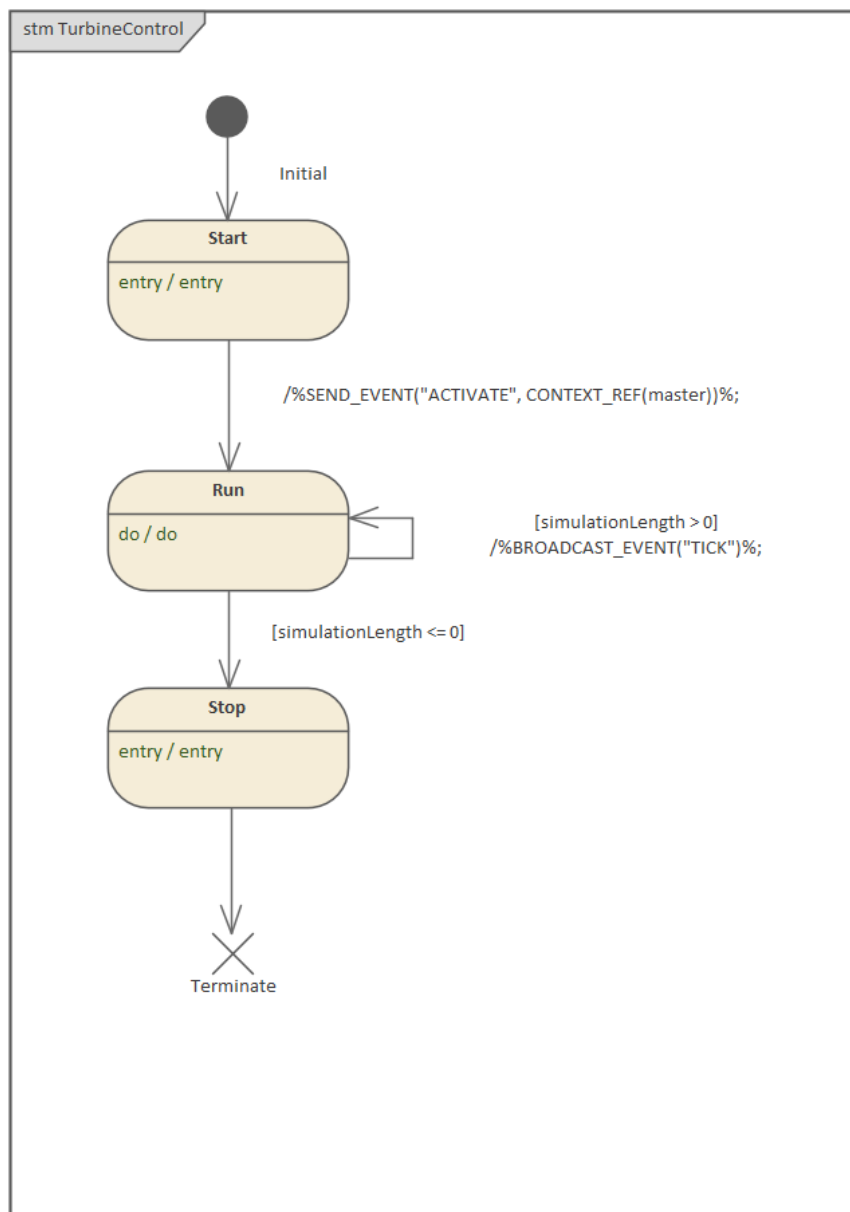

Example StateMachines

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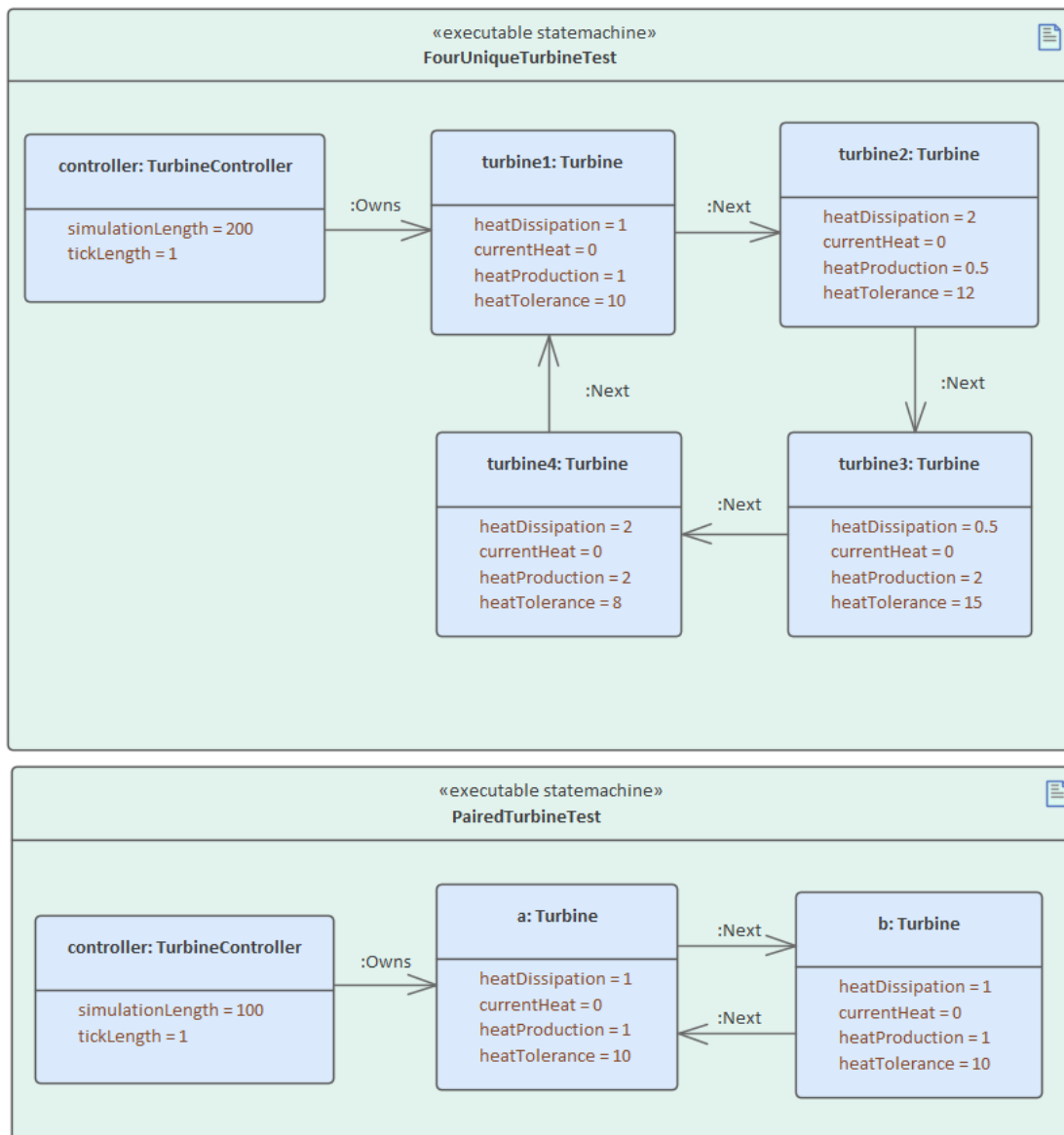

The top level controller.

Example Artifacts

From the example Class and StateMachine diagrams, we can create Executable StateMachines as shown here.

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

Simulation Results

When running a simulation, 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.
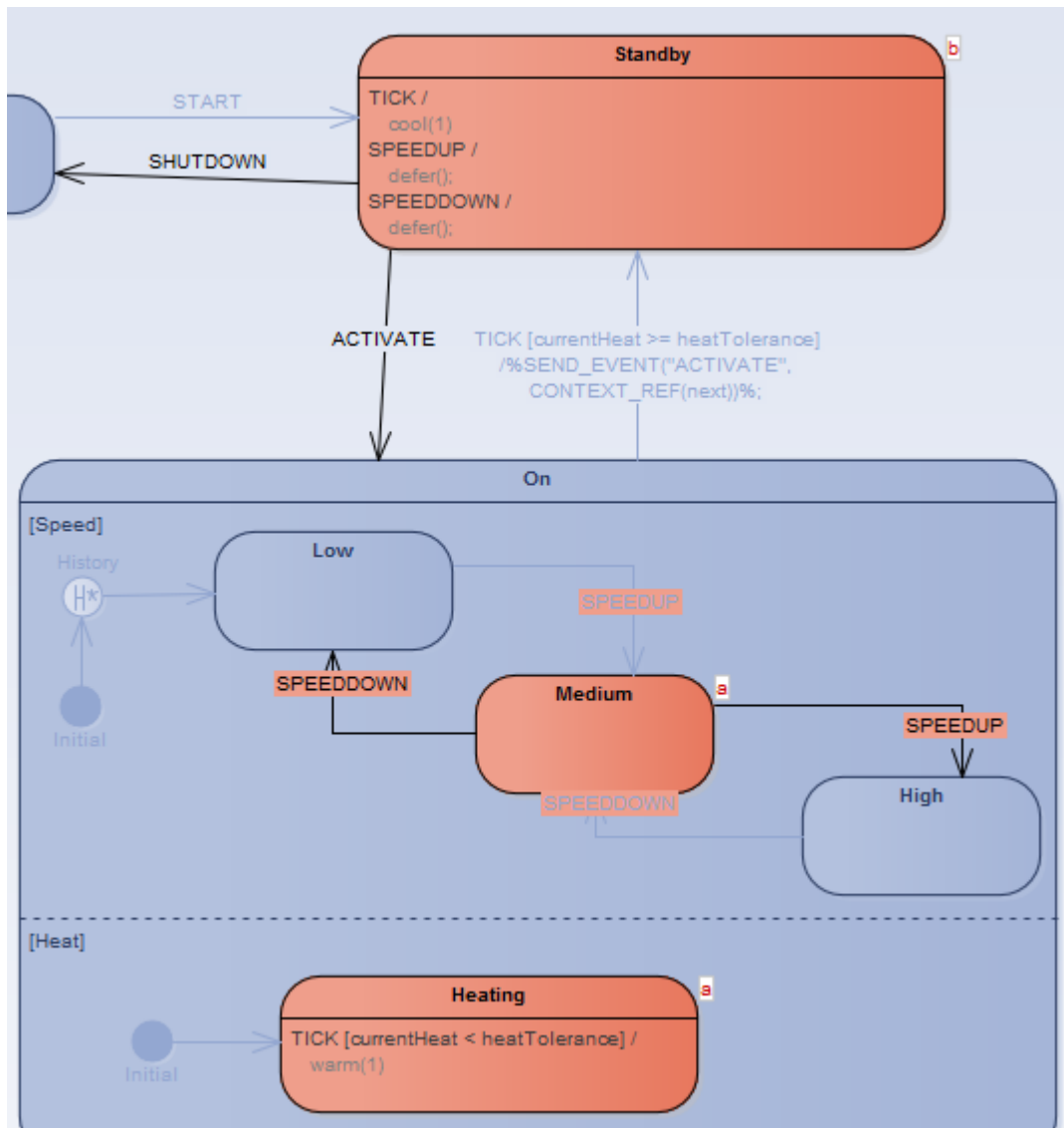
Example: Simulation Commands

This example demonstrates how we can use the Simulation window to observe Trace messages or send commands to control a StateMachine. Through the example, you can examine:
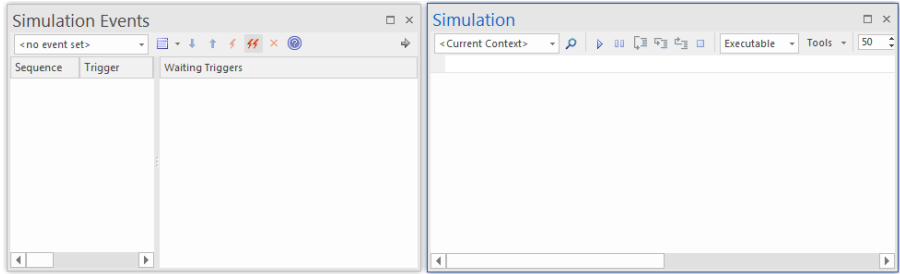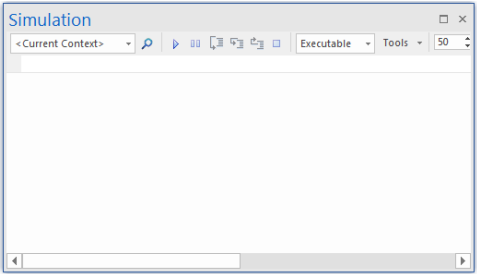
- An attribute of a context - the member variable defined in the Class, which is the context of the StateMachine; these attributes carry values in the scope of the context for all State behaviors and transition effects, to access and modify

- Each attribute of a Signal - the member variable defined in the Signal, which is referenced by an Event and can serve as an Event Parameter; each Signal Event occurrence might have different instances of a Signal

- The use of the 'Eval' command to query the runtime value of a context's attribute

- The use of the 'Dump' command to dump the current state's active count; it can also dump the current event deferred in the pool

This example is taken from the EAExample model:

Example Model.Model Simulation.Executable StateMachine.Simulation Commands

Access

| Ribbon	• Simulate > Dynamic Simulation > Simulator > Open Simulation Window) • Simulate > Dynamic Simulation > Events (for the Simulation Events window) These two windows are frequently used together in the simulation of Executable StateMachines.

Create Context and StateMachine

In this section we will create a Class called TransactionServer, which defines a StateMachine as its behavior. We then create an Executable StateMachine Artifact as the simulation environment.

Create the Context of the StateMachine
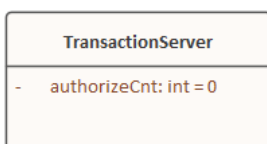
1. Create a Class element called *TransactionServer*.

2. In this Class, create an attribute called *authorizeCnt* with initial value 0.

3. In the Browser window, right-click on *TransactionServer* and select the 'Add | StateMachine' option.

Create the StateMachine
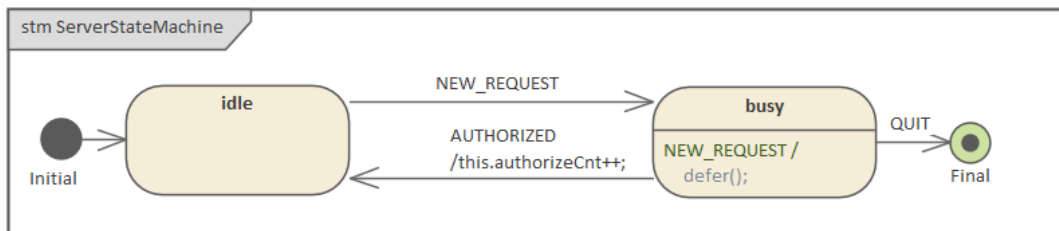
1. Create an Initial pseudostate called *Initial*.

2. Transition to a State called *idle*.

3. Transition to a State called *busy*, with the trigger NEW_REQUEST.

4. Transition:
    - To a Final pseudostate called *Final*, with the trigger QUIT
   - Back to *idle*, with the trigger AUTHORIZED, with the Effect 'this.authorizeCnt++;'

Create a Deferred Event for the State *busy*

1. Draw a self-transition for *busy*.

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();'.

Create a Signal and Attributes

1. Create a Signal element called *RequestSignal*.

2. Create an attribute called *requestType* with type 'int'.

3. Configure the Event NEW_REQUEST to reference *RequestSignal*.

**Create the Executable StateMachine Artifact**
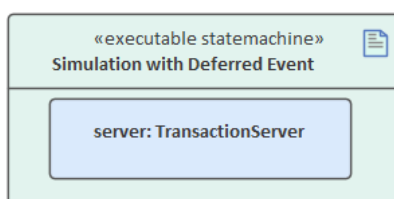
1. From the 'Simulation' page of the Diagram Toolbox, drag an 'Executable StateMachine' icon onto the diagram and call the element *Simulation with Deferred Event*.

2. Ctrl+Drag the *TransactionServer* element from the Browser window and drop it onto the Artifact as a property, with the name *server*.

3. Set the language of the Artifact to JavaScript, which does not require a compiler (for the example; in production you could also use C, C++, C#, or Java, which also support Executable StateMachines).

4. Click on the Artifact and select the 'Simulate > Executable States > Statemachine > Generate, Build and Run' ribbon option.

Simulation Window and Commands

When the simulation starts, *idle* is the current state.

The Simulation window shows that the Transition Effect, Entry and Do behavior is finished for state *idle*, and the StateMachine is waiting for a trigger.
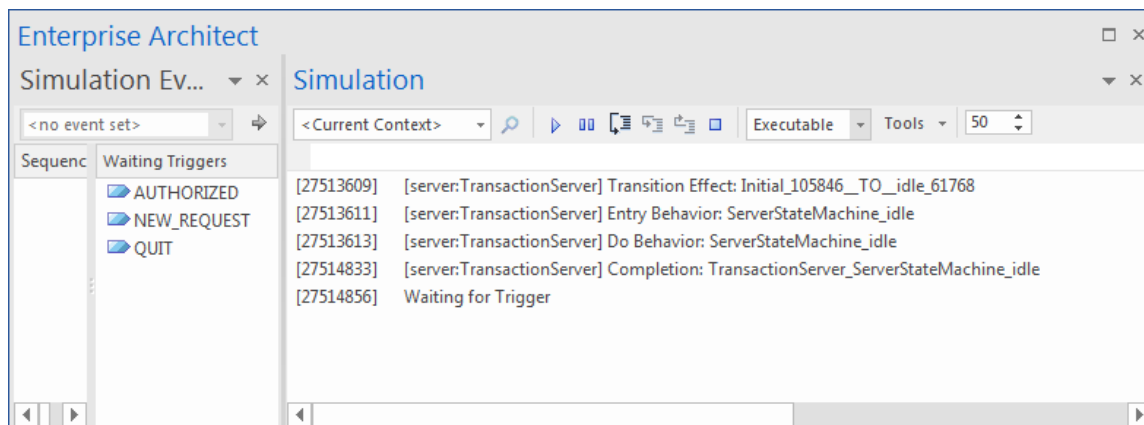
Event Data via Values for Signal Attributes

For the Trigger Signal Event NEW_REQUEST, the 'Trigger Parameter Entry' dialog displays to prompt for values for the listed attributes defined in the Signal *RequestSignal*, referenced by NEW_REQUEST.
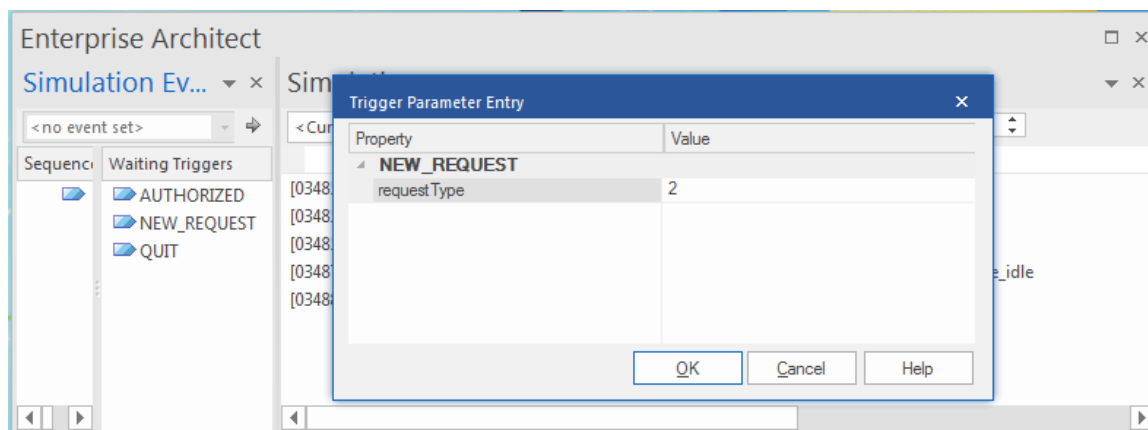
Type the value '2' and click on the OK button. The Signal attribute values are then passed to invoked methods such as the State's behaviors and the Transition's effects.
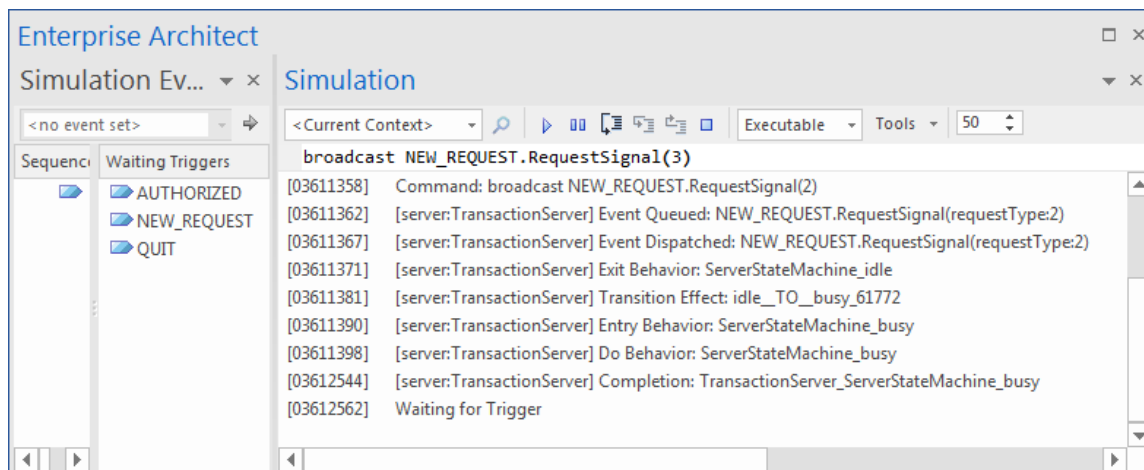
These messages are output to the Simulation window:

[03612562]	Waiting for Trigger
[03611358]	Command: **broadcast NEW_REQUEST.RequestSignal(2)**
[03611362]	[server:TransactionServer] Event Queued: NEW_REQUEST.RequestSignal(requestType:2)
[03611367]	[server:TransactionServer] Event Dispatched: NEW_REQUEST.RequestSignal(requestType:2)
[03611371]	[server:TransactionServer] Exit Behavior: ServerStateMachine_idle

[03611381]   [server:TransactionServer] Transition Effect: idle__TO__busy_61772

[03611390]   [server:TransactionServer] Entry Behavior: ServerStateMachine_busy

[03611398]   [server:TransactionServer] Do Behavior: ServerStateMachine_busy

[03612544]   [server:TransactionServer] Completion: TransactionServer_ServerStateMachine_busy

[03612562]   Waiting for Trigger

We can broadcast events by double-clicking on the item listed in the Simulation Events window. Alternatively, we can type a command string in the text field of the Simulation window (underneath the toolbar).
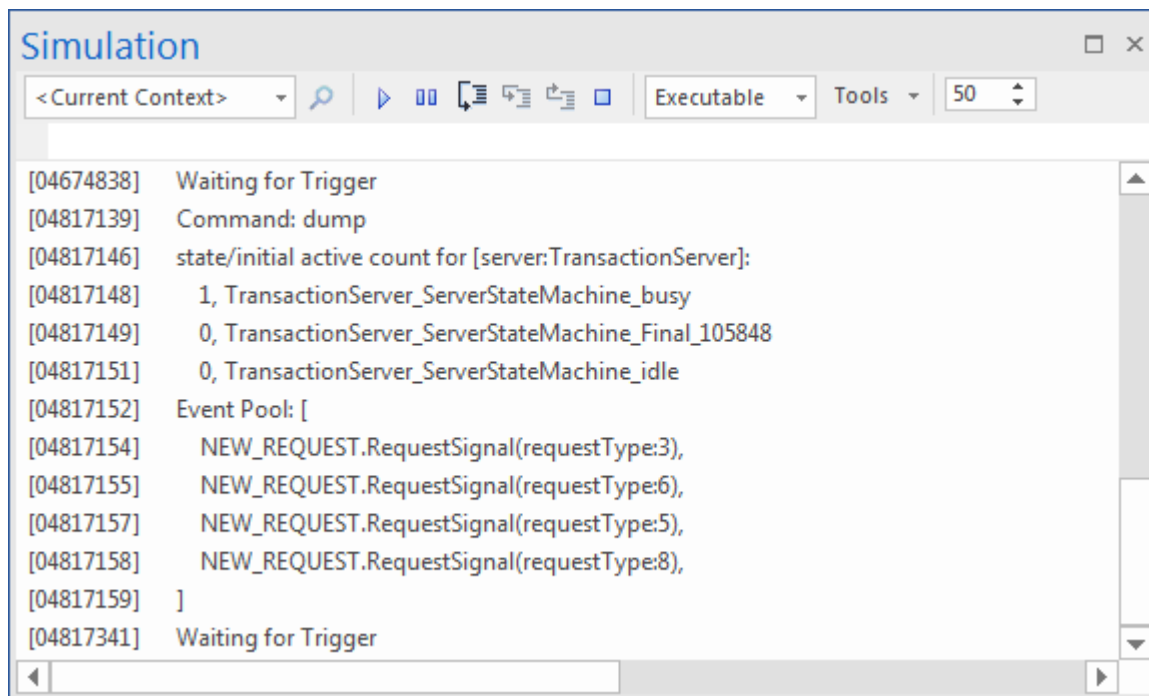

[03612562]   Waiting for Trigger

[04460226]   Command: **broadcast NEW_REQUEST.RequestSignal(3)**

[04460233]   [server:TransactionServer] Event Queued: NEW_REQUEST.RequestSignal(requestType:3)

[04461081]   Waiting for Trigger

The Simulation message indicates that the event occurrence is deferred (Event Queued, but not dispatched). We can run further commands using the text field:

[04655441]   Waiting for Trigger

[04664057]   Command: **broadcast NEW_REQUEST.RequestSignal(6)**

[04664066]   [server:TransactionServer] Event Queued: NEW_REQUEST.RequestSignal(requestType:6)

[04664803]   Waiting for Trigger

[04669659]   Command: **broadcast NEW_REQUEST.RequestSignal(5)**

[04669667]   [server:TransactionServer] Event Queued: NEW_REQUEST.RequestSignal(requestType:5)

[04670312]   Waiting for Trigger

[04674196]   Command: **broadcast NEW_REQUEST.RequestSignal(8)**

[04674204]   [server:TransactionServer] Event Queued: NEW_REQUEST.RequestSignal(requestType:8)

[04674838]   Waiting for Trigger

dump: Query 'active count' for a State and Event Pool

Type *dump* in the text field; these results display:

```
Simulation                                                                    □  ✕
<Current Context>      ▾  🔍   ▷  ◫◫  ⬐▤  ▤⬎  ⬑▤  ▢   Executable  ▾  Tools  ▾   50  ⬍

[04674838]   Waiting for Trigger                                                    ▲
[04817139]   Command: dump
[04817146]   state/initial active count for [server:TransactionServer]:
[04817148]      1, TransactionServer_ServerStateMachine_busy
[04817149]      0, TransactionServer_ServerStateMachine_Final_105848
[04817151]      0, TransactionServer_ServerStateMachine_idle
[04817152]   Event Pool: [
[04817154]      NEW_REQUEST.RequestSignal(requestType:3),
[04817155]      NEW_REQUEST.RequestSignal(requestType:6),
[04817157]      NEW_REQUEST.RequestSignal(requestType:5),
[04817158]      NEW_REQUEST.RequestSignal(requestType:8),
[04817159]   ]
[04817341]   Waiting for Trigger                                                    ▼
◀                                                                              ▶
```

From the 'active count' section, we can see that *busy* is the active state (active count is 1).

Tips: For a Composite State, the active count is 1 (for itself) *plus* the number of active regions.

From the 'Event Pool' section, we can see that there are four event occurrences in the Event Queue. Each instance of the signal carries different data.

The order of the events in the pool is the order in which they are broadcast.

eval: Query Run Time Value of the Context

Trigger AUTHORIZED,

[04817341]   Waiting for Trigger

[05494672]   Command: broadcast AUTHORIZED

[05494678]   [server:TransactionServer] Event Queued: AUTHORIZED

[05494680]   [server:TransactionServer] Event Dispatched: AUTHORIZED

[05494686]   [server:TransactionServer] Exit Behavior: ServerStateMachine_busy

[05494686]   [server:TransactionServer] **Transition Effect: busy__TO__idle_61769**

[05494687]   [server:TransactionServer] Entry Behavior: ServerStateMachine_idle

[05494688]   [server:TransactionServer] Do Behavior: ServerStateMachine_idle

[05495835]   [server:TransactionServer] Completion: TransactionServer_ServerStateMachine_idle

[05495842]   [server:TransactionServer] **Event Dispatched: NEW_REQUEST.RequestSignal(requestType:3)**

[05495844]   [server:TransactionServer] Exit Behavior: ServerStateMachine_idle

[05495846]   [server:TransactionServer] Transition Effect: idle__TO__busy_61772

[05495847]   [server:TransactionServer] Entry Behavior: ServerStateMachine_busy

[05495850]   [server:TransactionServer] Do Behavior: ServerStateMachine_busy

[05496349]   [server:TransactionServer] Completion: TransactionServer_ServerStateMachine_busy

[05496367]   Waiting for Trigger

- The transition from *busy* to *idle* is made, so we expect the effect to be executed

- One event is recalled from the pool and dispatched when *idle* is completed, causing *busy* to become the active state

- Type *dump* and notice that there are three events left in the pool; the first one is recalled and dispatched

```
[05693348]     Event Pool: [
[05693349]        NEW_REQUEST.RequestSignal(requestType:6),
[05693351]        NEW_REQUEST.RequestSignal(requestType:5),
[05693352]        NEW_REQUEST.RequestSignal(requestType:8),
[05693354]     ]
```

Type *eval server.authorizeCnt* in the text field. This figure indicates that the run time value of 'server.authorizeCnt' is 1.
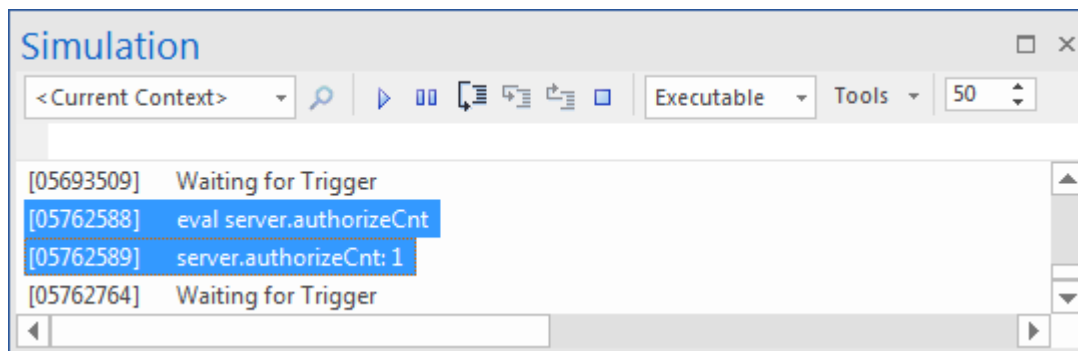
Trigger AUTHORIZED again. When the StateMachine is stable at *busy*, there will be two events left in the pool. Run *eval server.suthorizeCnt* again; the value will be 2.

Access Context's Member Variable from State Behavior and Transition Effect

Enterprise Architect's Executable StateMachine supports simulation for C, C++, C#, Java and JavaScript.

For C and C++, the syntax differs from C#, Java and JavaScript in accessing the context's member variables. C and C++ use the pointer '->' while the others simply use '.'; however, you can always use *this.variableName* to access the variables. Enterprise Architect will translate it to this->variableName for C and C++.

So for all languages, simply use this format for the simulation:

   this.variableName

Examples:

In the transition's effect:

   this.authorizeCnt++;

In some state's entry, do or exit behavior:

   this.foo += this.bar;

Note: by default Enterprise Architect is only replacing 'this->' with 'this' for C and C++; For example:

   this.foo = this.bar + myObject.iCount + myPointer->iCount;

will be translated to:

   this->foo = this->bar + myObject.iCount + myPointer->iCount;

A Complete List of Supported Commands

Since the Executable StateMachine Artifact can simulate multiple contexts together, some of the commands can specify an instance name.

run StateMachine:

As each context can have multiple StateMachines, the 'run' command can specify a StateMachine to start with.

- run instance.statemachine
- run all.all
- run instance
- run all
- run

For example:

    run

    run all

    run server

    run server.myMainStatemachine

broadcast & send Event:

- broadcast EventString
- send EventString to instance
- send EventString (equivalent to broadcast EventString)

For example:

    broadcast Event1

    send Event1 to client

dump Command:

- dump
- dump instance

For example:

    dump

    dump server

    dump client

eval Command:

- eval instance.variableName

For example:

    eval client.requestCnt

    eval server.responseCnt

exit Command:

- exit

The EventString's Format:

- EventName.SignalName(argument list)

Note: the argument list should match the attributes defined in the signal **by order**.

For example, if the Signal defines two attributes:

- foo
- bar

Then these EventStrings are valid:

- Event1.Signal1(10, 5)   --------- foo = 10; bar = 5
- Event1.Signal1(10,)   --------- foo = 10; bar is undefined
- Event1.Signal1(,5)   --------- bar = 10; foo is undefined
- Event1.Signal1(,)   --------- both foo and bar are not defined

If the Signal does not contain any attributes, we can simplify the EventString to:

- EventName

Example: Simulation in HTML with JavaScript

We already know that users can model an Executable StateMachine and simulate it in Enterprise Architect with the generated code. Using the two examples *CD Player* and the *Regular Expression Parser*, we will now demonstrate how you can integrate the generated code with your real projects.

Enterprise Architect provides two different mechanisms for client code to use a StateMachine:

- Active State Based - the client can query the current active state, then 'switch' the logic based on the query result
- Runtime Variable Based - the client does not act on the current active state, but does act on the runtime value of the variables defined in the Class containing the StateMachine

In the *CD Player* example, there are very few states and many buttons on the GUI, so it is quite easy to implement the example based on the Active State Mechanism; we will also query the runtime value for the current track.

In the *Regular Expression Parser* example the StateMachine handles everything, and a member variable *bMatch* changes its runtime value when states change. The client does not register how many states are there or which state is currently active.

In these topics, we demonstrate how to model, simulate and integrate a CD Player and a Parser for a specified Regular Expression, step by step:

- CD Player
- Regular Expression Parser

CD Player

The behavior of a CD Player application might appear intuitive; however, there are many rules related to when the buttons are enabled and disabled, what is displayed in the text fields of the window and what happens when you supply events to the application.

Suppose our example CD Player has these features:

- Buttons - Load Random CD, Play, Pause, Stop, Previous Track, Next Track and Eject
- Displays - Number Of Tracks, Current Track, Track Length and Time Elapsed

StateMachine for CD Player

A Class *CDPlayer* is defined with two attributes: *currentTrack* and *numberOfTracks*.

CDPlayer
- currentTrack: int
- numberOfTracks: int
+ next(): void
+ previous(): void
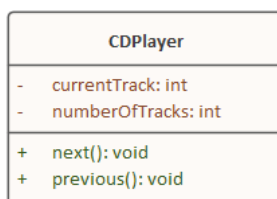
A StateMachine is used to describe the states of the CD Player:
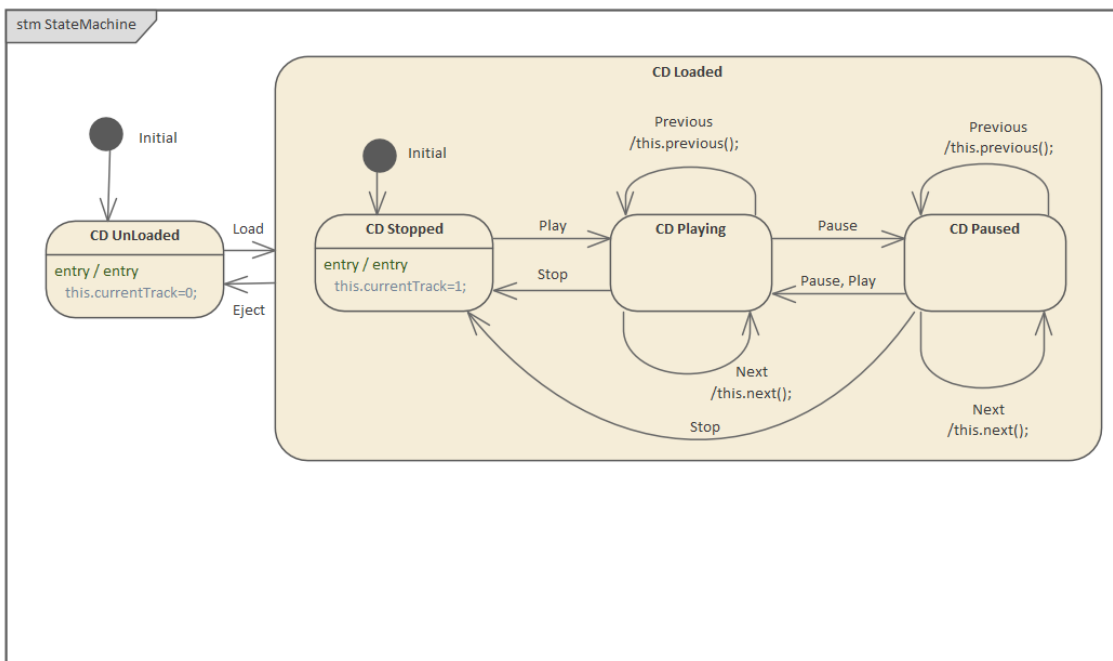
- On the higher level, the StateMachine has two States: *CD UnLoaded* and *CD Loaded*
- *CD Loaded* can be composed of three simple States: *CD Stopped*, *CD Playing*, *CD Paused*
- Transitions are defined with triggers for the events Load, Eject, Play, Pause, Stop, Previous and Next
- State behaviors and transition Effects are defined to change the value of attributes defined in *CDPlayer*; for example, the 'Previous' event will trigger the self transition (if the current state is *CD Playing* or *CD Paused*) and the Effect will be executed, which will decrement the value of *currentTrack* or wrap to the last track

We can create an Executable StateMachine Artifact and create a property typing to *CDPlayer*, then simulate the

StateMachine in Enterprise Architect to make sure the model is correct.
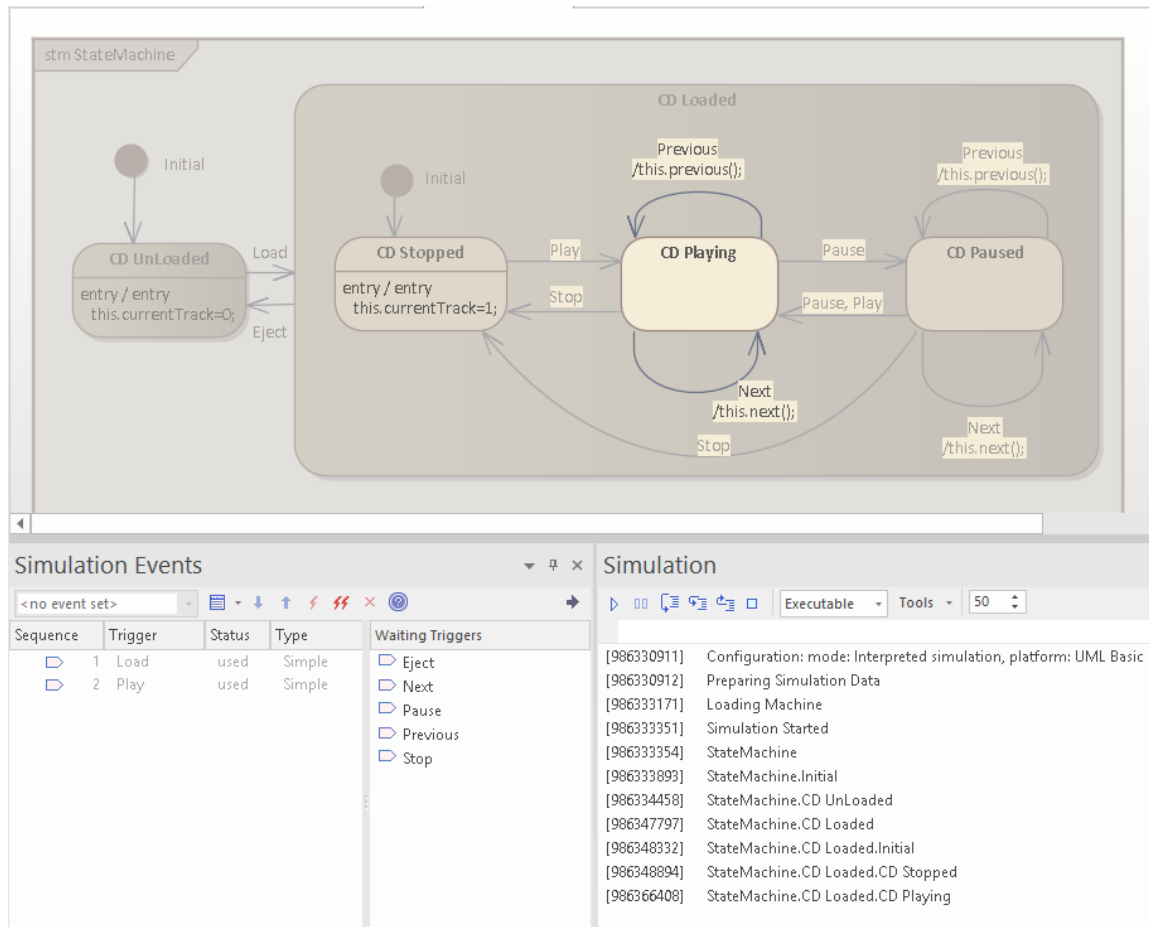
Inspect the code generated

Enterprise Architect will generate these files in a folder that you have specified:

- Back-end code: CDPlayer.js, ContextManager.js, EventProxy.js
- Client code: ManagerWorker
- Front-end code: statemachineGUI.js, index.html
- Other code: SimulationManager.js

File	Description
/CDPlayer.js	This file defines the Class CDPlayer and its attributes and operations. It also defines the Class's StateMachines with the State behaviors and the transition effects.
/ContextManager.js	This file is the abstract manager of contexts. The file defines the contents that are independent of the actual contexts, which are defined in the generalization of the *ContextManager*, such as *SimulationManager* and *ManagerWorker*.
	The simulation (Executable StateMachine Artifact) can involve multiple contexts; for example, in a tennis game simulation there will be one *umpire* typed to Class *Umpire,* and two players - *playerA* and *playerB* - typed to Class *Player*. Both Class *Umpire* and Class *Player* will define their own StateMachine(s).
/EventProxy.js	This file defines Events and Signals used in the simulation.

	If we are raising an Event with arguments, we model the Event as a Signal Event, which specifies a Signal Class; we then define attributes for the Signal Class. Each Event occurrence has an instance of the Signal, carrying the runtime values specified for the attributes.
/SimulationManager.js	This file is for simulation in Enterprise Architect.
/html/ManagerWorker.js	This file serves as a middle layer between the front-end and back-end. • The front-end posts a message to request information from the ManagerWorker • Since the ManagerWorker generalizes from ContextManager, it has full access to all the contexts such as querying the current active state and querying the runtime value of a variable • The ManagerWorker will post a message to the front-end with the data it retrieved from the back-end
/html/statemachineGUI.js	This file establishes the communication between the front-end and the ManagerWorker, by defining *stateMachineWorker*. It: • Defines the functions *startStateMachineWebWorker* and *stopStateMachineWebWorker* • Defines the functions *onActiveStateResponse* and *onRuntimeValueResponse* with place-holder code: //to do: write user's logic You could simply replace this comment with your logic, as will be demonstrated later in this topic
/html/index.html	This defines the HTML User Interface, such as the buttons and the input to raise Events or display information. You can define CSS and JavaScript in this file.

Customize index.html and statemachineGUI.js

Make these changes to the generated files:

• Create buttons and displays

• Create a CSS style to format the display and enable/disable the button images

• Create an ElapseTimeWorker.js to refresh the display every second

• Create a TimeElapsed function, set to Next Track when the time elapsed reaches the length of the track

• Create JavaScript as the button 'onclick' event handler

• Once an event is broadcast, request the active State and runtime value for *cdPlayer.currentTrack*

• On initialization, request the active State

In statemachineGUI.js find the function *onActiveStateResponse_cdPlayer*

• In CDPlayer_StateMachine_CDUnLoaded, disable all buttons and enable btnLoad

• In CDPlayer_StateMachine_CDLoaded_CDStopped, disable all buttons and enable btnEject and btnPlay

• In CDPlayer_StateMachine_CDLoaded_CDPlaying, enable all buttons and disable btnLoad and btnPlay

• In CDPlayer_StateMachine_CDLoaded_CDPaused, enable all buttons and disable btnLoad

In statemachineGUI.js find the function *onRuntimeValueResponse*

- In *cdPlayer.currentTrack*, we update the display for current track and track length

The Complete Example

The example can be accessed from the 'Resources' page of the Sparx Systems website, by clicking on this link:
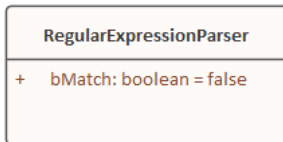CD Player Simulation

Click on the Load Random CD button, and then on the Start Simulation button.

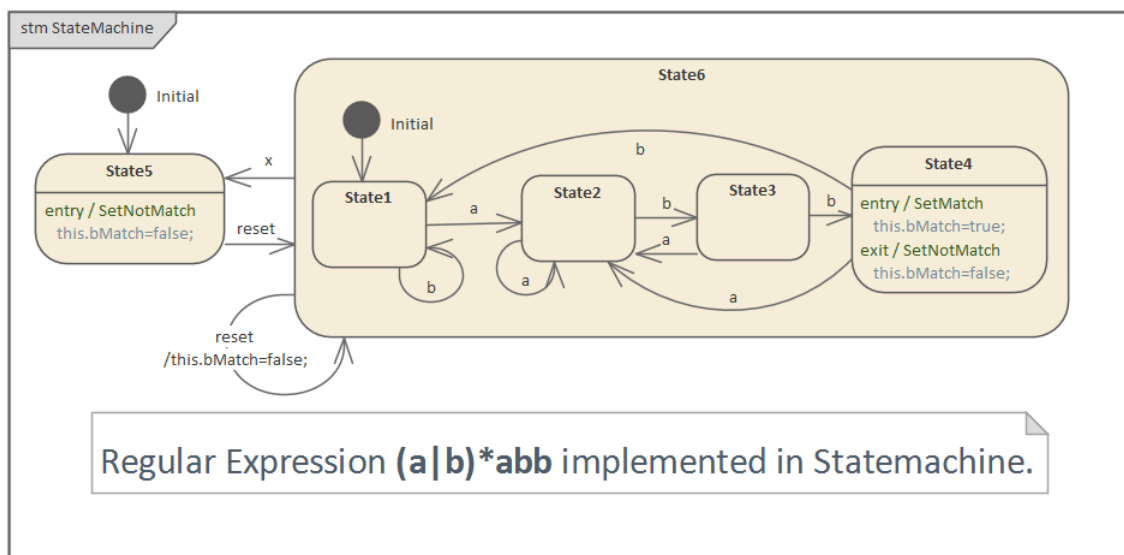# Regular Expression Parser

StateMachine for Regular Expression Parser

The Class *RegularExpressionParser* is defined with one attribute: bMatch.

RegularExpressionParser
+   bMatch: boolean = false

A StateMachine is used to describe the regular expression (a|b)*abb

Regular Expression **(a|b)*abb** implemented in Statemachine.

- The transition triggers are specified as events *a, b, x* and *reset*
- On entry to State4, bMatch is set to True; on exit from State4, bMatch is set to False
- On entry to State5, bMatch is set to False
- On self transition of State6, bMatch is set to False

Customize index.html and statemachineGUI.js

Make these changes to the generated files:
- Create an HTML input field and an image to indicate the result
- Create JavaScript as the field's *oninput* event handler
- Create the function 'SetResult' to toggle the pass/fail image
- Create the function 'getEventStr', which will return 'a' on '*a*' and 'b' on '*b*', but will return 'x' on any other character
- On initialize, broadcast '*reset*'
- On the broadcast event, request the runtime variable 'regxParser.bMatch'

In *statemachineGUI.js*, find the function 'onRuntimeValueResponse'.

- In 'regxParser.bMatch', we will receive 'True' or 'False' and pass it into 'SetResult' to update the image

The Complete Example

The example can be accessed from the 'Resources' page of the Sparx Systems website, by clicking on this link:

Regular Expression Parser Simulation

Example: Entering a State

The semantics of entering a State depend on the type of State and the manner in which it is entered.

In all cases, the entry Behavior of the State is executed (if defined) upon entry, but only after any effect Behavior associated with the incoming Transition is completed. Also, if a doActivity Behavior is defined for the State, this Behavior commences execution immediately after the entry Behavior is executed.

For a Composite State with one or more Regions defined, a number of alternatives exist for each Region:

- *Default entry*: This situation occurs when the owning Composite State is the direct target of a Transition; after executing the entry Behavior and forking a possible doActivity Behavior execution, State entry continues from an initial Pseudostate via its outgoing Transition (known as the default Transition of the State) if it is defined in the Region
If no initial Pseudostate is defined, this Region will not be active

- *Explicit entry*: If the incoming Transition or its continuations terminate on a directly contained Substate of the owning composite State, then that Substate becomes active and its entry Behavior is executed after the execution of the entry Behavior of the containing composite State
This rule applies recursively if the Transition terminates on an indirect (deeply nested) Substate

- *Shallow history entry*: If the incoming Transition terminates on a shallowHistory Pseudostate of this Region, the active Substate becomes the Substate that was most recently active (except FinalState) prior to this entry, unless this is the first entry into this State; if it is the first entry into this State or the previous entry had reached a Final, a default shallow history Transition will be taken if it is defined, otherwise the default State entry is applied

- *Deep history entry*: The rule for this case is the same as for shallow history except that the target Pseudostate is of type deepHistory and the rule is applied recursively to all levels in the active State configuration below this one

- *Entry point entry*: If a Transition enters the owning composite State through an entryPoint Pseudostate, then the outgoing Transition originating from the entry point and penetrating into the State in this region is taken; if there are more outgoing Transitions from the entry points, each Transition must target a different Region and all Regions are activated concurrently

For orthogonal States with multiple Regions, if the Transition explicitly enters one or more Regions (in the case of a Fork or entry point), these Regions are entered explicitly and the others by default.

In this example, we demonstrate a model with all these entry behaviors for an orthogonal State.

Modeling a StateMachine
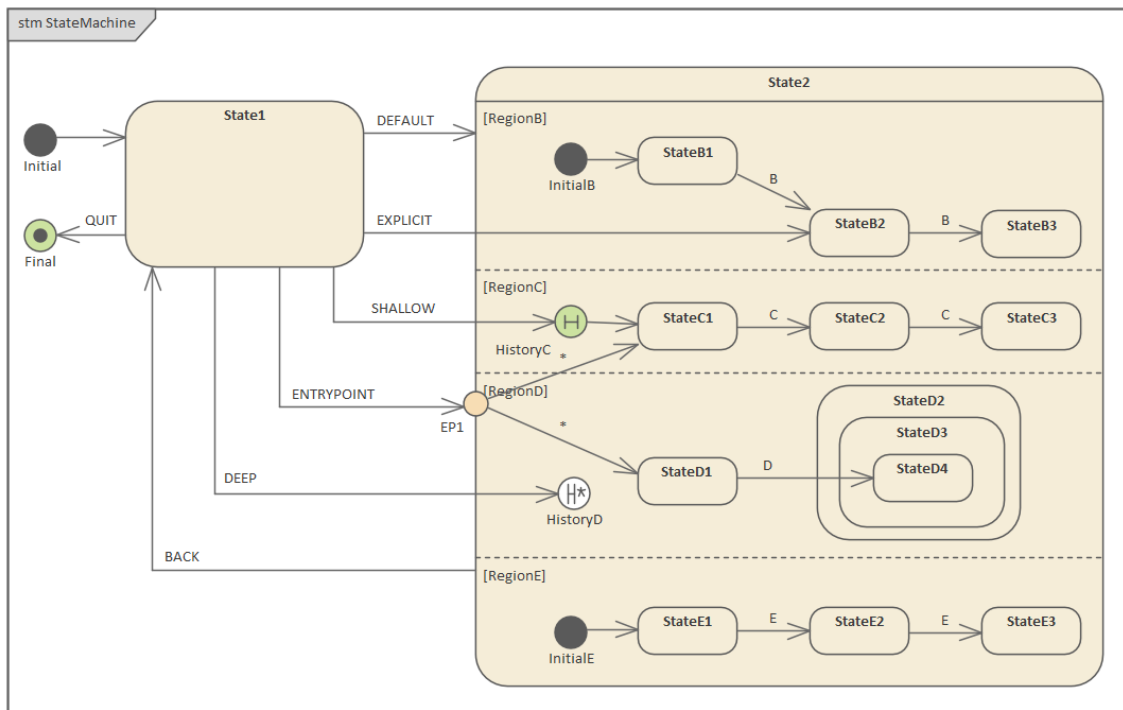
Context of StateMachine

1. Create a Class element named *MyClass*, which serves as the context of the StateMachine.
2. Right-click on *MyClass* in the Browser window and select the 'Add | StateMachine' option.

StateMachine

1. Add to the diagram an *Initial* Node, a State named *State1*, a State named *State2*, and a Final element named *final.*
2. Enlarge *State2* on the diagram, right-click on it and select the 'Advanced | Define Concurrent Substates' option, and define *RegionB, RegionC, RegionD* and *RegionE*.
3. Right-click on *State2* and select the 'New Child Element | Entry Point' option to create the Entry Point *EP1*.
4. In *RegionB*, create the elements *InitialB*, transition to *StateB1*, transition to *StateB2*, transition to *StateB3*; all transitions triggered by Event *B*.
5. In *RegionC*, create the elements shallow *HistoryC* (right-click on History node | Advanced | Deep History | uncheck), transition to *StateC1*, transition to *StateC2*, transition to *StateC3*; all transitions triggered by Event *C.*
6. In *RegionD*, create the elements deep *HistoryD* (right-click on History node | Advanced | Deep History | check), transition to *StateD1*, create *StateD2* as parent of *StateD3*, which is parent of *StateD4*; transition from *StateD1* to *StateD4*; triggered by Event *D*.
7. In *RegionE*, create the elements *InitialE*, transition to *StateE1*, transition to *StateE2*, transition to *StateE3*; all transitions triggered by Event *E.*
8. Draw transitions from the Entry Point *EP1* to *StateC1* and *StateD1*.

Draw transitions for different entry types:

1. Default Entry: *State1* to *State2*; triggered by Event DEFAULT.
2. Explicit Entry: *State1* to *StateB2*; triggered by Event EXPLICIT.
3. Shallow History Entry: *State1* to *HistoryC*; triggered by Event SHALLOW.
4. Deep History Entry: *State1* to *HistoryD*; triggered by Event DEEP.
5. Entry Point Entry: *State1* to *EP1*; triggered by Event ENTRYPOINT.

Other Transitions:

1. Composite State Exit: from *State2* to *State1*; triggered by Event BACK.

2. *State1* to *Final*, triggered by Event QUIT.
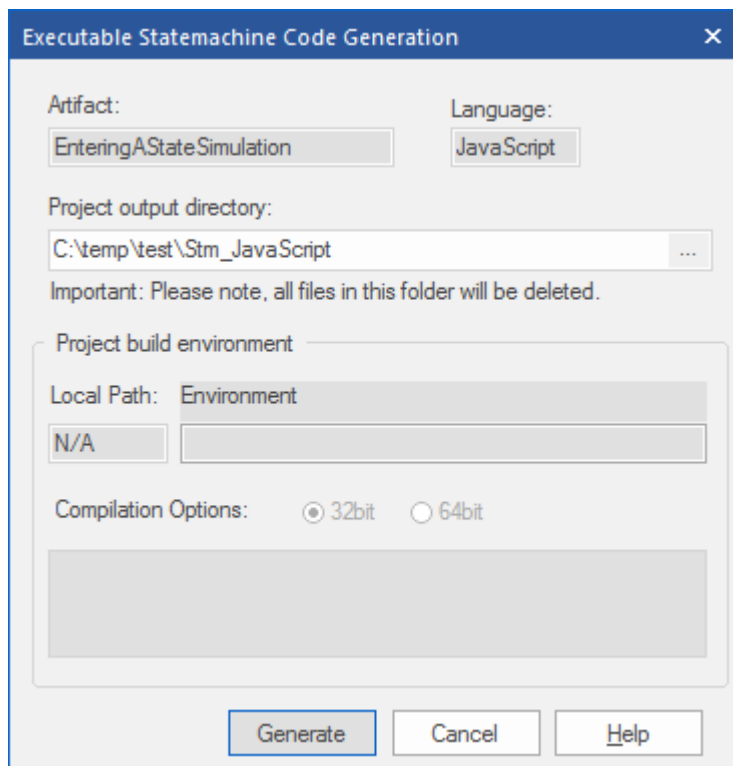
Simulation

Artifact

Enterprise Architect supports C, C++, C#, Java and JavaScript. We use JavaScript in this example because we don't need to install a compiler. (For other languages, either Visual Studio or JDK are required.)

3. On the 'Simulation' page of the Diagram Toolbox, drag the Executable StateMachine icon onto a diagram and create an Artifact named *EnteringAStateSimulation*. Set the language to JavaScript.

4. Ctrl+drag the *MyClass* element from the Browser window onto the *EnteringAStateSimulation* Artifact, select the 'Paste as Property' option and give the Property the name *myClass*.
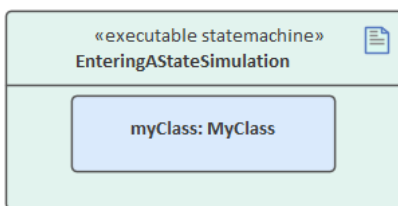
Code Generation

9. Click on *EnteringAStateSimulation* and select the 'Simulate > Executable States > Statemachine > Generate, Build and Run' ribbon option.

10. Specify a directory for the generated source code.

Note: The contents of this directory will be cleared before generation; make sure you specify a directory that is used only for StateMachine simulation purposes.

Run Simulation

Tips: You can view the execution trace sequence from the Simulation window, which you open by selecting the 'Simulate > Dynamic Simulation > Simulator > Open Simulation Window' ribbon option

When the simulation begins, *State1* is active and the StateMachine is waiting for events.
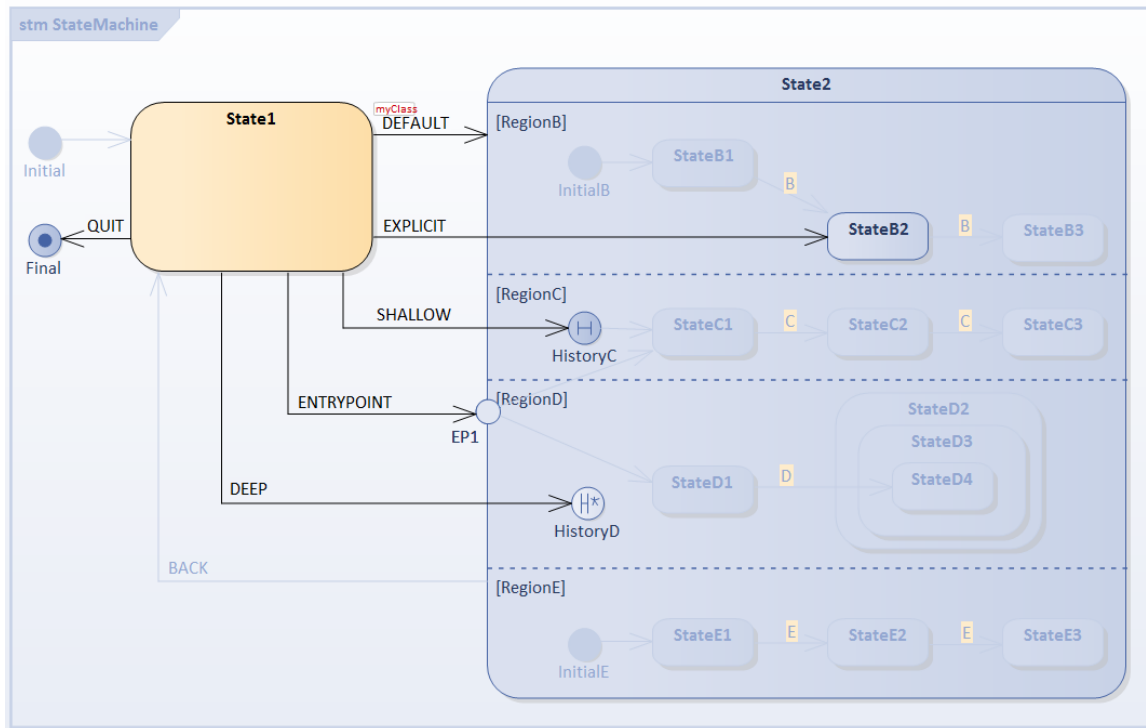
Open the Simulation Events (Triggers) window using the 'Simulate > Dynamic Simulation > Events' ribbon option.

1) Select the Default Entry: Trigger Sequence [DEFAULT].
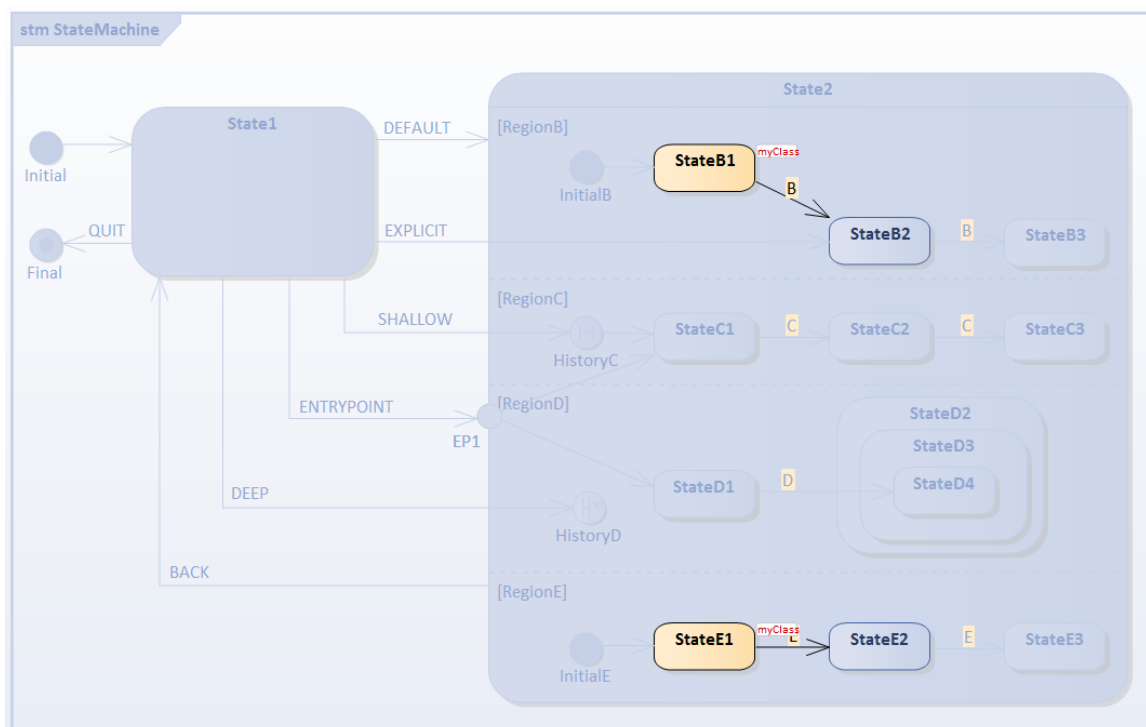
- *RegionB* is activated because it defines *InitialB*; the transition outgoing from it will be executed, *StateB1* is the active state

- *RegionE* is activated because it defines *InitialE*; the transition outgoing from it will be executed, *StateE1* is the active state

- *RegionC* and *RegionD* are inactive because no Initial Pseudostates were defined
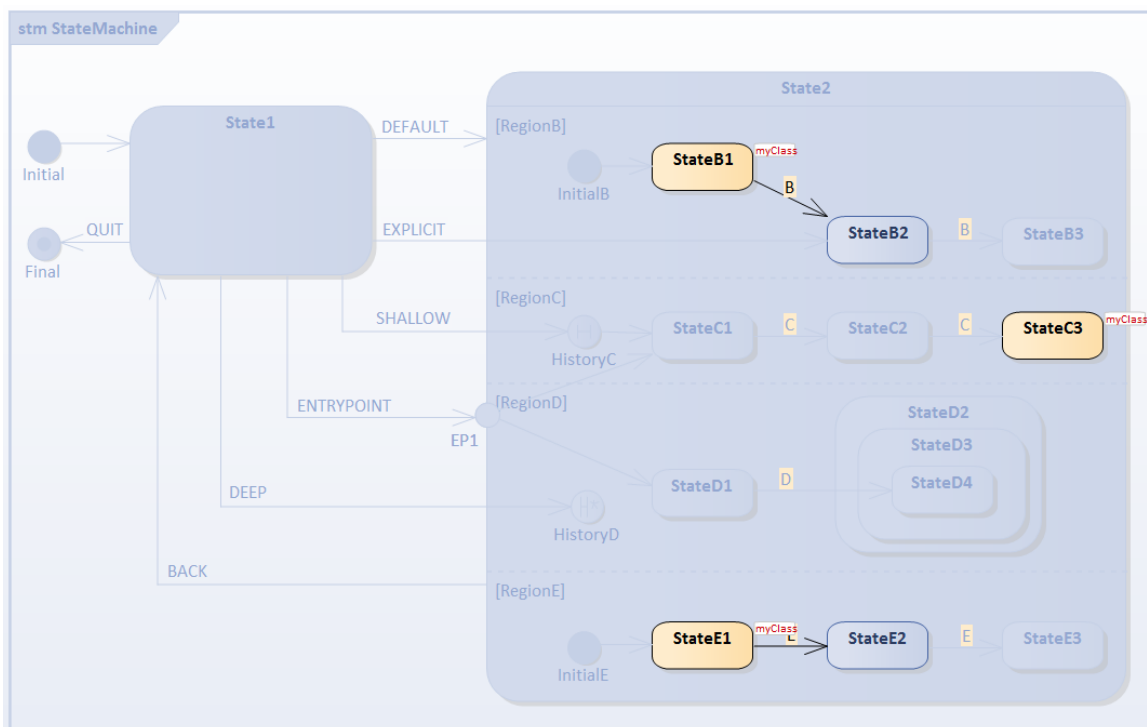
Select the Trigger [BACK] to reset.

2) Select the Explicit Entry: Trigger Sequence [EXPLICIT].
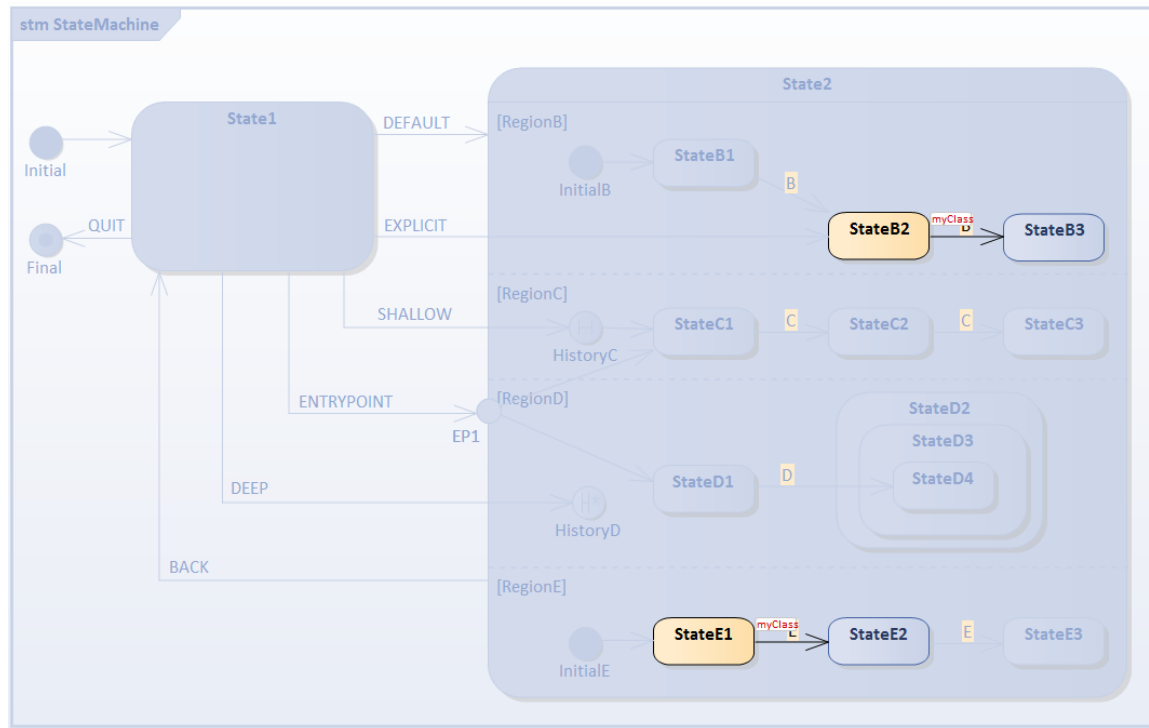
- *RegionB* is activated because the transition targets the contained vertex *StateB2*

- *RegionE* is activated because it defines *InitialE*; the transition outgoing from it will be executed, *StateE1* is the active state

- *RegionC* and *RegionD* are inactive because no Initial Pseudostates were defined

Select the Trigger [BACK] to reset.

3) Select the Default History Transition: Trigger Sequence [SHALLOW].
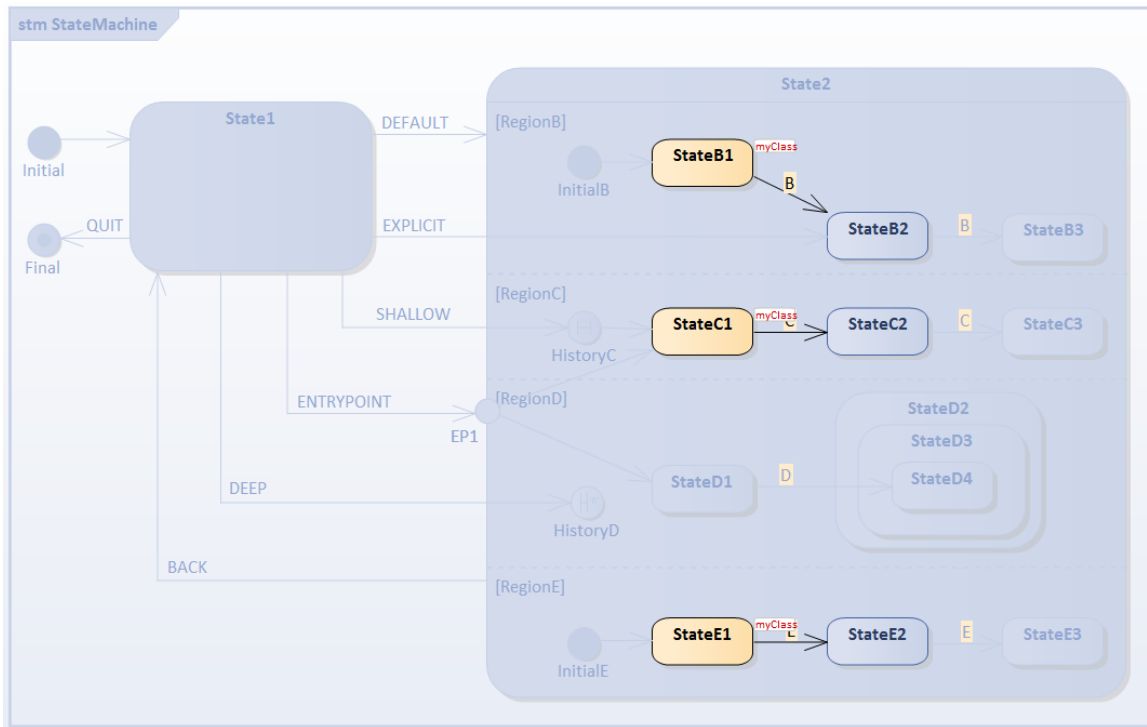
- *RegionC* is activated because the transition targets the contained vertex *HistoryC*; since this region is entered for the first time (and the History pseudostate has nothing to 'remember'), the transition outgoing from *HistoryC* to *StateC1* is executed

- *RegionB* is activated because it defines *InitialB*; the transition outgoing from it will be executed, *StateB1* is the active state

- *RegionE* is activated because it defines *InitialE*; the transition outgoing from it will be executed, *StateE1* is the active state

- *RegionD* is inactive because no Initial Pseudostate was defined

4) Prepare for testing Shallow History Entry: Trigger Sequence [C, C].

- We assume shallow history pseudostate *HistoryC* can remember *StateC3*

Select the Trigger [BACK] to reset.

5) Select the Shallow History Entry: Trigger Sequence [SHALLOW].
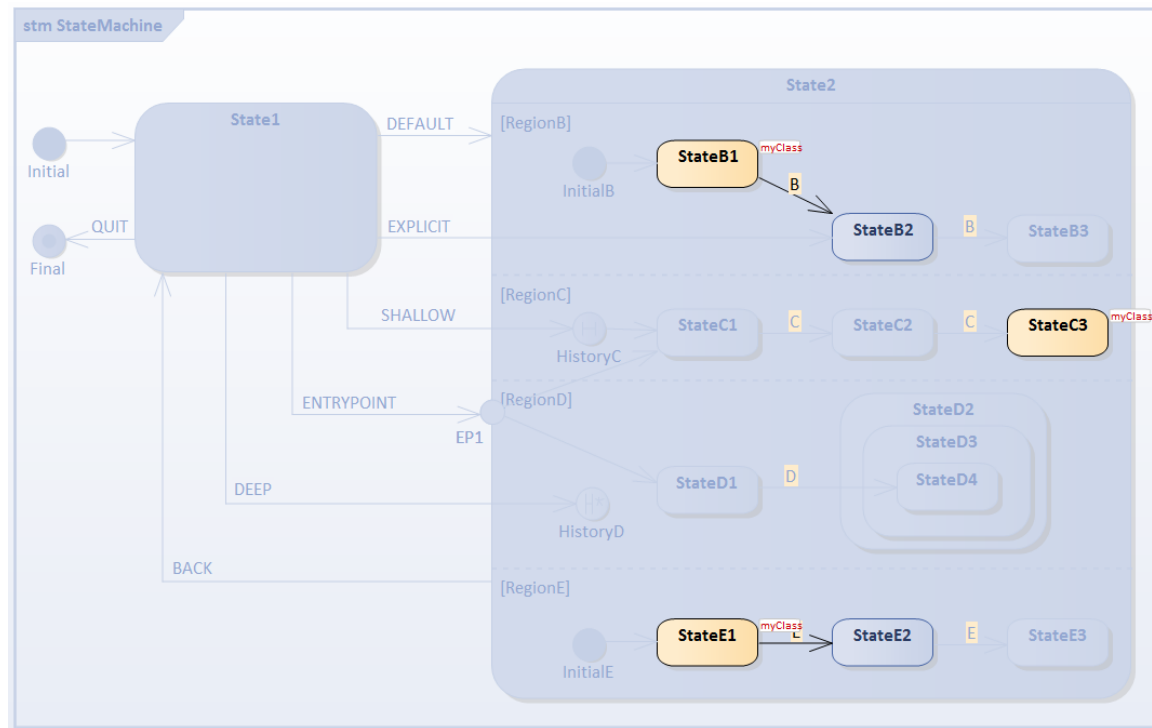
- For *RegionC*, *StateC3* is activated directly
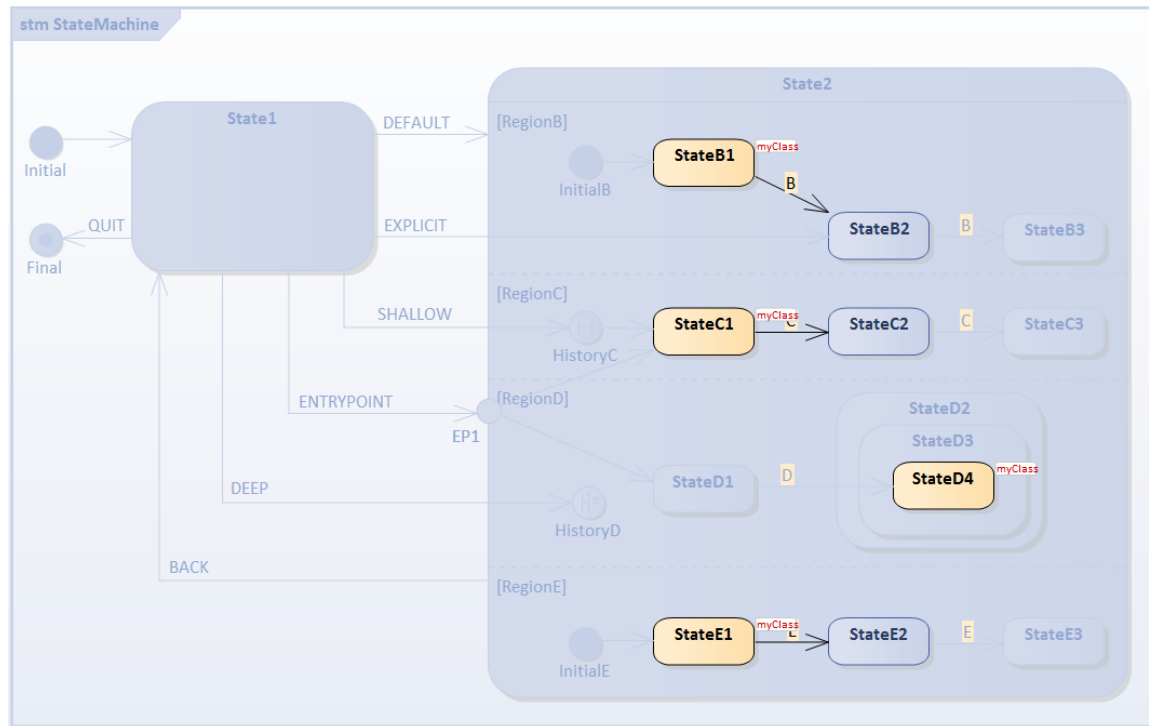
Select the Trigger [BACK] to reset.

6) Select the Entry Point Entry: Trigger Sequence [ENTRYPOINT].
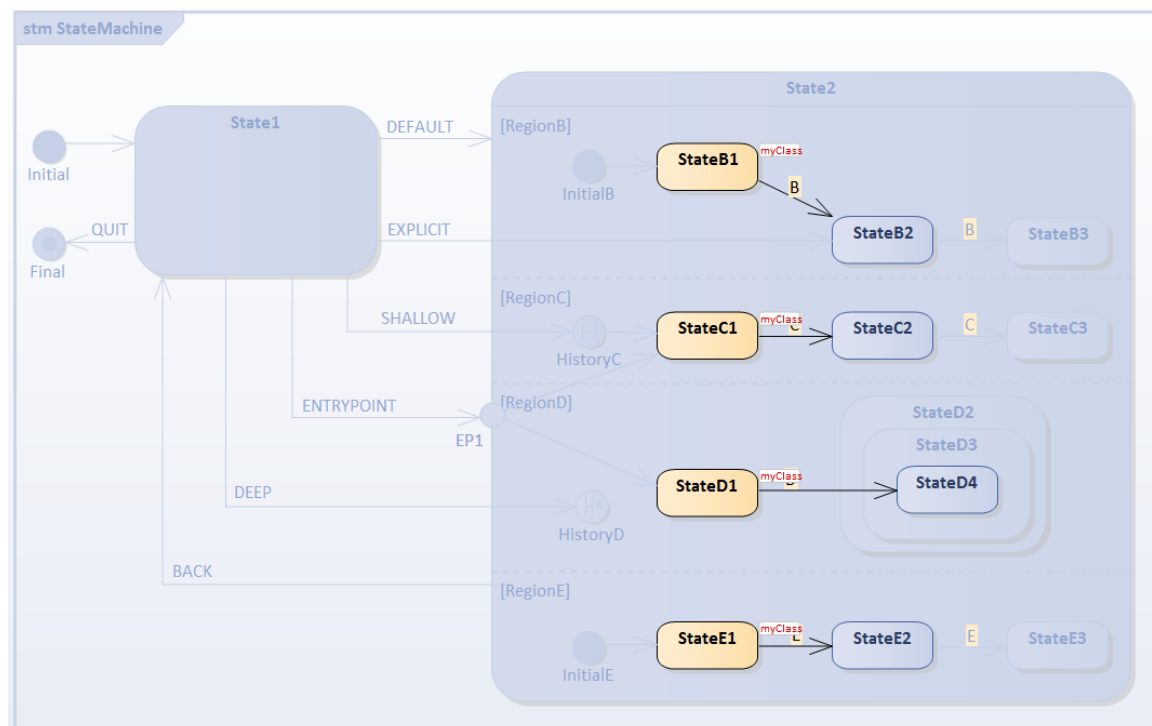
- *RegionC* is activated because the transition from *EP1* targets the contained *StateC1*
- *RegionD* is activated because the transition from *EP1* targets the contained *StateD1*

- *RegionB* is activated because it defines *InitialB*; the transition outgoing from it will be executed, *StateB1* is the active state

- *RegionE* is activated because it defines *InitialE*; the transition outgoing from it will be executed, *StateE1* is the active state

7) Prepare for testing Deep History: Trigger Sequence [D].

- We assume deep history pseudostate *HistoryD* can remember *StateD2*, *StateD3* and *StateD4*

Select the Trigger [BACK] to reset.

8) Select the Deep History Entry: Trigger Sequence [DEEP].
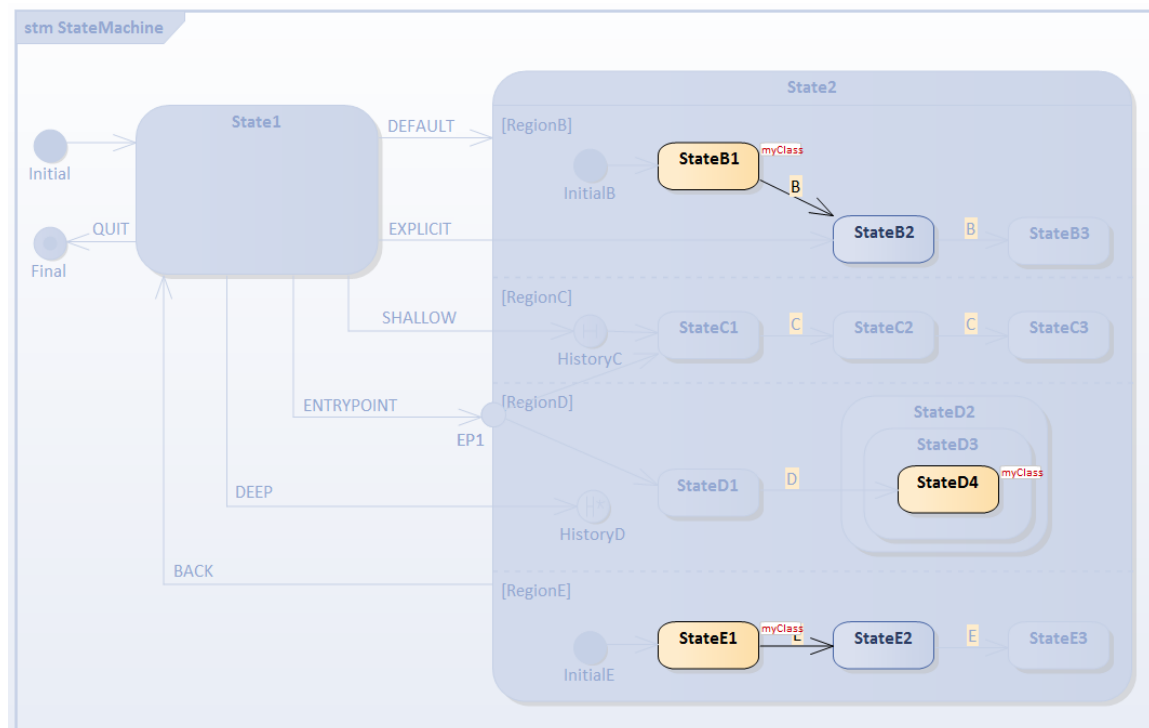
- For *RegionD*, *StateD2*, *StateD3* and *StateD4* are entered; the traces are:

- myClass[MyClass].StateMachine_State1 EXIT
- myClass[MyClass].State1__TO__HistoryD_105793_61752 Effect
- myClass[MyClass].StateMachine_State2 ENTRY
- myClass[MyClass].StateMachine_State2 DO
- myClass[MyClass].InitialE_105787__TO__StateE1_61746 Effect
- myClass[MyClass].StateMachine_State2_StateE1 ENTRY
- myClass[MyClass].StateMachine_State2_StateE1 DO
- myClass[MyClass].InitialB_105785__TO__StateB1_61753 Effect
- myClass[MyClass].StateMachine_State2_StateB1 ENTRY
- myClass[MyClass].StateMachine_State2_StateB1 DO
- myClass[MyClass].StateMachine_State2_StateD2 ENTRY
- myClass[MyClass].StateMachine_State2_StateD2_StateD3 ENTRY
- myClass[MyClass].StateMachine_State2_StateD2_StateD3_StateD4 ENTRY

Example: Fork and Join

Fork pseudostates split an incoming Transition into two or more Transitions, terminating in Vertices in orthogonal Regions of a Composite State. The Transitions outgoing from a Fork pseudostate cannot have a guard or a trigger, and the Effect behaviors of the individual outgoing Transitions are, at least conceptually, executed concurrently.

Join pseudostates are a common target Vertex for two or more Transitions originating from Vertices in different orthogonal Regions. Join pseudostates perform a synchronization function, whereby all incoming Transitions have to complete before execution can continue through an outgoing Transition.

In this example, we demonstrate the behavior of a StateMachine with Fork and Join pseudostates.

Modeling StateMachine
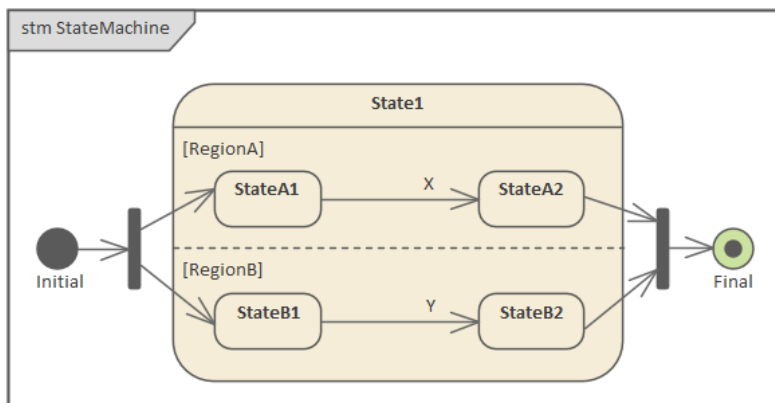
Context of StateMachine

- Create a Class element named *MyClass*, which serves as the context of a StateMachine
- Right-click on *MyClass* in the Browser window and select the 'Add | StateMachine' option
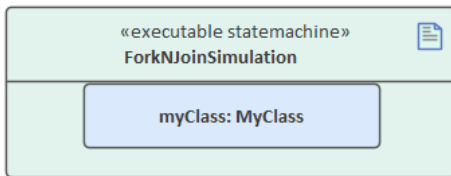
StateMachine

- Add an *Initial* Node, a *Fork*, a State named *State1*, a *Join*, and a *Final* to the diagram
- Enlarge *State1*, right-click on it on the diagram and select the 'Advanced | Define Concurrent Substates | Define' option and define *RegionA* and *RegionB*
- In *RegionA*, define *StateA1*, transition to *StateA2*, triggered by event *X*
- In *RegionB*, define *StateB1*, transition to *StateB2*, triggered by event *Y*
- Draw other transitions: *Initial* to *Fork*; *Fork* to *StateA1* and *StateB1*; *StateA2 and StateB2* to *Join*; *Join* to *Final*

Simulation

Artifact

Enterprise Architect supports C, C++, C#, Java and JavaScript; we will use JavaScript in this example because we don't need to install a compiler (for the other languages, either Visual Studio or JDK are required).

- From the Diagram Toolbox select the 'Simulation' page and drag the Executable StateMachine icon onto the diagram to create an Artifact; name it *ForkNJoinSimulation* and set its 'Language' field to 'JavaScript'
- Ctrl+Drag *MyClass* from the Browser window and drop it on the *ForkNJoinSimulation* Artifact as a Property; give it the name *myClass*

Code Generation

- Click on *ForkNJoinSimulation* and select the 'Simulate > Executable States > Statemachine > Generate, Build and Run' ribbon option
- Specify a directory for the generated source code

Note: The contents of this directory will be cleared before generation; make sure you point to a directory that exists only for StateMachine simulation purposes.
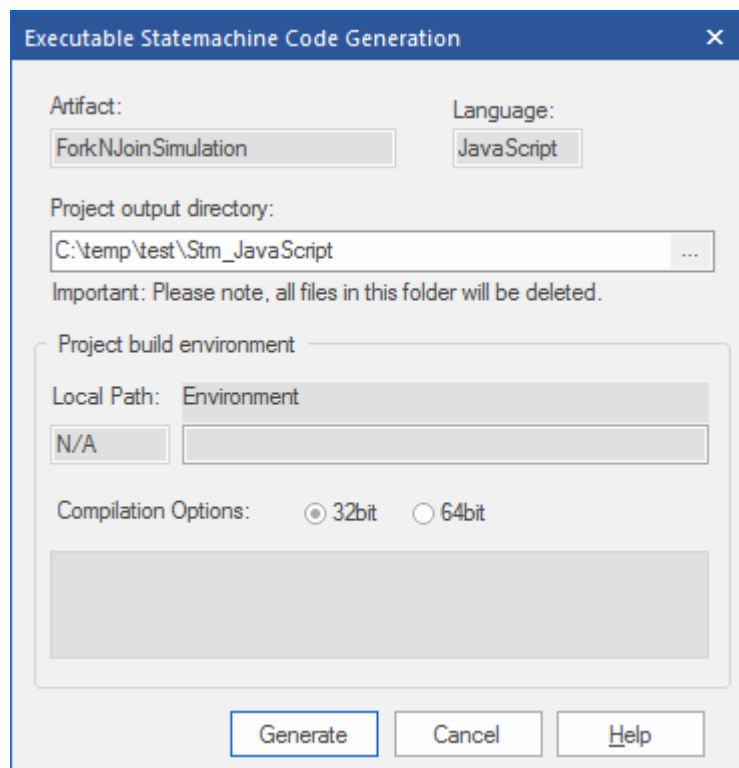

Run Simulation

When the simulation is started, *State1*, *StateA1* and *StateB1* are active and the StateMachine is waiting for events.
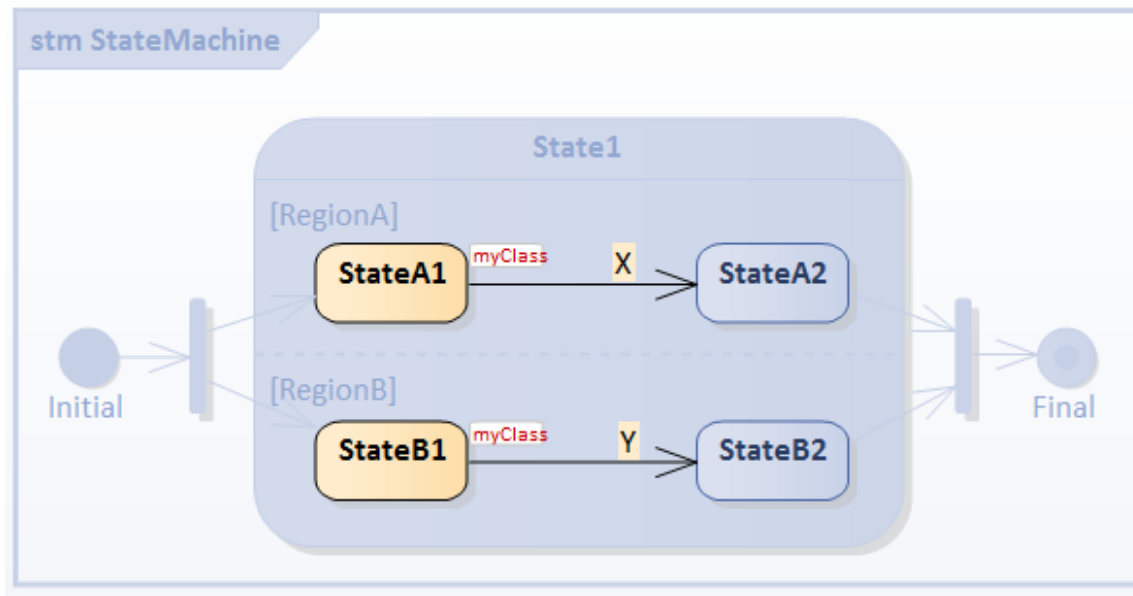
Select the 'Simulate > Dynamic Simulation > Events' ribbon option to display the Simulation Events window.

On Trigger event *X*, *StateA1* will exit and enter *StateA2*; after the entry and doActivity behavior has run, the completion events of *StateA2* are dispatched and recalled. Then the transition from *StateA2* to the *Join* pseudostate is enabled and traversed.

Note: *Join* must wait for all incoming Transitions to complete before execution can continue through an outgoing Transition. Since the branch from *RegionB* is not complete (because *StateB1* is still active and waiting for triggers) the transition from *Join* to *Final* will not be executed at this moment.
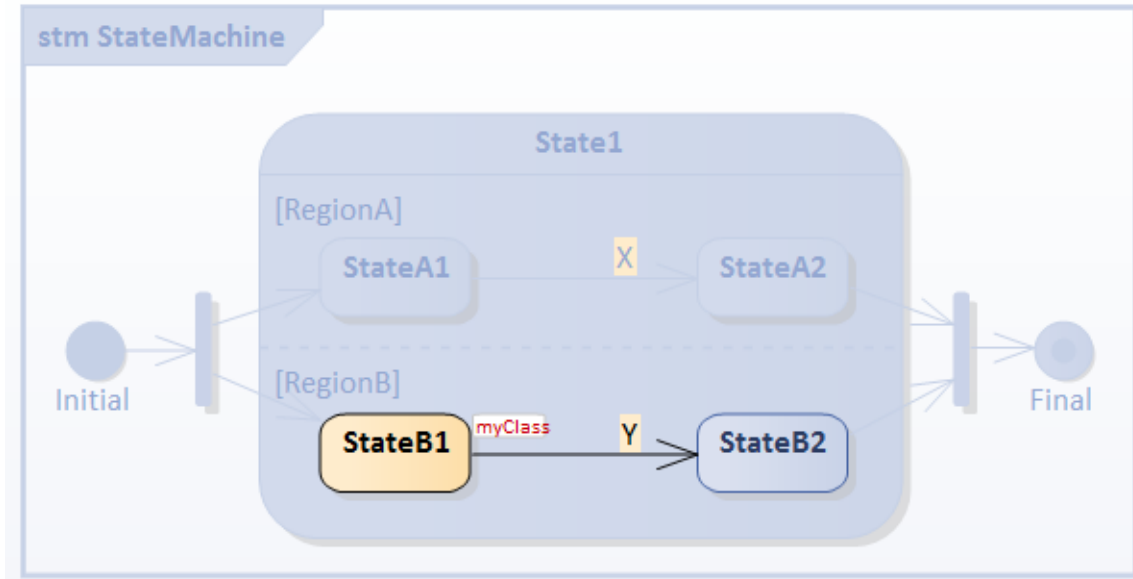
On Trigger event *Y*, *StateB1* will exit and enter *StateB2*; after the entry and doActivity behavior has run, completion events of *StateB2* are dispatched and recalled. Then the transition from *StateB2* to the *Join* pseudostate is enabled and traversed. This satisfies the criteria of all the incoming transitions of *Join* having completed, so the transition from *Join* to *Final* is executed. Simulation has ended.

*Tips: Y*ou can view the execution trace sequence from the Simulation window ('Simulate > Dynamic Simulation > Simulator > Open Simulation Window' ribbon option).

    myClass[MyClass].Initial_82285__TO__fork_82286_82286_61745 Effect

myClass[MyClass].StateMachine_State1 ENTRY

myClass[MyClass].StateMachine_State1 DO

myClass[MyClass].fork_82286_82286__TO__StateA1_57125 Effect

myClass[MyClass].StateMachine_State1_StateA1 ENTRY

myClass[MyClass].StateMachine_State1_StateA1 DO

myClass[MyClass].fork_82286_82286__TO__StateB1_57126 Effect

myClass[MyClass].StateMachine_State1_StateB1 ENTRY

myClass[MyClass].StateMachine_State1_StateB1 DO

Trigger X

myClass[MyClass].StateMachine_State1_StateA1 EXIT

myClass[MyClass].StateA1__TO__StateA2_57135 Effect

myClass[MyClass].StateMachine_State1_StateA2 ENTRY

myClass[MyClass].StateMachine_State1_StateA2 DO

myClass[MyClass].StateMachine_State1_StateA2 EXIT

myClass[MyClass].StateA2__TO__join_82287_82287_57134 Effect

Trigger Y

myClass[MyClass].StateMachine_State1_StateB1 EXIT

myClass[MyClass].StateB1__TO__StateB2_57133 Effect

myClass[MyClass].StateMachine_State1_StateB2 ENTRY

myClass[MyClass].StateMachine_State1_StateB2 DO

myClass[MyClass].StateMachine_State1_StateB2 EXIT

myClass[MyClass].StateB2__TO__join_82287_82287_57132 Effect

myClass[MyClass].StateMachine_State1 EXIT

myClass[MyClass].join_82287_82287__TO__Final_105754_57130 Effect

Example: 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:

1. Select 'Simulate > Dynamic Simulation > Simulator > Open Simulation Window'. Also select 'Simulate > Dynamic Simulation > Events' to open the Simulation Events window.

2. The Simulator Events window helps you to trigger events; double-click on a trigger in the 'Waiting Triggers' column.

3. 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; the output might resemble this:
   24850060]   Event Pool: [NEW,NEW,NEW,NEW,NEW,]

Deferred Event Example

This example shows a model using Deferred Events, and the Simulation Events window showing all available Events.

We firstly set up the contexts (the Class elements containing the StateMachines), simulate them in a simple context and raise the event from outside it; then simulate in a client-server context with the Send event mechanism.

Create Context and StateMachine

Create the server context
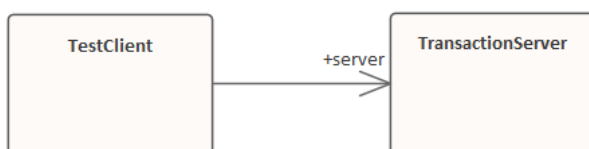

Create a Class diagram and:

5. A Class element *TransactionServer*, to which you add a StateMachine *ServerStateMachine*.

6. A Class element *TestClient*, to which you add a StateMachine *ClientStateMachine*.

7. An Association from *TestClient* to *TransactionServer*, with the target role named *server*.

Modeling for *ServerStateMachine*
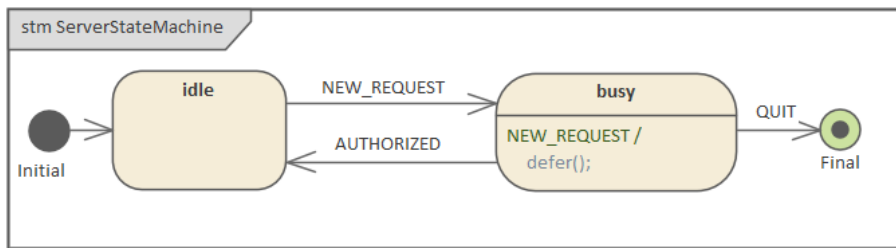
4. Add an Initial Node *Initial* to the StateMachine diagram, and transition to a State *idle*.
5. Transition (with event NEW_REQUEST as Trigger) to a State *busy*.
6. Transition (with event QUIT as Trigger) to a Final State *Final*.
7. Transition (with event AUTHORIZED as Trigger) to *idle*.
8. Transition (with event NEW_REQUEST as Trigger and *defer();* as effect) to *busy*

Modeling for *ClientStateMachine*
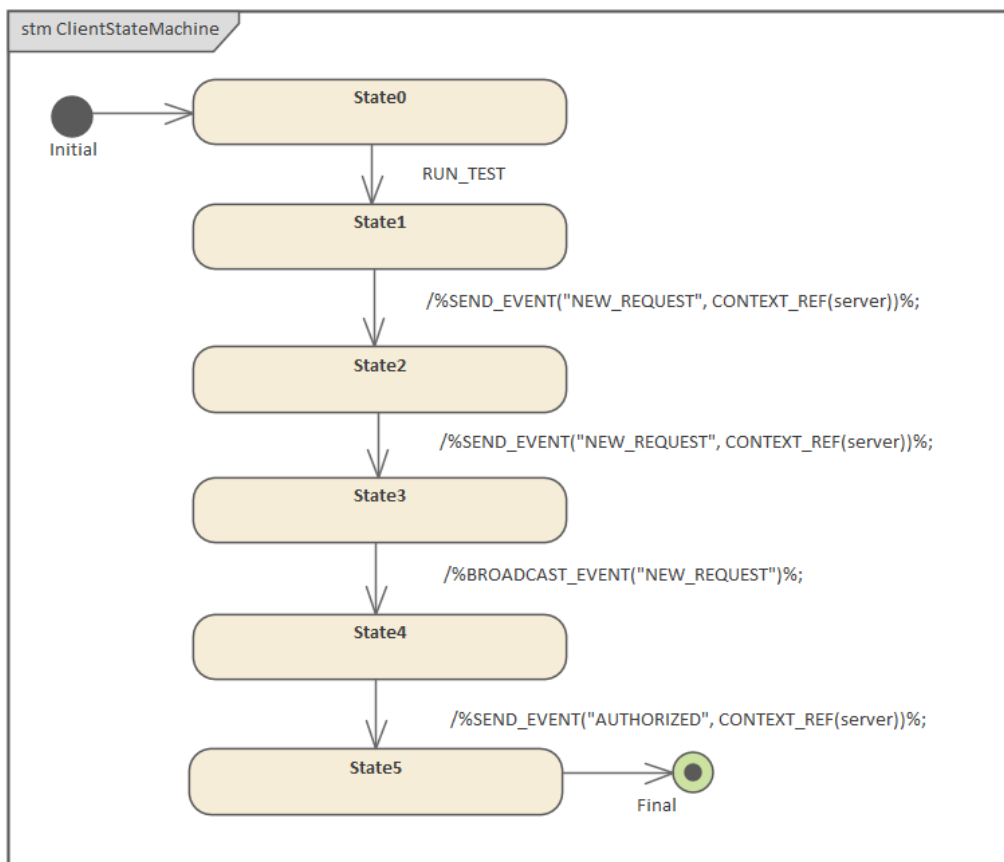
1. Add an Initial Node *Initial* to the StateMachine diagram, and transition to a State *State0*.
2. Transition (with event RUN_TEST as trigger) to a State *State1*.
3. Transition (with effect: %SEND_EVENT("NEW_REQUEST", CONTEXT_REF(server))%;) to a State *State2*.
4. Transition (with effect: %SEND_EVENT("NEW_REQUEST", CONTEXT_REF(server))%;) to a State *State3*.
5. Transition (with effect: %BROADCAST_EVENT("NEW_REQUEST")%;) to a State *State4*.
6. Transition (with effect: %SEND_EVENT("AUTHORIZED", CONTEXT_REF(server))%;) to a State *State5*.

7.   Transition to a Final State *Final.*

Simulation in a simple context

Create the Simulation Artifact
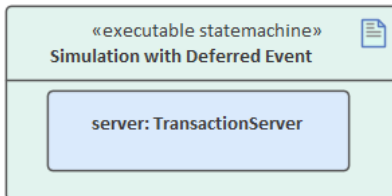
1.   Create an Executable StateMachine Artifact with the name *Simulation with Deferred Event* and the 'Language' field set to *JavaScript.*

2.   Enlarge it, then Ctrl+drag the *TransactionServer* element onto the Artifact and paste it as a property with the name *server.*
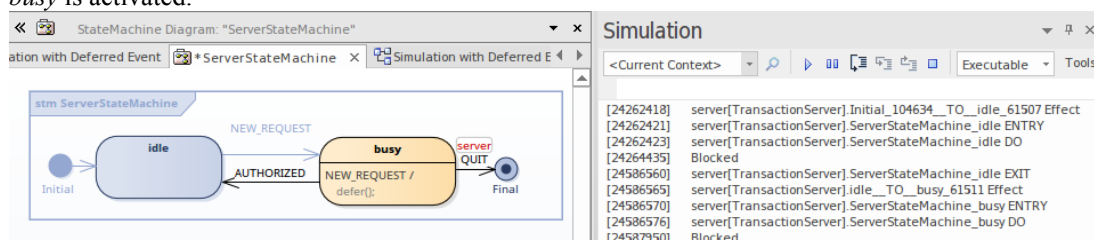
Run the Simulation

1.   Select the Artifact, then select the 'Simulate > Executable States > Statemachine > Generate, Build and Run' option, and specify a directory for your code (Note: all the files in the directory will be deleted before simulation starts).

2.   Click on the Generate button.

3.   Select the 'Simulate > Dynamic Simulation > Events' option to open the Simulation Event window.
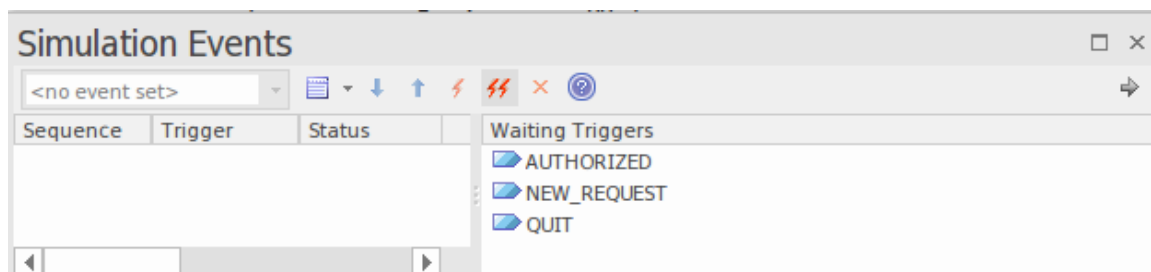
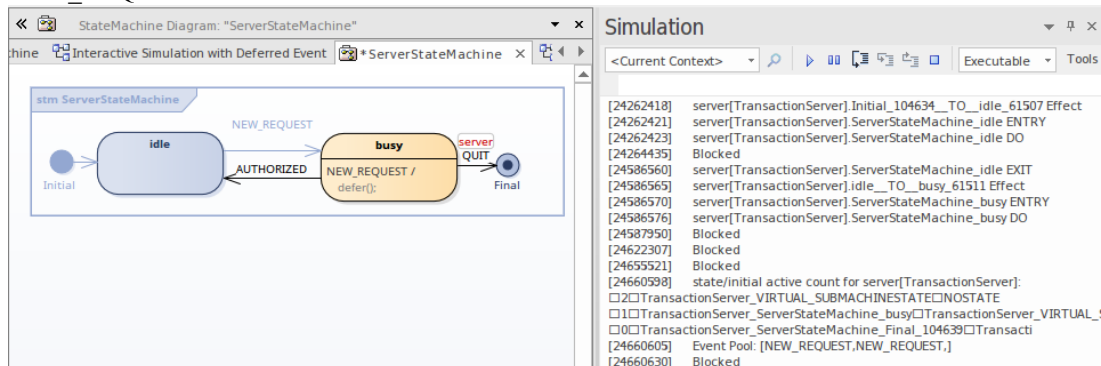When simulation starts, *idle* will be the active state.
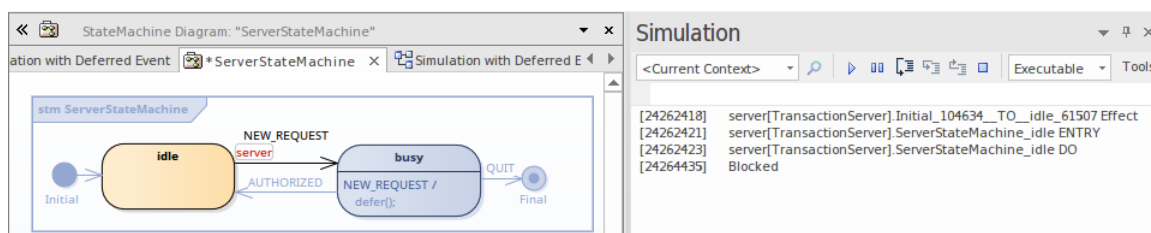
1.   Double-click on NEW_REQUEST in the Simulation Event window to execute it as the Trigger; *idle* is exited and *busy* is activated.


2.   Double-click on NEW_REQUEST in the Simulation Event window to execute it again as the Trigger; *busy* remains

activated, and an instance of NEW_REQUEST is appended in the Event Pool.

3.   Double-click on NEW_REQUEST in the Simulation Event window to execute it a third time as the Trigger; *busy* remains activated, and an instance of NEW_REQUEST is appended in the Event Pool.

4.   Type *dump* in the Simulation window command line; notice that the event pool has two instances of NEW_REQUEST.


5.   Double-click on AUTHORIZED in the Simulation Event window to execute it as the Trigger; these actions take place:
       - *busy* is exited and *idle* becomes active
       - a NEW_REQUEST event is retrieved from the pool, *idle* is exited and *busy* becomes active

6.   Type *dump* in the Simulation window command line; there is now only one instance of NEW_REQUEST in the Event Pool.
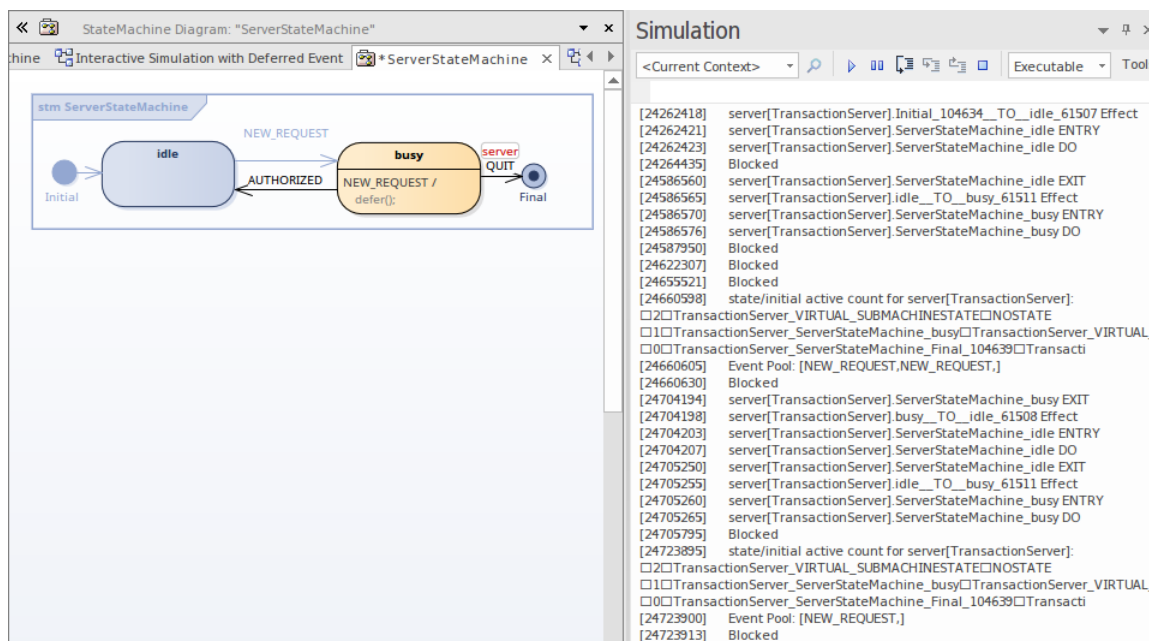
Interactive simulation via Send/Broadcast Event

Create the Simulation Artifact
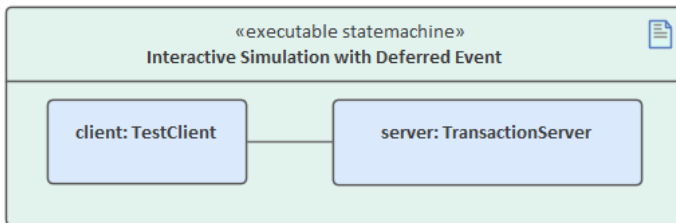
3.   Create an Executable StateMachine Artifact with the name *Interactive Simulation with Deferred Event* and the 'Language' field set to *JavaScript;* enlarge the element.

4.   Ctrl+Drag the *TransactionServer* element onto the Artifact, and paste it as a property with the name *server.*

5.   Ctrl+Drag the *TestClient* element onto the Artifact, and paste it as a property with the name *client.*

6.   Create a connector from *client* to *server.*

7.   Click on the connector and press Ctrl+L to select the association from the *TestClient* element to the *TransactionServer* element.

Run Interactive Simulation

4.   Launch the simulation in the same way as for the simple context.

     Once the simulation has started, the *client* remains at *State0* and the *server* remains at *idle.*
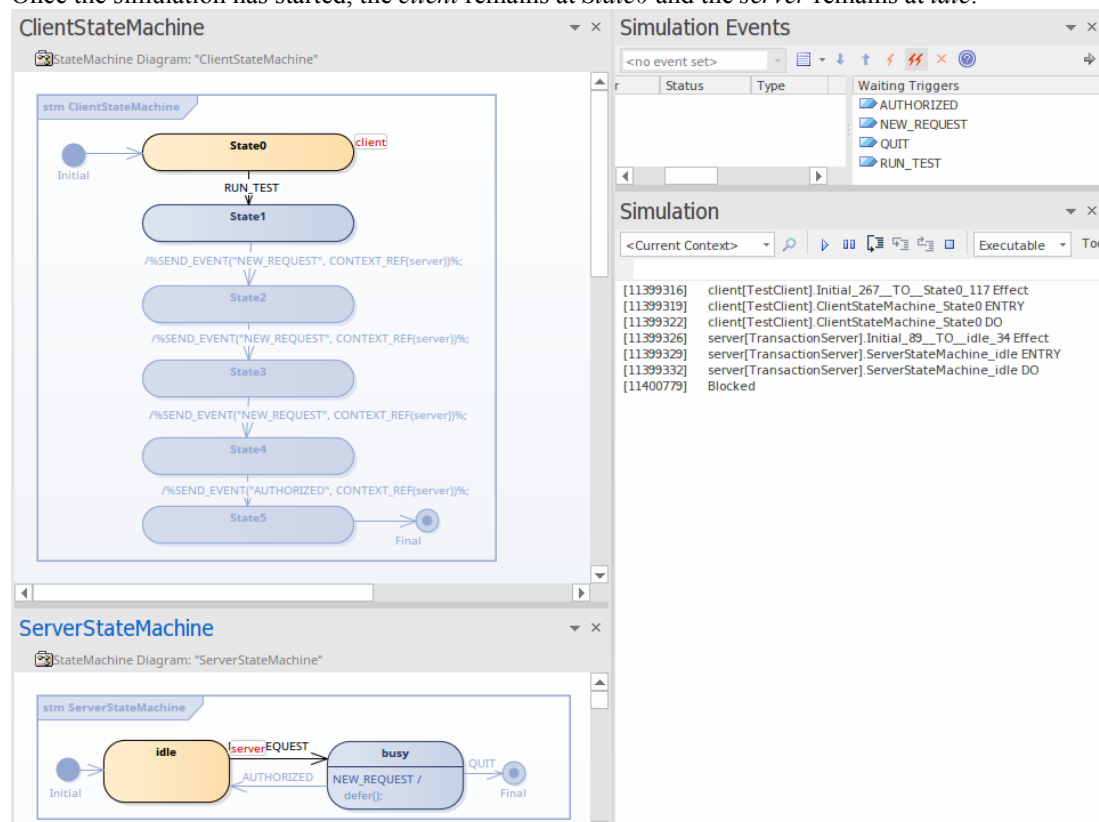

5.   Double-click on RUN_TEST in the Simulation Event window to trigger it. The event NEW_REQUEST will be triggered three times (by SEND_EVENT and BROADCAST_EVENT) and AUTHORIZED will be triggered once by SEND_EVENT.
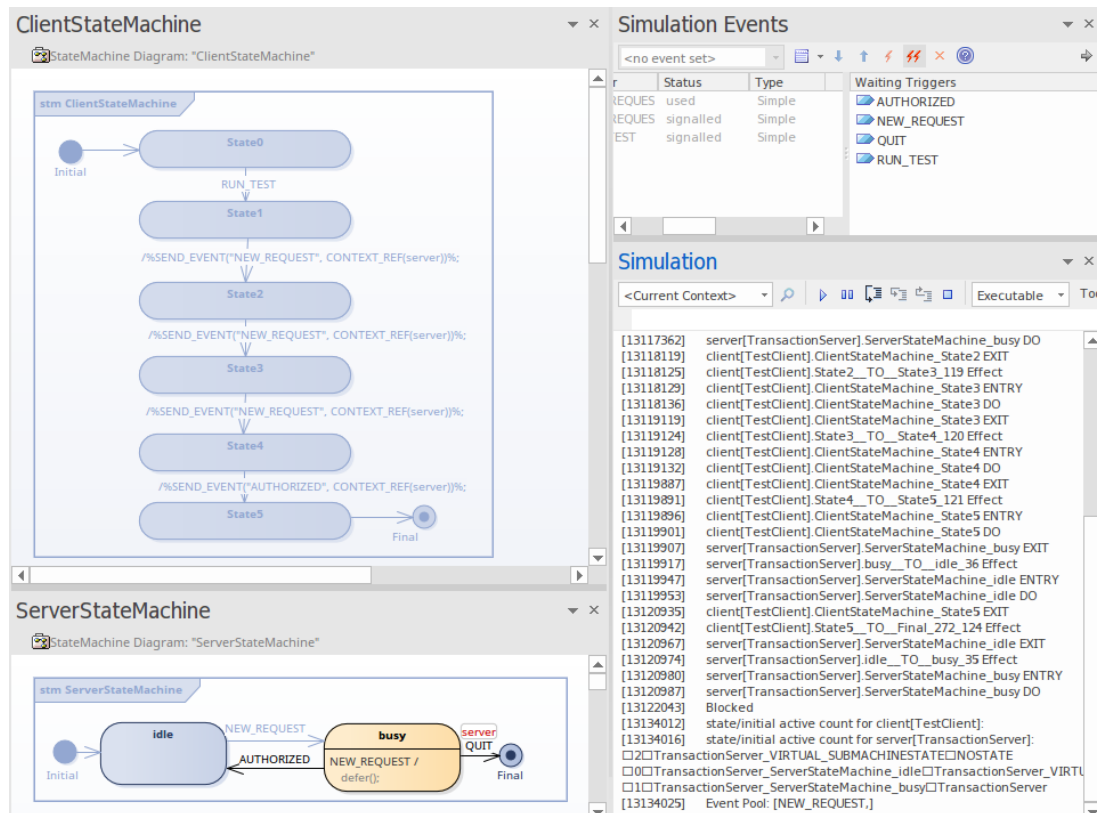
Type *dump* in the Simulation window command line. There is one instance of NEW_REQUEST left in the Event Pool. The result matches our manual triggering test.

Example: Entry and Exit Points (Connection Point References)

Enterprise Architect provides support for Entry and Exit points, and for Connection Point References. In this example, we define two StateMachines for *MyClass - StateMachine* and *SubMachine*.
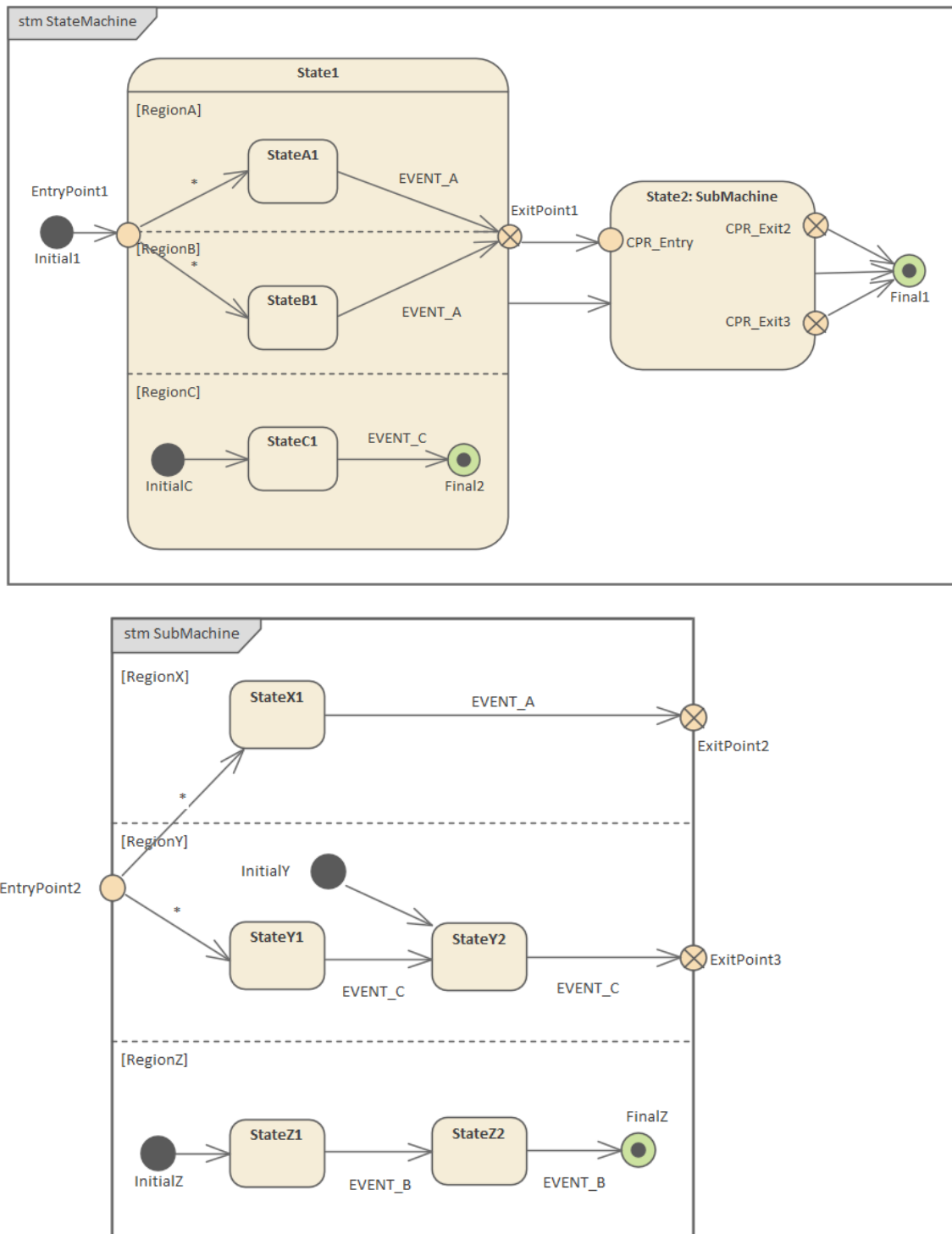
- *State1* is a Composite State (also called an Orthogonal State because it has multiple Regions) with three Regions:

RegionA, RegionB and RegionC

- State2 is a SubMachine State calling SubMachine, which has three Regions: RegionX, RegionY, and RegionZ
- EntryPoint1 is defined on State1 to activate two of the three Regions; EntryPoint2 is defined on SubMachine to activate two of the three Regions
- ExitPoint1 is defined on State1; two exit points ExitPoint2 and ExitPoint3 are defined on SubMachine
- Connection Point References are defined on State2 and bind to the Entry/Exit Points of the typing SubMachine
- Initial nodes are defined to demonstrate default activation of the Regions

Entering a State: Entry Point Entry

EntryPoint1 on State1

When a Transition targeted on EntryPoint1 is enabled, State1 is activated followed by the contained Regions.
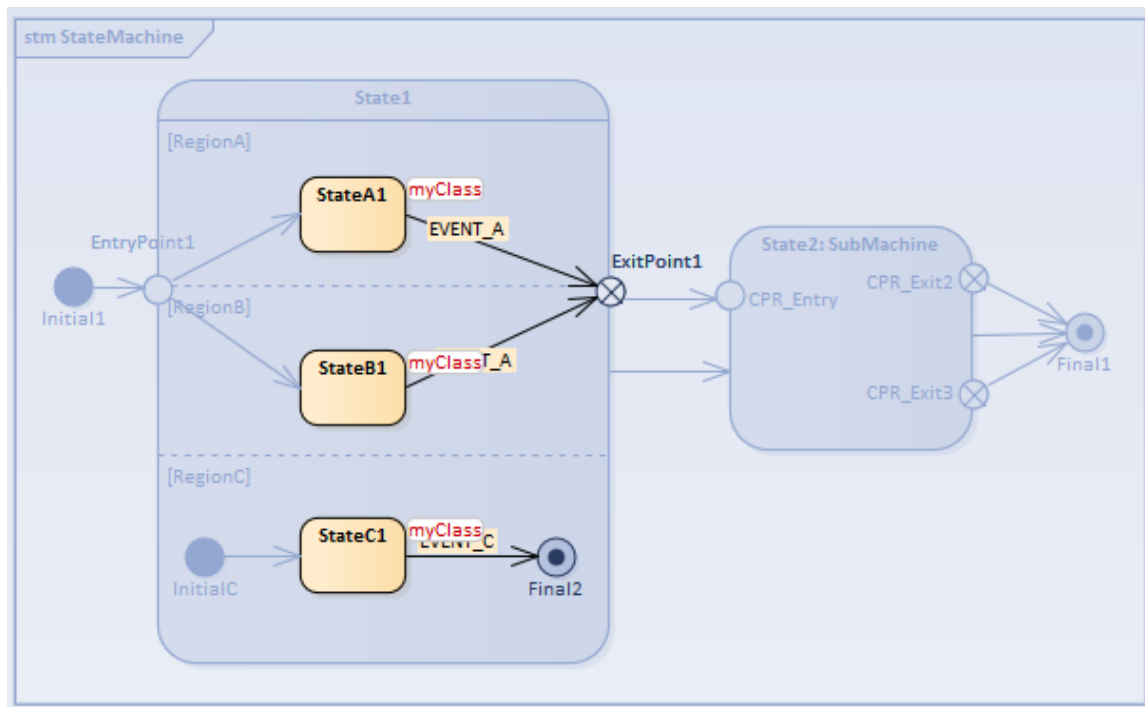
- Explicit activation occurs for RegionA and RegionB, because each of them is entered by a Transition terminating on one of the Region's contained Vertices
- Default activation occurs for RegionC, because it defines an Initial pseudostate InitialC and the Transition originating from the InitialC to StateC1 starts execution

EntryPoint2 on SubMachine

The Trigger Sequence to be simulated is: [EVENT_C, EVENT_A].

When a Transition targeted on Connection Point Reference CPR_Entry on State2 is enabled, State2 is activated, followed by the SubMachine's activation through the binding entry points.
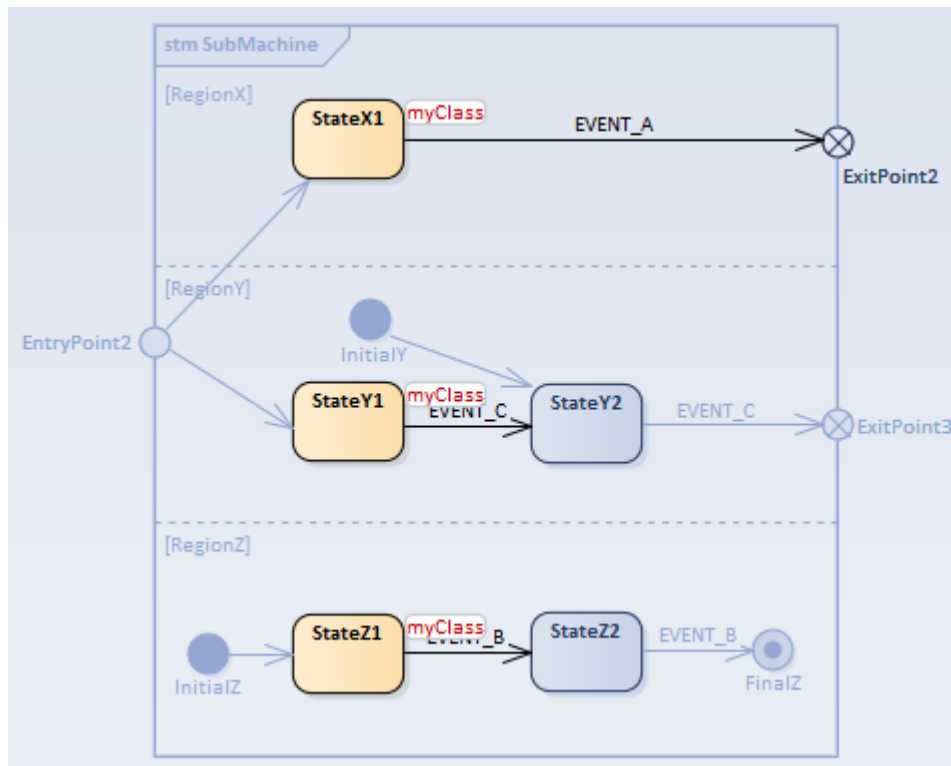
- Explicit activation occurs for *RegionX* and *RegionY*, because each of them is entered by a Transition terminating on one of the Region's contained Vertices - *StateX1* in *RegionX*, *StateY1* in *RegionY*

- Default activation occurs for *RegionZ*, because it defines an Initial pseudostate *InitialZ* and the Transition originating from *InitialZ* to *StateZ1* starts execution
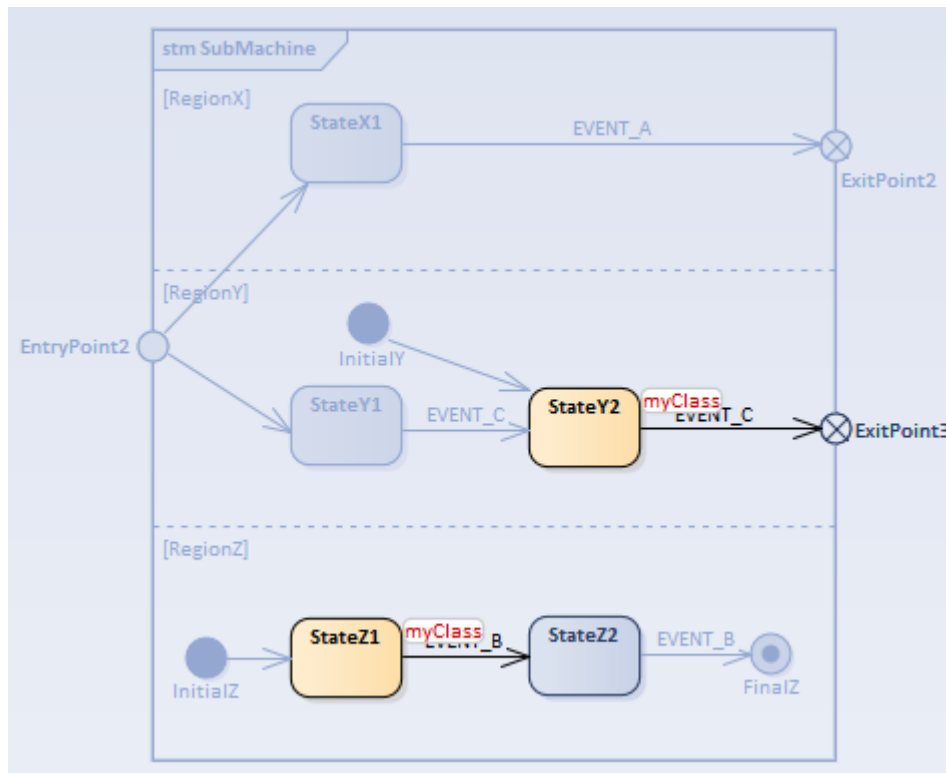
Entering a State: Default Entry

This situation arises when the Composite State is the direct target of a Transition.

Default Entry of *State2*

The Trigger Sequence to be simulated is: [EVENT_A, EVENTC].

When a Transition targeted directly on *State2* is enabled, *State2* is activated, followed by default activation for all the Regions of the SubMachine.

- *RegionX's* State is inactive because it does not define an Initial node
- *RegionY* is activated through *InitialY* and the Transition to *StateY2* is executed
- *RegionZ* is activated through *InitialZ* and the Transition to *StateZ1* is executed

State Exit

State1 Exit

- Trigger Sequence [EVENT_C, EVENT_A]: *RegionC* is inactivated first, then *RegionA* and *RegionB*; after the exit behavior of *State1* is executed, the Transition outgoing from *ExitPoint1* is enabled
- Trigger Sequence [EVENT_A, EVENT_C]: *RegionA* and *RegionB* are inactivated first, then *RegionC*; after the exit behavior of *State1* is executed, the Transition outgoing directly from *State1* is enabled

State2 Exit

Trigger Sequence [EVENT_C, EVENT_A], so the current state resembles this:
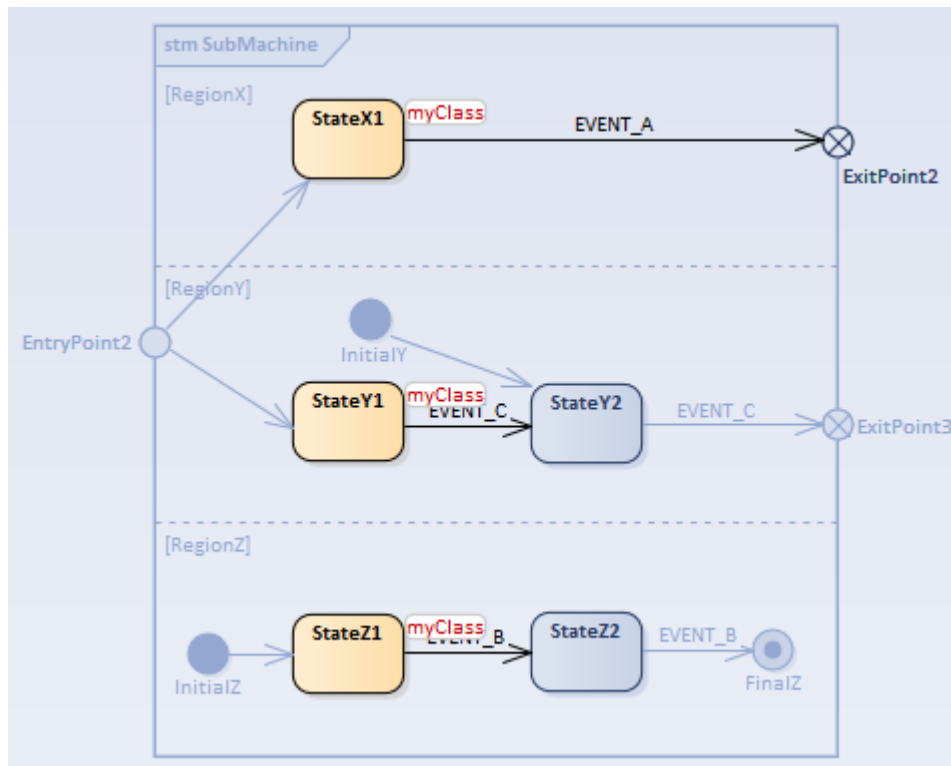
- Trigger Sequence [EVENT_A, EVENT_C, EVENT_C, EVENT_B, EVENT_B]: *RegionX* is inactivated first, then *RegionY*, and *RegionZ* is the last; after the exit behavior of *State2* is executed, the Transition outgoing directly from *State2* is enabled

- Trigger Sequence [EVENT_A, EVENT_B, EVENT_B, EVENT_C, EVENT_C]: *RegionX* is inactivated first, then *RegionZ*, and *RegionY* is the last; after the exit behavior of *State2* is executed, the Transition outgoing from *CPR_Exit3* is enabled (*ExitPoint3* on *SubMachine* is bound to *CPR_Exit3* of *State2*)

- Trigger Sequence [EVENT_C, EVENT_C, EVENT_B, EVENT_B, EVENT_A]: *RegionY* is inactivated first, then *RegionZ*, and *RegionX* is the last; after the exit behavior of *State2* is executed, the Transition outgoing from *CPR_Exit2* is enabled (*ExitPoint2* on *SubMachine* is bound to *CPR_Exit2* of *State2*)

Example: History Pseudostate

State History is a convenient concept associated with Regions of Composite States, whereby a Region keeps track of the configuration a State was in when it was last exited. This allows easy return to that State configuration, if necessary, when the Region next becomes active (for example, after returning from handling an interrupt), or if there is a local Transition that returns to its history.

Enterprise Architect supports two types of History Pseudostate:

- Deep History - representing the full State configuration of the most recent visit to the containing Region; the effect is the same as if the Transition terminating on the deepHistory Pseudostate had, instead, terminated on the innermost State of the preserved State configuration, including execution of all entry Behaviors encountered along the way

- Shallow History - representing a return to only the top-most substate of the most recent State configuration, which is entered using the default entry rule

In this example, the Classes *DeepTurbineManager* and *ShallowTurbineManager* are exactly the same except that the contained StateMachine for the first has a deepHistory Pseudostate and for the second has a shallowHistory Pseudostate.

Both StateMachines have three Composite States: *Turbine_01*, *Turbine_02* and *Turbine_03*, each of which has *Off* and *On* States and a History Pseudostate in its Region.

In order to better observe the difference between Deep History and Shallow History, we execute the two StateMachines in one simulation.
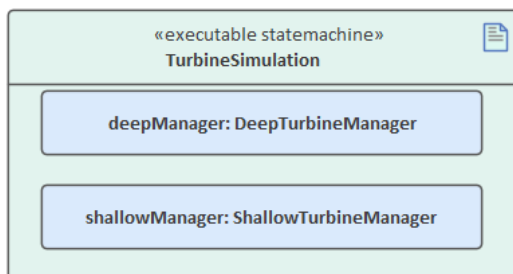
The StateMachine in *DeepTurbineManager* is illustrated in this diagram:
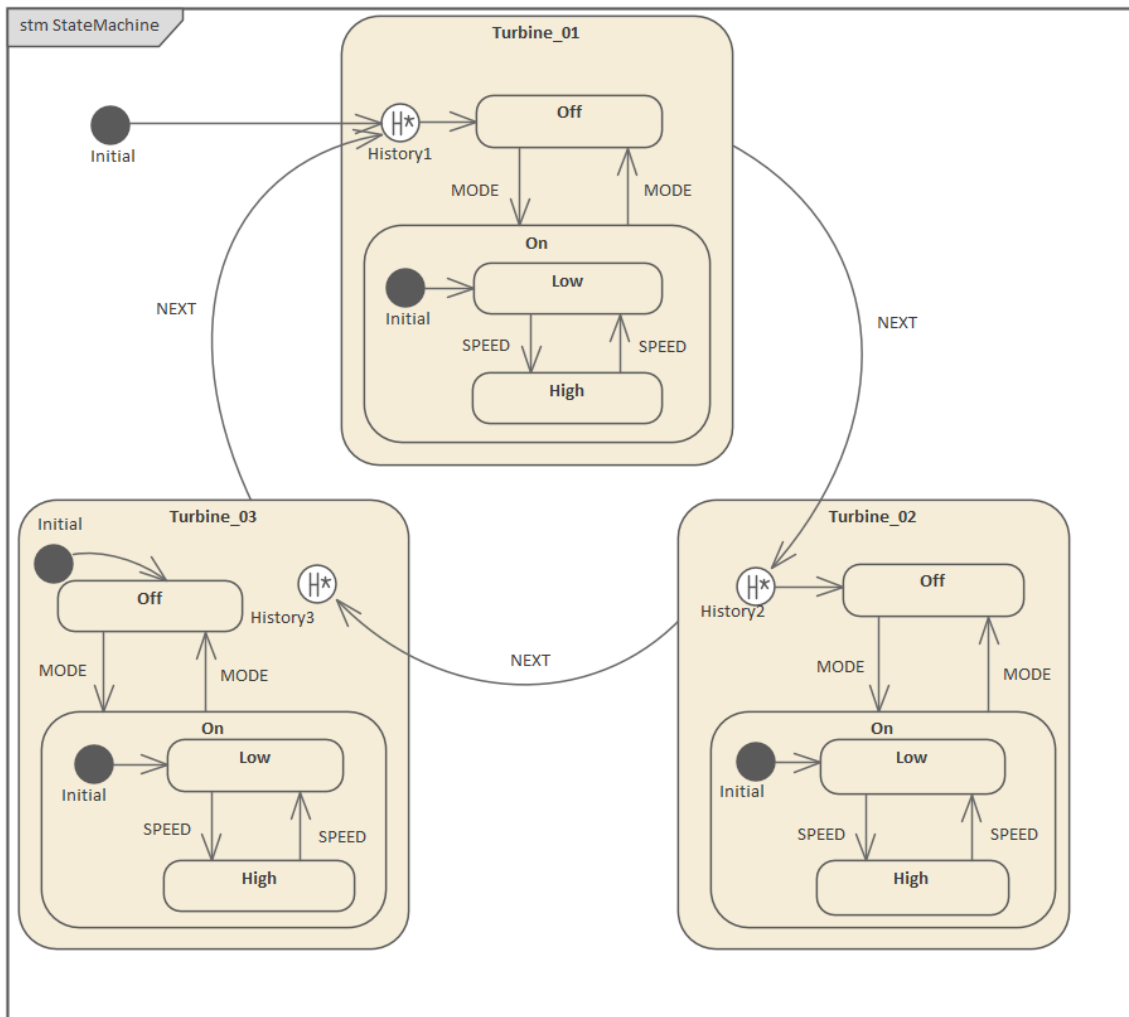
The StateMachine in *ShallowTurbineManager* is illustrated in this diagram:
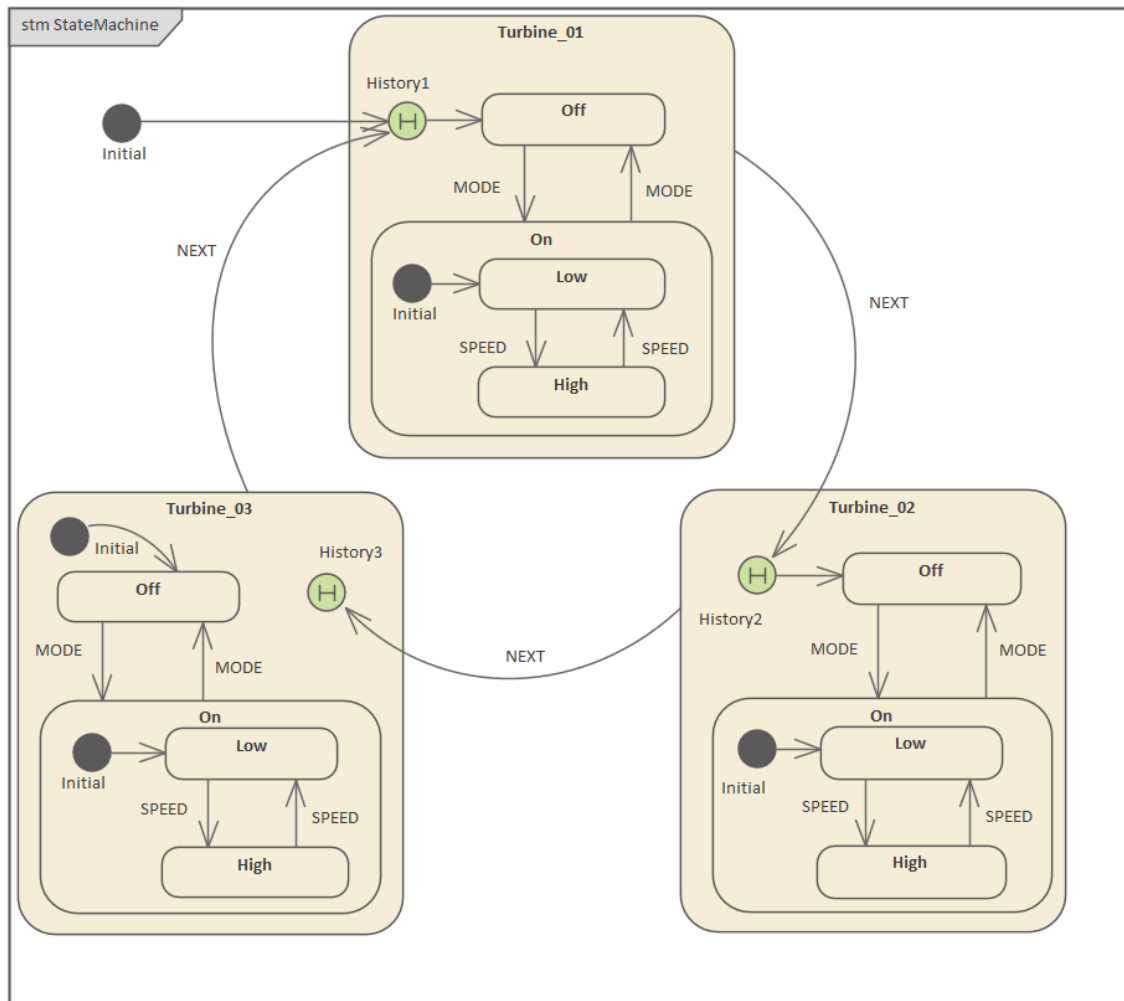
Tip: If you right-click on the History node on the diagram and select the 'Advanced | Deep History' option, you can toggle the type of History Pseudostate between shallow and deep.

First Time Activation of States

After simulation starts, *Turbine_01* and its substate *Off* are activated.

Trigger Sequence: [MODE, SPEED]
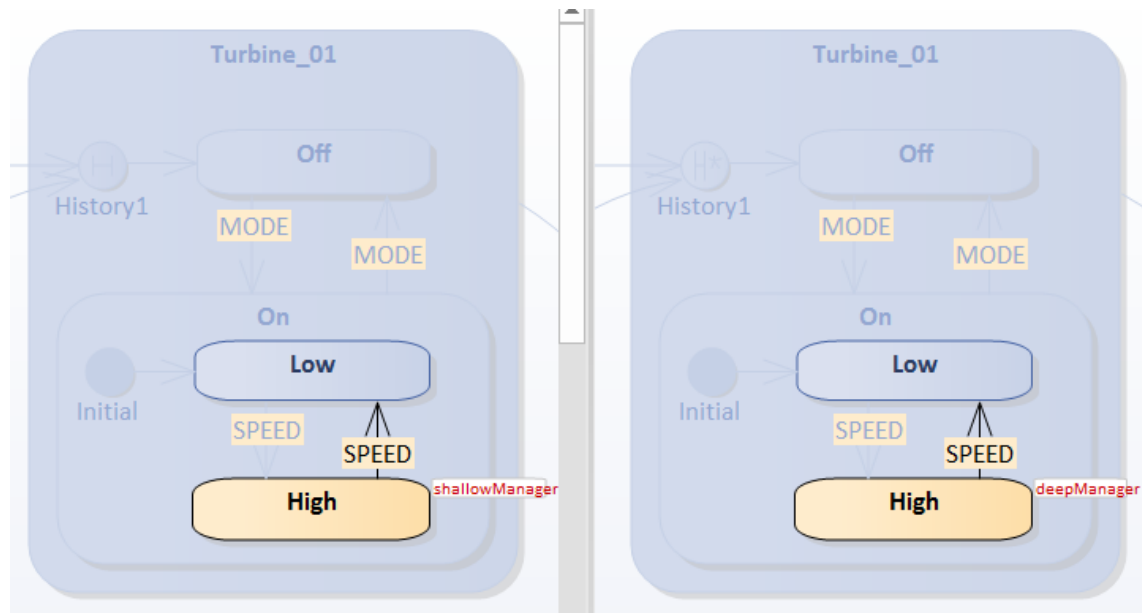
Then the active State configuration includes:

- Turbine_01
- Turbine_01.On
- Turbine_01.On.High

This applies to both *deepManager* and *shallowManager.*
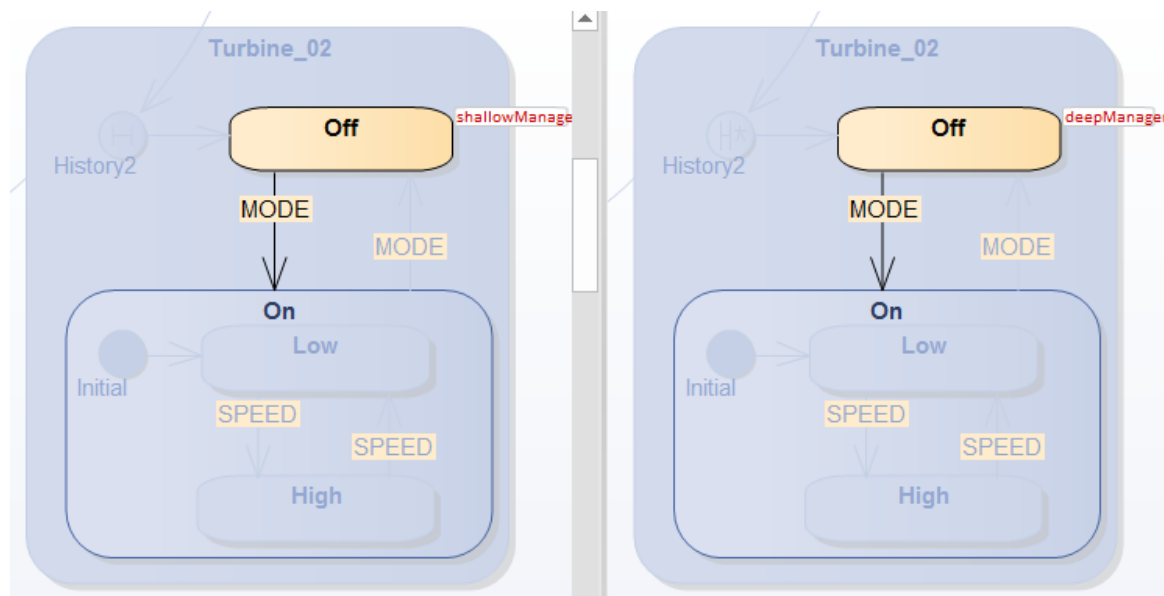
Trigger Sequence: [NEXT]

This trace sequence can be observed from the Simulation window (Simulate > Dynamic Simulation > Simulator > Open Simulation Window):

01   shallowManager[ShallowTurbineManager].StateMachine_Turbine_01_On_High EXIT

02   shallowManager[ShallowTurbineManager].StateMachine_Turbine_01_On EXIT

03   shallowManager[ShallowTurbineManager].StateMachine_Turbine_01 EXIT

04   shallowManager[ShallowTurbineManager].Turbine_01__TO__History2_105720_61730 Effect

05   shallowManager[ShallowTurbineManager].StateMachine_Turbine_02 ENTRY

06   shallowManager[ShallowTurbineManager].StateMachine_Turbine_02 DO

07   shallowManager[ShallowTurbineManager].History2_105720__TO__Off_61731 Effect

08   shallowManager[ShallowTurbineManager].StateMachine_Turbine_02_Off ENTRY

09   shallowManager[ShallowTurbineManager].StateMachine_Turbine_02_Off DO

Note: Since *deepManager* has exactly the same trace as *shallowManager*, the trace for *deepManager* is filtered out from this sequence.

We can learn that:

- Exiting a Composite State commences with the innermost State in the active State configuration (see lines 01 - 03 in the trace sequence)
- The Default History Transition is only taken if execution leads to the History node (see line 04) and the State has never been active before (see line 07)


Then the active State configuration includes:

- Turbine_02
- Turbine_02.Off

This applies to both *deepManager* and *shallowManager.*

Trigger Sequence: [NEXT, MODE]
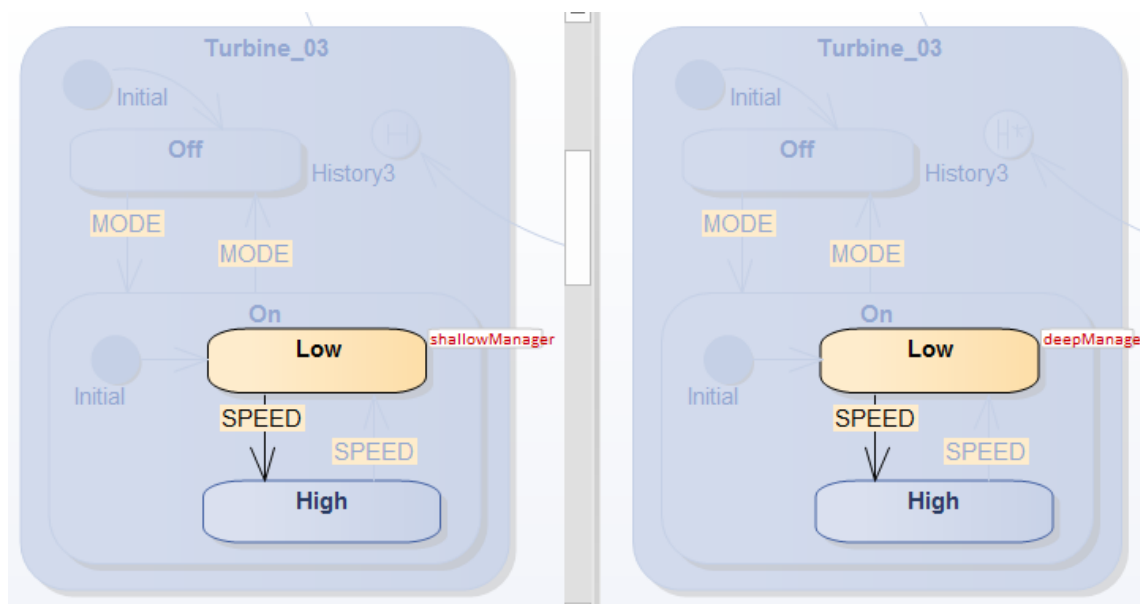
This trace sequence can be observed from the Simulation window:

    Trigger [NEXT]

01   shallowManager[ShallowTurbineManager].StateMachine_Turbine_02_Off EXIT

02   shallowManager[ShallowTurbineManager].StateMachine_Turbine_02 EXIT

03   shallowManager[ShallowTurbineManager].Turbine_02__TO__History3_105713_61725 Effect

04   shallowManager[ShallowTurbineManager].StateMachine_Turbine_03 ENTRY

05   shallowManager[ShallowTurbineManager].StateMachine_Turbine_03 DO

06   shallowManager[ShallowTurbineManager].Initial_105706__TO__Off_61718 Effect

07   shallowManager[ShallowTurbineManager].StateMachine_Turbine_03_Off ENTRY

08   shallowManager[ShallowTurbineManager].StateMachine_Turbine_03_Off DO

Trigger [MODE]

Message omitted...

Note: Since *deepManager* has exactly the same trace as *shallowManager,* the trace for *deepManager* is filtered out from this sequence.

We can learn that:

- Since there is no default History Transition defined for *History3*, the standard default entry of the State is performed; an Initial node is found in the Region contained by *Turbine_03,* so the Transition originating from *Initial* is enabled (see line 06)

Then the active state configuration includes:

- Turbine_03
- Turbine_03.On
- Turbine_03.On.Low

This applies to both *deepManager* and *shallowManager.*

History Entry of States

As a reference, we show the Deep History snapshot of each Turbine after its first activation:

Turbine_01
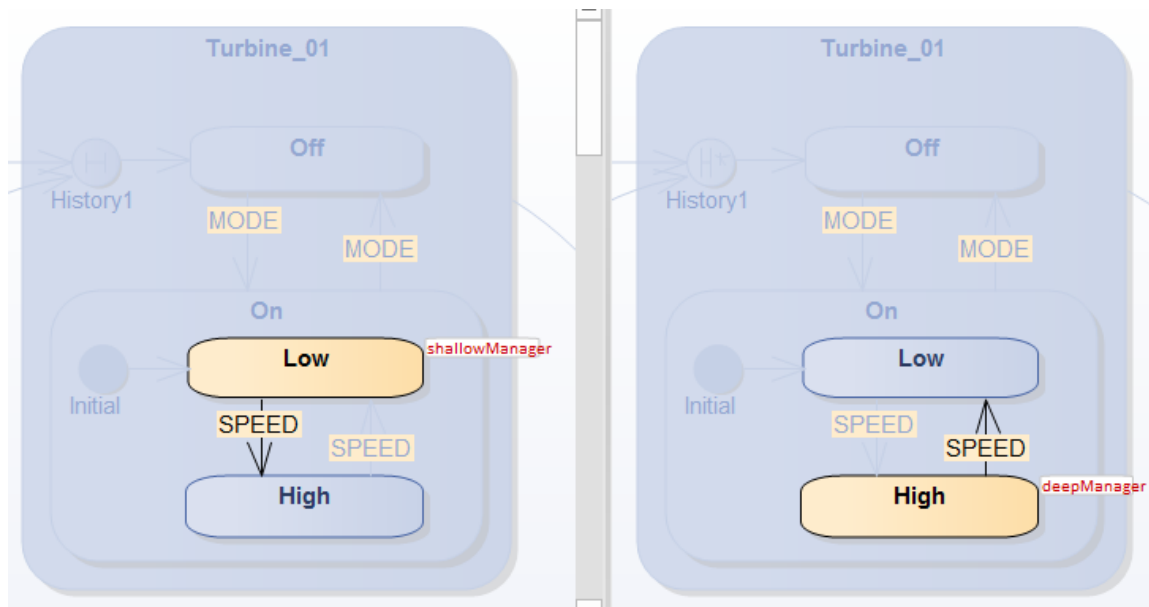
- Turbine_01.On
- Turbine_01.On.High

Turbine_02

- Turbine_02.Off

Turbine_03

- Turbine_03.On
- Turbine_03.On.Low

When we further Trigger NEXT, Turbine_01 will be activated again.

This trace sequence can be observed from the Simulation window:

For shallowManager:

    Trigger [NEXT]

01   shallowManager[ShallowTurbineManager].StateMachine_Turbine_03_On_Low EXIT

02   shallowManager[ShallowTurbineManager].StateMachine_Turbine_03_On EXIT

03   shallowManager[ShallowTurbineManager].StateMachine_Turbine_03 EXIT

04   shallowManager[ShallowTurbineManager].Turbine_03__TO__History1_105711_61732 Effect

05   shallowManager[ShallowTurbineManager].StateMachine_Turbine_01 ENTRY

06   shallowManager[ShallowTurbineManager].StateMachine_Turbine_01 DO

07   shallowManager[ShallowTurbineManager].StateMachine_Turbine_01_On ENTRY

08   shallowManager[ShallowTurbineManager].StateMachine_Turbine_01_On DO

09   shallowManager[ShallowTurbineManager].Initial_105721__TO__Low_61729 Effect

10   shallowManager[ShallowTurbineManager].StateMachine_Turbine_01_On_Low ENTRY

11   shallowManager[ShallowTurbineManager].StateMachine_Turbine_01_On_Low DO

We can learn that:

- The shallowHistory node restores Turbine_01 as far as Turbine_01.On

- Then the Region contained by Composite State Turbine_01.On will be activated by the *Initial* node, which activated at *Low*

For deepManager:

    Trigger [NEXT]

01   deepManager[DeepTurbineManager].StateMachine_Turbine_03_On_Low EXIT

02   deepManager[DeepTurbineManager].StateMachine_Turbine_03_On EXIT

03   deepManager[DeepTurbineManager].StateMachine_Turbine_03 EXIT

04   deepManager[DeepTurbineManager].Turbine_03__TO__History1_105679_61708 Effect

05   deepManager[DeepTurbineManager].StateMachine_Turbine_01 ENTRY

06   deepManager[DeepTurbineManager].StateMachine_Turbine_01 DO

07   deepManager[DeepTurbineManager].StateMachine_Turbine_01_On ENTRY

08   deepManager[DeepTurbineManager].StateMachine_Turbine_01_On_High ENTRY

We can learn that:

- The *deepHistory* node restores Turbine_01 as far as Turbine_01.On.High

Trigger [NEXT] to exit Turbine_01 and activate Turbine_02

Both *shallowManager* and *deepManager* activate Turbine_02.Off, which is the History snapshot when they exited.

Trigger [NEXT] to exit Turbine_02 and activate Turbine_03

Both *shallowManager* and *deepManager* activate Turbine_03.On.Low. However, the sequences of *shallowManager* and *deepManager* are different.

For *shallowManager*, the *shallowHistory* can only restore as far as Turbine_03.On. Since an *Initial* node is defined in Turbine_03.On, the Transition originating from *Initial* will be enabled and Turbine_03.On.Low is reached.

01   shallowManager[ShallowTurbineManager].StateMachine_Turbine_02_Off EXIT

02   shallowManager[ShallowTurbineManager].StateMachine_Turbine_02 EXIT

03   shallowManager[ShallowTurbineManager].Turbine_02__TO__History3_105713_61725 Effect

04   shallowManager[ShallowTurbineManager].StateMachine_Turbine_03 ENTRY

05   shallowManager[ShallowTurbineManager].StateMachine_Turbine_03 DO

06   shallowManager[ShallowTurbineManager].StateMachine_Turbine_03_On ENTRY

07   shallowManager[ShallowTurbineManager].StateMachine_Turbine_03_On DO

08   shallowManager[ShallowTurbineManager].Initial_105727__TO__Low_61728 Effect

09   shallowManager[ShallowTurbineManager].StateMachine_Turbine_03_On_Low ENTRY

10   shallowManager[ShallowTurbineManager].StateMachine_Turbine_03_On_Low DO

For *deepManager*, the *deephistory* can restore as far as Turbine_03.On.Low directly.

01   deepManager[DeepTurbineManager].StateMachine_Turbine_02_Off EXIT

02   deepManager[DeepTurbineManager].StateMachine_Turbine_02 EXIT

03   deepManager[DeepTurbineManager].Turbine_02__TO__History3_105680_61701 Effect

04   deepManager[DeepTurbineManager].StateMachine_Turbine_03 ENTRY

05   deepManager[DeepTurbineManager].StateMachine_Turbine_03 DO

06   deepManager[DeepTurbineManager].StateMachine_Turbine_03_On ENTRY

07   deepManager[DeepTurbineManager].StateMachine_Turbine_03_On_Low ENTRY

Event Macro: EVENT_PARAMETER

EVENT_PARAMETER is a function macro used to access attribute of a signal instance, in State's behavior, Transition's guard and effect. This macro will be expanded to executable code according to the simulation language.

Access Default Macro Definition

Ribbon | Develop | Source Code | Options | Edit Code Templates | Language | Stm Event Parameter

Usage Format

%EVENT_PARAMETER(SignalType, SignalAttributeName)%

For example: A signal 'MySignal' has two attributes, 'foo:int' and 'bar:int'; these use cases are valid:

*   Transition's effect: %EVENT_PARAMETER(MySignal, foo)%
*   State's behavior: %EVENT_PARAMETER(MySignal, bar)%
*   Transition's guard: %EVENT_PARAMETER(MySignal, bar)% > 10
*   State's behavior: %EVENT_PARAMETER(MySignal, bar)%++
*   Trace the value to Simulation Window: %TRACE(EVENT_PARAMETER(MySignal, foo))%

Macro Expansion Example

For Signal "MySignal" with attribute "value", Examples of macro expansion for Transition with signal trigger:%EVENT_PARAMETER(MySignal, value)%

C	((MySignal*)signal)->value
C++	static_cast<MySignal*>(signal)->value
C#	((MySignal)signal).value
Java	((EventProxy.MySignal)signal).value
JavaScript	signal.value

Example

This example demonstrated how to use EVENT_MACRO in Transition's effect, guard and State's behavior.
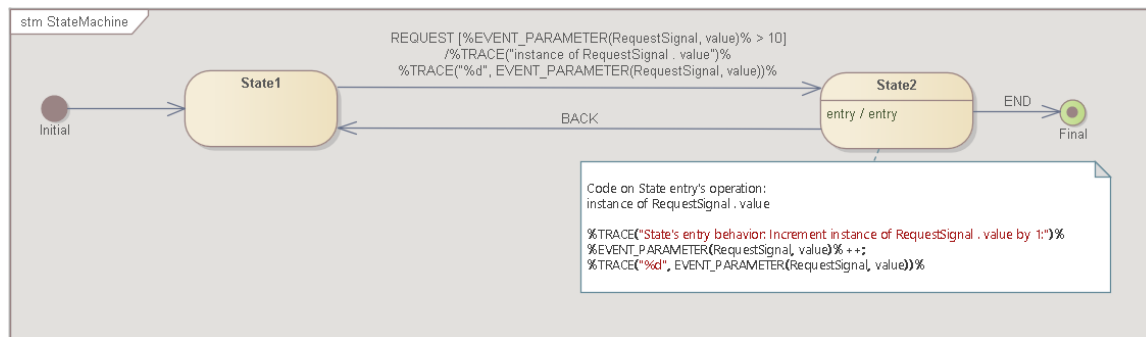
While running the simulation,

(1) trigger REQUEST and specify number 1 in for attribute value.

Since the condition for the guard is false, the active state will remain as State1.

(2) trigger REQUEST and specify number 11 for the attribute value.

Since the condition for the guard is true, the active state will change from State1 to State2;

The Transition's effect is executed. Here we traced the runtime value for the signal's attribute to the simulation window..

State2's behavior is executed. Here we incremented the runtime value for the signal's attribute and traced it to the simulation window.

[32608107]   [Part1:TransactionServer] Transition Effect: Initial_4019__TO__State1_4420

[32608118]   [Part1:TransactionServer] Entry Behavior: StateMachine_State1

[32608124]   [Part1:TransactionServer] Do Behavior: StateMachine_State1

[32608877]   [Part1:TransactionServer] Completion: TransactionServer_StateMachine_State1

[32608907]   Waiting for Trigger

[32613165]   Command: **broadcast REQUEST.RequestSignal(1)**

[32613214]   [Part1:TransactionServer] Event Queued: REQUEST.RequestSignal(value:1)

[32613242]   [Part1:TransactionServer] Event Dispatched: REQUEST.RequestSignal(value:1)

[32613279]   Waiting for Trigger

[32619541]   Command: **broadcast REQUEST.RequestSignal(11)**

[32619546]   [Part1:TransactionServer] Event Queued: REQUEST.RequestSignal(value:11)

[32619551]   [Part1:TransactionServer] Event Dispatched: REQUEST.RequestSignal(value:11)

[32619557]   [Part1:TransactionServer] Exit Behavior: StateMachine_State1

[32619562]   [Part1:TransactionServer] Transition Effect: State1__TO__State2_4421

[32619567]   **instance of RequestSignal . value**

[32619571]   **11**

[32619576]   [Part1:TransactionServer] Entry Behavior: StateMachine_State2

[32619584]   **State's entry behavior: Increment instance of RequestSignal . value by 1:**

[32619590]   **12**

[32619594]   [Part1:TransactionServer] Do Behavior: StateMachine_State2

[32620168]   [Part1:TransactionServer] Completion: TransactionServer_StateMachine_State2

[32620211]   Waiting for Trigger

[32622266]   Command: broadcast END

[32622272]   [Part1:TransactionServer] Event Queued: END

[32622310]   [Part1:TransactionServer] Event Dispatched: END

[32622349]   [Part1:TransactionServer] Exit Behavior: StateMachine_State2

[32622359]   [Part1:TransactionServer] Transition Effect: State2__TO__Final_4023_4423

[32622896]   [Part1:TransactionServer] Completion: TransactionServer_VIRTUAL_SUBMACHINESTATE

Limitation and Workaround

Since the macro expansion involve type casting, it is user's responsibility to ensure the type casting is valid.

And here are workarounds for some common cases that the type casting will encounter issues in the model.

- **One Transition has multiple triggers of different signal types**

For example, a Transition has triggerA(specified SignalA) and triggerB(specified SignalB); A macro %EVENT_PARAMETER(SignalA, attributeOfA)% will not work when this Transition is triggered by triggerB.

We suggest to create two Transitions, one with triggerA(specified SignalA), and the other with triggerB(SignalB).

- **One State is the target of multiple Transitions of different signal triggers**

For example, both TransitionA and TransitionB are targeting to MyState, TransitionA is triggered by SignalA, and TransitionB is triggered by SignalB.

If EVENT_PARAMETER is used in state's behavior code, one macro will not be able to work for both cases.

We suggest to move the logic dealing with signal's attribute from State's behavior to Transition's effect.

- **Customize template and generated code**

User can also change the default template to add Run Time Type Identification (RTTI) before type casting. User can also modify the generated code after generation, which can also satisfy the simulation purpose after compilation.