ENTERPRISE ARCHITECT

用户指南系列

可执行状态机

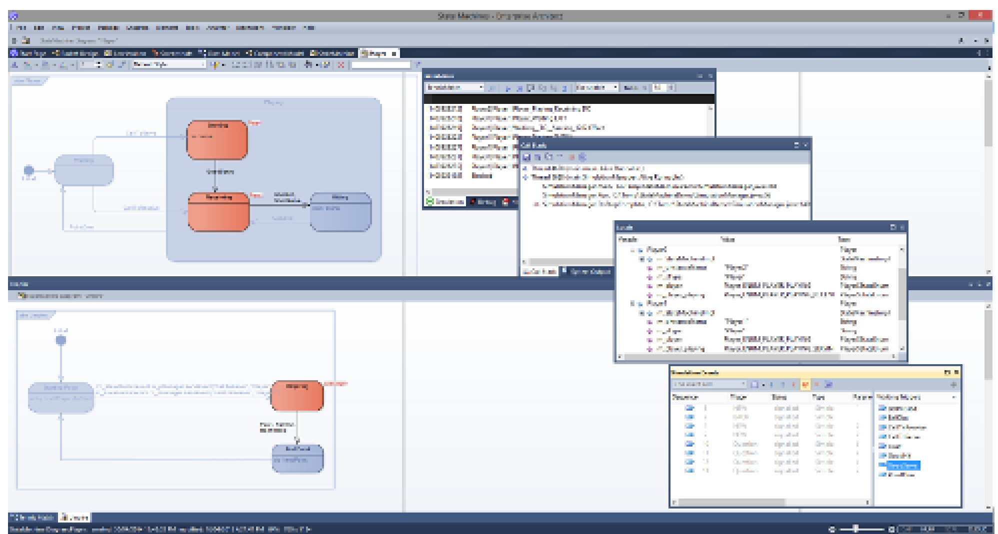
Author: Sparx Systems
Date: 13/11/2024
Version: 17.0

创建于

目录

可执行状态机 3
- 建模可执行状态机 5
- 可执行状态机工件 9
- 可执行状态机代码生成 11
- 可执行状态机的调试执行 16
- 可执行状态机的执行与仿真 18
- 示例：可执行状态机 19
- 示例：仿真命令 23
- 示例：在 HTML 中使用JavaScript进行仿真 31
  - 激光唱机 32
  - 正则表达式解析器 36
- 示例：进入状态 38
- 示例：分叉和汇合 46
- 示例：延迟事件模式 50
- 示例：入口和出口点（连接点参考） 56
- 示例：历史伪状态 61
- 事件宏：EVENT_PARAMETER 69

可执行状态机



可执行状态机提供了一种快速生成、执行和模拟复杂状态模型的方法。与使用Enterprise Architect的仿真引擎动态模拟状态图相比，多平台可执行状态机提供了完成的语言特定实现，可以形成多个软件的行为 引擎"。执行的可视化基于与仿真功能的无缝集成。模型的演变现在提出了更少的编码挑战。代码生成、编译和执行由Enterprise Architect负责。对于那些有特殊要求的人，每种语言都提供了一组代码模板。模板可以由您自定义，以您认为合适的任何方式定制生成的代码。

这些主题向您介绍建模可执行状态机的基础知识，并帮助您了解如何生成和模拟它们。

Enterprise Architect统一版和终极版支持可执行状态机的创建、使用和生成代码

建造和执行状态机概述

建造和使用可执行状态机非常简单，但确实需要一点计划和一些有关如何将不同组件链接起来以构建有效执行模型的知识。幸运的是，您不必花费数小时来获得正确的模型并修复编译错误，然后您就可以开始可视化您的设计。

在勾勒出模型的广泛机制后，您可以在几分钟内生成代码来驱动、编译、执行和可视化它。这些要点总结了开始执行和模拟状态机所需的内容。

功能	描述
编译类和状态模型	第一个任务是构建描述要构建的实体和行为的标准UML类和状态模型。对您的模型感兴趣的每一类都应该有自己的状态机，描述控制其整体行为的各种状态和转换。
创造一个可执行状态机的工件	一旦你已经建立了类模型和状态，就该设计可执行状态机的工件了。这将描述所涉及的类和对象，以及它们的初始属性和关系。正是链接多个对象的脚本决定了它们在运行时将如何通信。请注记，可以在一个可执行状态机工件或多个对象绑定在一起，并将其绑定为单个类的实例。这些将在运行时具有自己的状态和行为，并且可以在必要时进行交互。
生成代码并编译	无论您使用JavaScript 、C++、 Java还是 C#， Enterprise Architect的工程能力都为您提供了一个有效的工具，允许您随时重新生成可执行文件，并且不会丢失您可能制作的任何定制代码。这是项目生命周期中的主要优势。可能还值得注意的是，生成的整个代码库是独立且可移植的。代码绝不会与模拟引擎使用的任何基础设施相结合。
执行状态机	那么我们如何看待这些状态机的行为呢？一种方法是为每个平台构建代码

	库，将其集成到一个或多个系统中，在可能的几个部署场景中 就地"检查行为。或者我们可以使用Enterprise Architect执行它。无论是Java 、 JavaScript 、C、C++ 还是 C#， Enterprise Architect都会负责创建运行时、模型的托管、其行为的执行以及所有状态机的再现。
可视化状态机	可执行状态机可视化集成了Enterprise Architect的仿真工具。观察图表上发生的状态转换以及针对哪个object 。轻松识别共享相同状态的对象。重要的是，这些行为在多个平台上保持一致。您还可以控制机器的运行速度，以更好地了解事件的时间线。
调试状态机	当状态应该改变但不改变时，当转换不应该被启用但被启用时，当行为 - 简而言之 - 不受欢迎并且从模型中没有立即显现时，我们可以转向调试。Enterprise Architect的可视化执行分析器附带了 ExecutableStateMachine 代码生成支持的所有语言的调试器。调试提供了许多好处，其中之一可能是验证/证实附加到状态机中的行为的代码，以确保它被实际反映在执行过程中。

建模可执行状态机

可执行状态机模型和计算机模型所需的大部分工作是基于标准 UML 的类状态建模，尽管必须遵守一些约定以确保形成良好的代码库。唯一新奇的构造工件就是用元素来构成可执行状态机实例或场景的配置。工件：用于指定详细信息

- 代码语言（ JavaScript 、C#、 Java 、C++ 包括 C ）
- 场景中涉及的类和状态机
- 包括运行状态的实例规范；注记这可能包括同一状态机的多个实例，例如在网球比赛模拟中使用了两次 球员"类

可执行状态机基本建模工具和对象

这些是构建可执行状态机时使用的主要建模元素。

元素类型	描述
类和类图表	类定义了与被建模的状态机相关的object类型。例如，在一个简单的网球比赛场景中，您可以为球员、比赛、击球和裁判中的每一个定义一个类。每个都有自己的状态机，并且在运行时将由每个相关实体的object实例表示。有关类和类图的更多信息，请参阅*UML*建模指南。
状态机	对于您定义的将在场景中具有动态行为的每个类，您通常会定义一个或多个UML状态机。每个状态机将确定适用于拥有类的一个方面的合法的基于状态的行为。例如，可以有一个状态机代表玩家的情绪状态，一个跟踪他当前的健康和能量水平，一个代表他的输赢状态。所有这些状态机都会在状态机场景开始执行时被初始化并启动。
可执行状态机工件	这种刻板工件是用来指定可执行状态机的参与者、配置的核心元素和启动条件。从场景的角度来看，它用于确定涉及哪些（类）实例，它们可能会触发器发送和发送什么事件，以及它们在什么启动条件下运行。 从配置方面工件，你可以用来建立一个分析器脚本的链接，该分析器脚本将确定输出目录、代码语言、编译脚本等。右键单击工件设备将允许您生成、构建、编译和可视化您的机器的状态机执行。

状态机支持的构造

此表详细说明了支持的状态机造以及与每种类型相关的任何限制或一般约束。

构建	描述
状态机	• 简单状态机：状态机有一个区域 • 正交状态机：状态机包含多个区域 顶级区域（状态机所有）激活语义： 默认激活：状态机开始执行时。 入口入口：从入口过渡到包含区域中的顶点。 • 注记*1* ：每个拥有区域的状态机，入口只有一个转移从入口点到该区域内的区域 • 注记*2*：这个状态机可以被一个子机状态引用——连接点引用应该在子机

	中状态为转换的源/目标；连接点引用表示在状态机中定义并由子机状态引用的*Entry*/出口的用法 多状态机：浏览器窗口中的列出顺序决定执行顺序。 • 当涉及到一个子机状态时，类下可能有多个状态机 • 使用浏览器窗口工具栏中的上移或下移箭头调整状态机的顺序；置顶的将设置为主状态机 不支持 • 协议状态机 • 状态机重新定义
状态	• 简单状态：没有内部顶点或过渡 • 复合状态：只包含一个区域 • 正交状态：包含多个区域 • 子机状态：指整个状态机
复合状态Entry	• 默认条目 • 显式输入 • 浅历史条目 • 深度历史条目 • 入口
子状态	• 子状态和嵌套子状态 进入和退出语义，其中转换包括多个嵌套级别的状态，将服从嵌套行为（例如 OnEntry 和 OnExit）的正确执行。
过渡支持	• 外部转移 • 本地转移 • 内部转移 （画一个自我转移 并改变转移 对内部友好） • 完成转移 和完成事件 • 转移 警卫 • 复合过渡 • 触发优先级和选择算法 有关更多详细信息，请参阅*OMG UML*规范。
触发器的事件	可执行状态机仅支持 Signals 的事件处理。 要使用调用、计时或更改事件类型，您必须定义一个外部机制以基于这些事件生成信号。
信号	属性可以在 Signals 中定义；属性的值可以用作转移 中的事件参数转移 防护条件。 例如，这是 C++ 中转换效果中的代码集： 如果（信号->signalEnum == ENUM_SIGNAL2）

	{
	int xVal = ((Signal2*)signal)->myVal;
	}
	Signal2 生成如下代码：
	类 Signal2：公共信号{
	上市：
	信号2(){};
	Signal2(std::vector<String>& lstArguments);
	int myVal;
	};
	注记：更多细节可以通过生成一个可执行状态机并参考生成的 "EventProxy"文件找到。

最初的	Initial Pseudostate 表示区域的起点。最多是一个转移 的源转移 ;在一个区域中最多可以有一个 Initial Vertex。
区域	**默认激活和显式激活：** 转换终止于包含状态： • 如果在区域中定义了初始伪状态：**默认激活** • 如果没有定义初始 Pseudostate，则区域将保持非活动状态，并且包含状态被视为简单状态 • 如果转换终止于区域包含的顶点之一：**显式激活**，导致其所有正交区域的默认激活，除非这些区域也显式输入（多个正交区域可以通过源自的转换显式并行显式输入同分叉伪态） 例如，如果为正交状态定义了三个区域，并且*RegionA*和*RegionB*具有 Initial Pseudostate，则明确激活*RegionC* 。默认激活适用于*RegionA*和*RegionB* ；包含状态将具有三个活动区域。
选择	当复合转换遍历达到此伪状态时，动态评估所有传出转换的守卫条件约束。
连接点	静态条件分支：在执行任何复合转换之前评估保护约束。
分叉/汇合	非线程，每个活动区域交替移动一步，基于完成事件池机制。
入口点/出口点节点	非线程用于正交状态或正交状态机；每个活动区域交替移动一个步骤，基于完成事件池机制。
历史节点	• 状态：代表其所属状态的最近活动状态配置 • ShallowHistory：代表其包含状态的最近活跃的子态状态，而不是那个子状态的子态
延期事件	划一个自我转移 , 并改变转移 *kind* to类型();'在过渡的 影响"字段中。
连接点参考	A点参考表示子机状态引用的状态机中定义的条目/出口的使用（作为子机状态的一部分）。子机状态的连接点引用可以作为Transitions的源和目标。它们代表进入或退出子机状态所引用的状态机。

状态行为	状态 "entry"、 状态"和 "exit"行为被定义为对状态的操作。默认情况下，您将用于每个行为的代码键入到 行为操作的属性窗口的 代码"面板。请注记，您可以通过自定义生成模板将其更改为将代码键入 "行为"面板。 生成的 运行"行为将在继续之前运行完成。代码与其他入口行为不并发；'doActivity' 行为被实现为 'execute in order序列entry' 行为。

对其它/类中的行为的引用

如果子机状态引用了当前上下文或类之外的行为元素，则必须在当前上下文类中添加一个<<import>>连接器到容器上下文类。例如：

Class1中的子机状态S1指Class2中的状态机ST2

因此，我们将 <<import>> 连接器从 Class1 添加到 Class2，以便可执行状态机生成代码以正确生成子机状态S1的代码。（在类1上，单击快速链接箭头并拖动到类2，然后从连接器类型菜单中选择 导入"。）

再可执行状态机工件

您可以使用单个工件文件创建组件的多个模型或版本。工件一个电阻器，举例来说，可以重复使用箔电阻器和绕线电阻器这很可能是相似对象的情况，尽管它们由相同的分类器表示，但通常表现出不同的运行状态。从建模的角度来看， A名为 "resistorType"的属性取值 "wire"而不是 "foil"可能是所需的全部。然 可以重新使用相同的状态机来测试可能由于运行状态的变化而导致的行为变化。这是过程：

节	行动
创建或打开部件图	打开要处理的部件图。这可能是包含您的原始工件。
选择要复制的可执行状态机	现在在浏览器窗口中找到工件可执行状态机。
创建新部件	在按住 Ctrl 键的同时，将原始工件拖到您的图表上。系统将提示您两个问题。 第一个答案是**物件**，第二个答案是**全部**。重工件属性并将其与原始属性区分开来，然后继续更改其属性。

可执行状态机工件

一个可执行状态机工件产生状态机交互的关键它指定将参与模拟的对象、它们的状态以及它们如何连接。 A较大的状态状态机可执行状态机是在一个工件中使用多个部件的多个工件来表示一个实例，因此您可以设置模拟每个状态机的多个实例并观察它们如何交互。示例示例可执行状态机帮助中提供了一个示例。

创建一个可执行状态机的属性

每个可执行状态机场景都涉及一个或多个状态机。包含的状态机由UML属性元素指定；每个属性都会有一个UML分类器（类）来确定该类型包含的状态机。作为多个属性包含的多种类型最终可以包含许多状态机，这些状态机都是在代码中创建并在执行时初始化的。

行动	描述
从浏览器可执行状态机>>的窗口上工件一个类	最简单的方法是在可执行状态机可执行状态机从类浏览器窗口中定义属性。在显示的对话框中，选择创建属性的选项。您可以指定一个名称来描述该属性可执行状态机将如何引用此属性。 注记：根据您的选择，您可能需要按住 Ctrl 键来选择创建属性。可以使用 按住 Ctrl 以显示此对话框"复选框随时更改此行为。
使用和连接多个UML属性	一个可执行状态机描述了多个状态机的交互。这些可以是同一状态机的不同实例，同一实例的不同状态机，或者来自不同基本类型的完全不同的状态机。要创建多个属性，将使用相同的状态在相同的状态机上类工件。要使用不同的类型，请根据需要从浏览器窗口中删除不同的类。

定义属性的初始状态

运行的状态机，可执行状态机在上下文运行的情况下类。可执行状态机允许您通过将属性值分配给各种类属性来定义每个实例的初始状态。例如，如果这些属性与正在运行的场景相关，您可以指定玩家的年龄、身高、体重或类似属性。通过这样做，可以设置详细的初始条件，这些条件将影响场景的发展方式。

行动	描述
设置属性值对话框	可以通过右键单击属性并选择 特征|"来打开用 分配属性值的对话框。设置属性值'，或使用键盘快捷键 Ctrl+Shift+R。
赋值	设置属性值"对话框允许您为原始类中定义的任何属性定义值。为此，请选择变量，将运算符设置为 ="并输入所需的值。

定义属性之间的关系

除了要分配给每个属性所拥有的值之外，属性可执行状态机还允许您根据其他属性如何引用它们作为实例的类模型来定义每个属性的引用方式。

行动	描述
创建连接器	使用复合工具箱中的连接器关系连接多个属性。

	Connector 或者，使用快速链接器在两个属性之间创建关系并选择"连接器"作为关系类型。
映射到类模型	一旦两个属性之间存在连接器，您可以将其映射回它在类模型中表示的关联。为此，请选择连接器并使用键盘快捷键 Ctrl+L。将显示"选择一个关联"对话框，它允许生成的状态机在执行期间向填充关系中指定的角色的实例发送信号。

可执行状态机代码生成

为一种可执行状态机生成的代码是基于它的属性。这可能是Java 、C、C++、C# 或JavaScript 。无论是哪种语言， Enterprise Architect都会生成适当的代码，这些代码可以立即构建和运行。在运行它之前不需要手动干预。事实上，在初始生成后，任何一个可执行状态机都可以通过点击按钮生成、构建和执行。

支持的语言

可执行状态机支持以下平台语言的代码生成：

- 微软本机 C/C++
- 微软.NET (C#)
- 脚本(JavaScript)
- 甲骨文Java （ Java ）

从Enterprise Architect Release 14. 1开始，支持代码生成而不依赖于模拟环境（编译器）。例如，如果您没有安装 Visual Studio，您仍然可以从模型生成代码并在您自己的项目中使用它。如果您想在Enterprise Architect中模拟模型，仍然需要编译器。

仿真环境（编译器设置）

如果您想在Enterprise Architect中模拟可执行状态机模型，以下语言需要这些平台或编译器：

语言平台	框架路径示例
微软本机 (C/C++)	C:\Program 文件 (x86)\Microsoft Visual Studio 12.0 C:\Program Files (x86)\Microsoft Visual Studio\2017\专业(或其他版本)
微软.NET (C#)	C:\窗口\Microsoft.NET\Framework\v3.5（或更高版本）
脚本(JavaScript)	A N
甲骨文Java （ Java ）	C:\Program Files (x86)\ Java \jdk1.7.0_17（或更高版本）

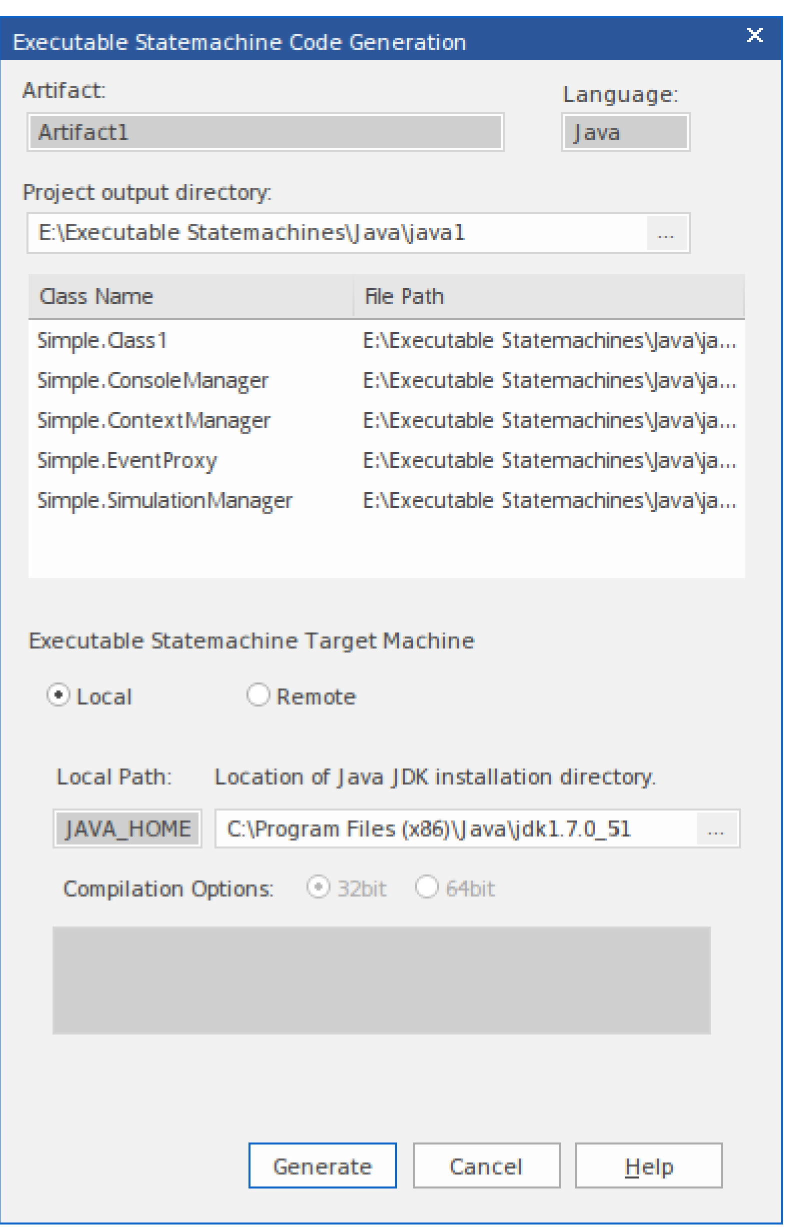
访问

功能区	仿真> 可执行状态> 状态机>生成、编译和运行或 仿真> 可执行状态> 状态机>生成

生成代码

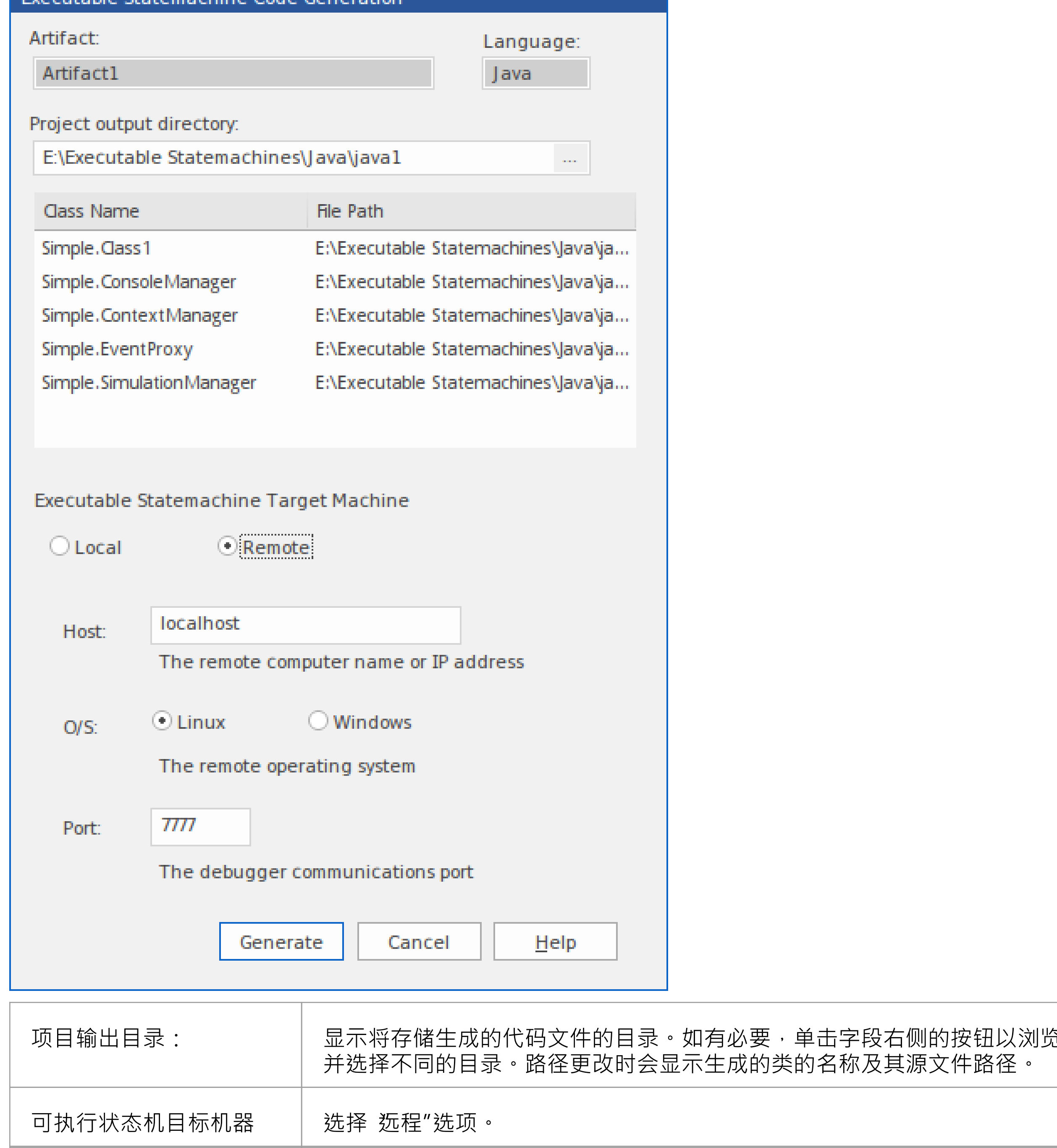
仿真> 可执行状态 > 状态机"功能区选项提供了用于为状态机生成代码的命令。工件选择可执行状态机，然后使用功能区选项生成显示的 可执行状态机代码生成"对话框取决于代码语言。

生成代码（ Java on窗口 ）



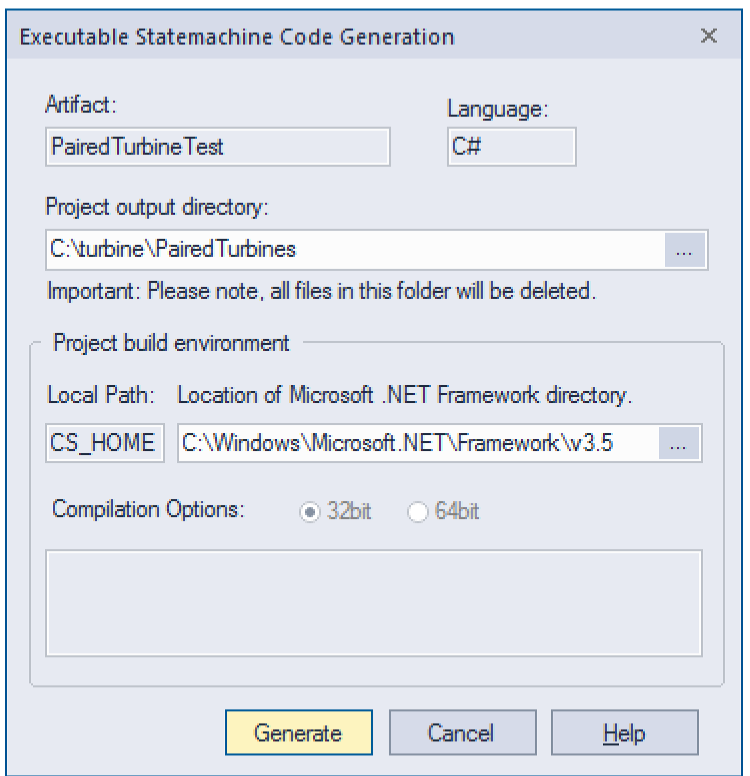
项目输出目录	显示将存储生成的代码文件的目录。如有必要，单击字段右侧的按钮以浏览并选择不同的目录。生成的类的名称和它们的源文件路径显示在这之后。
可执行状态机目标机器	选择 本地"选项。
Java JDK	输入要使用的Java JDK的安装目录。

生成代码（ Linux 上的Java ）



项目输出目录：	显示将存储生成的代码文件的目录。如有必要，单击字段右侧的按钮以浏览并选择不同的目录。路径更改时会显示生成的类的名称及其源文件路径。
可执行状态机目标机器	选择 远程"选项。
系统	选择 Linux。
端口	这是要使用的调试器端口。您将在分析器脚本生成的 调试"和 分析器"部分找到对这个端口号的引用。

生成代码（其它语言）



同时在 系统输出可执行状态机输出"页面打开窗口，在代码生成过程中显示哪些进度信息、警告或错误输出。

在 可执行状态机"字段和 属性名称"对话框中的 代码工件"字段显示元素的 属性"对话框中元素的 属性名称"和编码语言。

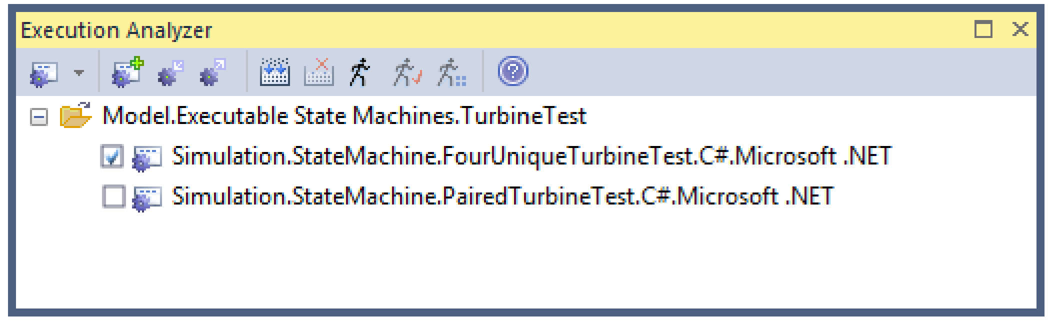
字段/选项	描述
项目输出目录	显示将存储生成的代码文件的目录。如有必要，单击字段右侧的按钮以浏览并选择不同的目录。
项目构建环境	此面板中的字段因信息元素和脚本中定义的工件而异。但是，每种支持的语言都提供了一个选项来定义构建和运行生成的代码所需的目标框架的路径；示例显示在本主题的支持的语言部分。 此路径的本地路径 ID 是在 本地路径"对话框中定义的，此处显示在 状态机代码生成"对话框和 可执行状态机代码生成"对话框中。

生成

点击此按钮生成状态机代码。代码生成将覆盖项目输出目录中的任何现有文件。文件集将包括所有必需的文件，包括状态机引用的每个类的文件。

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

每个可执行状态机还会生成一个执行分析器脚本，该脚本是用于构建、运行和调试运行可执行状态机的配置脚本。



建造代码

Enterprise Architect可执行状态机的代码可以通过以下三种方式之一进行构建。

方法	描述
功能区生成、编译和运行命令	再次对选中的生成可执行状态机，生成整个代码库。然后编译源代码并开始模拟。
功能区编译命令	编译已生成的代码。这可以在生成代码后直接使用，如果您对构建过程（分析器脚本进行了更改或以某种方式修改了生成的代码。
执行分析器脚本	生成的执行分析器脚本包含一个构建源代码的命令。这意味着当它处于活动状态时，您可以直接使用内置快捷键 Ctrl+Shift+F12 进行构建。
编译输出	构建时，所有输出都显示在系统输出窗口的 编译"页面上。您可以双击任何编译器错误以在相应行打开源编辑器。

利用现有代码

由Enterprise Architect生成并执行的可执行状态机可以利用没有类模型的现有代码。为此，您将创建一个抽象类元素，仅命名要在外部代码库中调用的操作。然后，您将在此接口和状态机类之间创建一个脚本，在分析器中手动添加所需的链接。对于Java ，您可以将 .jar 文件添加到类路径。对于本机代码，您可以将 .dll 添加到链接中。

可执行状态机的调试执行

可执行状态机的创建甚至在代码生成之后也提供了好处。使用执行分析器， Enterprise Architect能够连接到生成的代码。因此，您可以直观地调试和验证代码的正确行为；从您的状态机生成的完全相同的代码，由模拟演示并最终整合到现实世界的系统中。

调试状态机

能够调试可执行状态机会带来额外的好处，例如能够：

- 中断仿真的执行和所有正在执行的状态机
- 视图模拟中涉及的每个状态机实例的原始状态
- 在任意时间点视图源代码和调用堆栈
- 通过在源代码行上放置跟踪点来跟踪有关执行状态的其他信息
- 通过使用操作点和控件来执行控件（例如，错误时中断）
- 诊断由于代码或建模更改引起的行为变化

如果你已经生成、构建和运行成功了一个可执行状态机，你就可以调试它了！生成过程中创建的分析器脚本配置为提供调试功能。要开始调试，只需使用仿真控件开始运行可执行状态机控件。然而，根据被调试行为的性质，我们可能会先设置一些断点。

在状态转换中中断执行

像任何调试器一样，我们可以使用断点来检查代码中某个点的执行状态机。在图表或浏览器窗口中找到感兴趣的类，然后按 F12 查看源代码。从生成期间使用的命名约定很容易找到状态转换的代码。如果您想在特定转换处中断，请在编辑器中找到转换函数，然后通过单击函数内一行的左边距来放置断点标记。当你运行可执行状态机时，调试器会在这个状态机停止，你可以查看任何涉及到的状态机的原始状态。

有条件地中断执行

每个断点都可以采用条件和跟踪语句。当遇到断点并且条件评估为True时，执行将停止。否则执行将照常继续。您可以使用原始变量的名称构成条件，并使用标准相等操作数比较它们：< > = >= <=。例如：

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

要将条件添加到您设置的断点，请右键单击断点并选择 属性”。按住 Ctrl 键的同时单击断点，可以快速编辑属性。

追踪辅助信息

可以使用 TRACE 子句从状态机本身内部跟踪信息，例如效果。调试还提供称为跟踪点的跟踪特征。这些只是断点，而不是中断，在遇到它们时打印跟踪语句。输出显示在仿真控件窗口中。它们可以用作诊断辅助来显示和序列事件的顺序以及实例更改状态的顺序。

观看调用堆栈

每当遇到断点时，调用堆栈序列都可以使用 .使用分析器来确定执行的位置

可执行状态机的执行与仿真

Enterprise Architect的众多特征之一是其执行模拟的能力。 Enterprise Architect ，可以挂接到可执行状态机器仿真，直观地展示状态机工件功能实时执行。

开始仿真

仿真控件提供了一个搜索按钮，您可以使用该工具工件来运行可执行状态机。控件维护一个最近可执行状态机下拉列表供您选择。您也可以使用上下文工件的可执行状态机菜单来启动模拟。

控制速度

仿真控件提供了速度设置。您可以使用它来调整模拟执行的速率。速度表示为 0 到 100 之间的值（值越大越快）。 A值将导致模拟在每一步后停止；这需要使用工具栏控件手动逐步完成模拟。

活动状态的符号

随着可执行状态机的执行，显示相关状态机图。显示在每个步骤到完成周期结束时更新。您会注意到，仅突出显示完成步骤的实例的活动状态。其他州仍然黯淡无光。

很容易识别哪个实例处于哪个状态，因为状态标有当前处于该特定状态的任何实例的名称。如果相同类型的两个或多个工件状态将具有相同的属性状态，则每个属性名称都有一个单独的标签。

生成计时图表

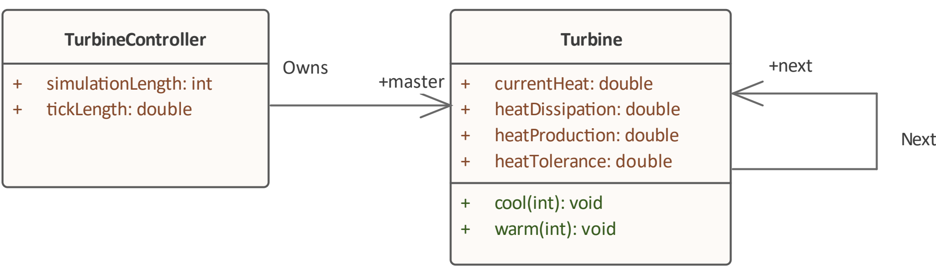
完成计算机的可执行状态机后，可以从输出中生成时序图。去做这个：

在仿真窗口工具栏中，单击 "工具|生成图表"。

示例：可执行状态机

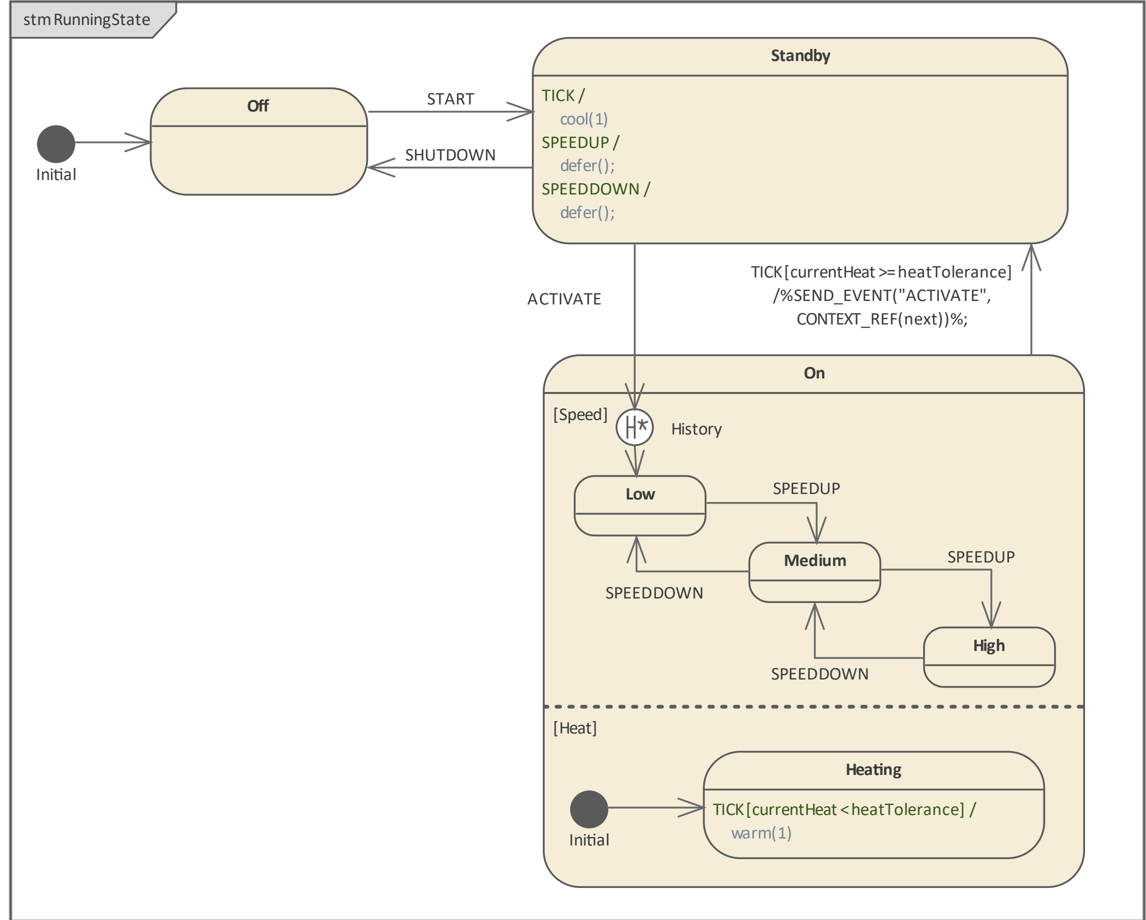
示例类模型

此图显示了本主题中描述的状态机使用的示例类模型。

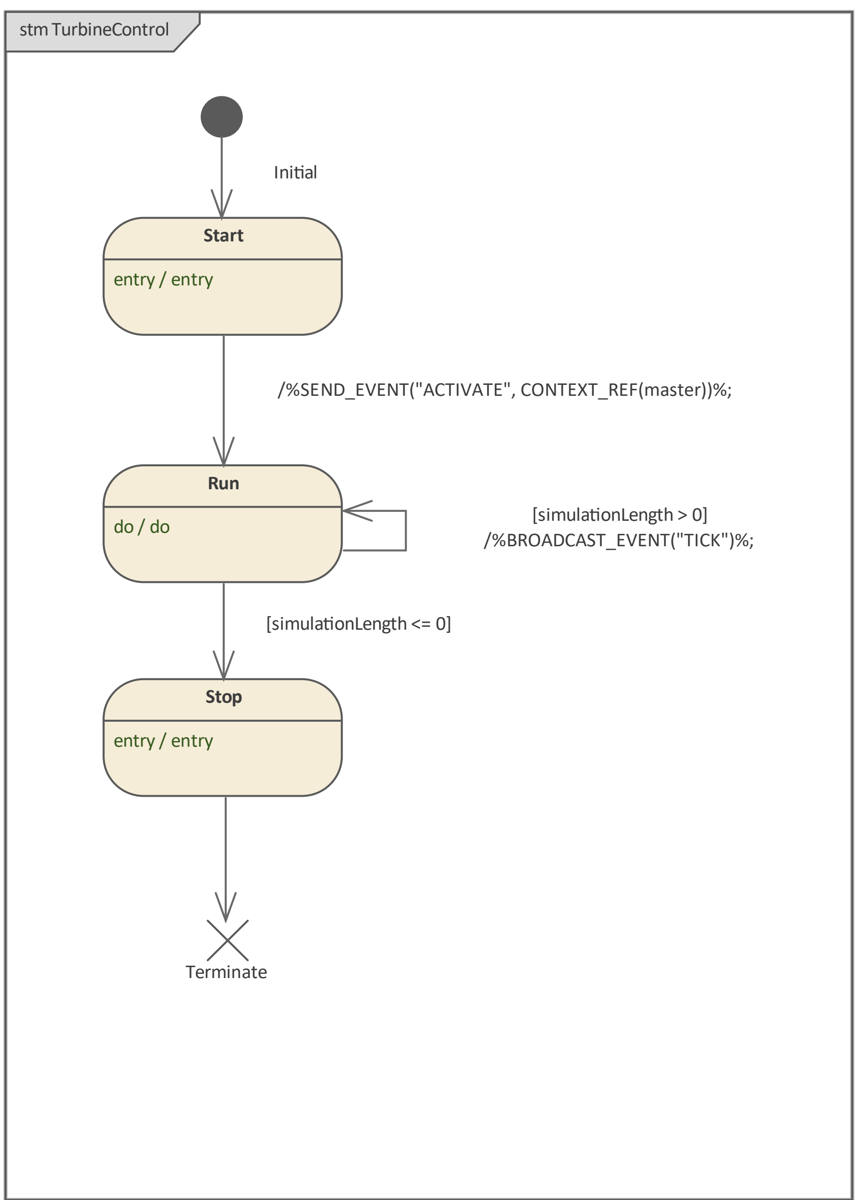


示例状态机

这两张图展示了两种状态机的定义。第一个引用另一个相同类型的状态机，而第二个驱动第一个存在的任何实例。

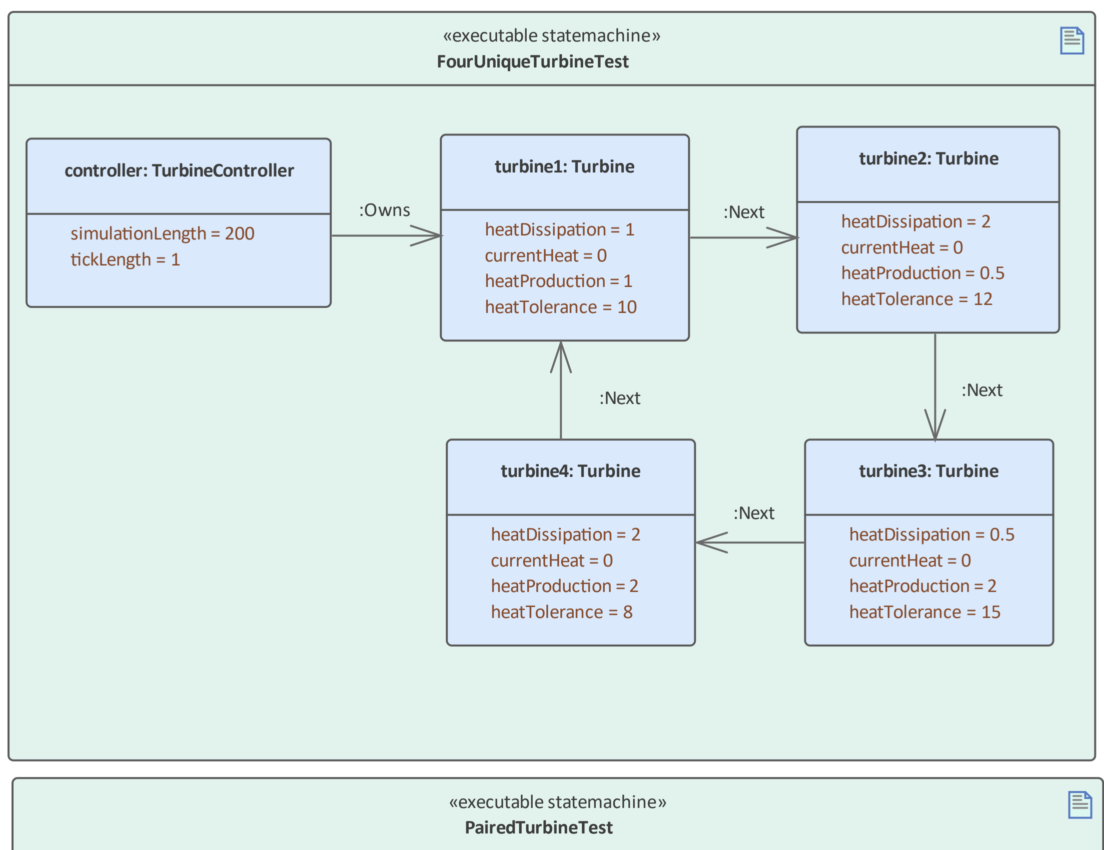


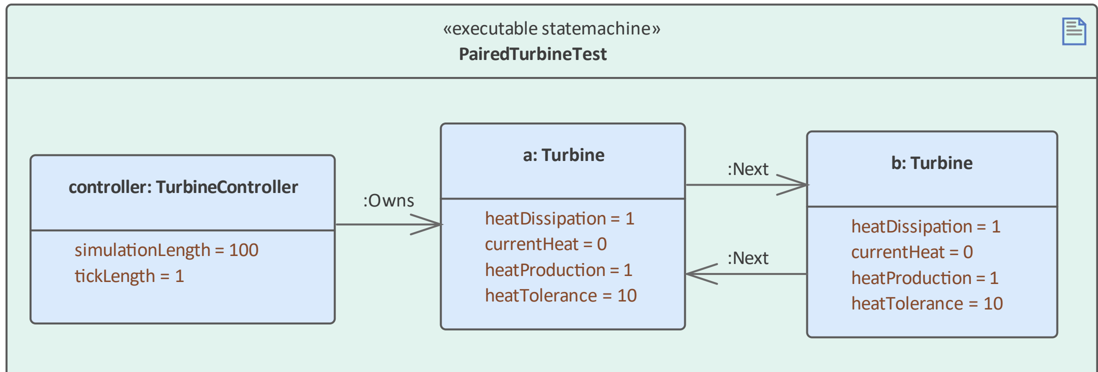
顶级控制器。



示例工件

从示例类和状态机图中，我们可以创建如下所示可执行状态机。

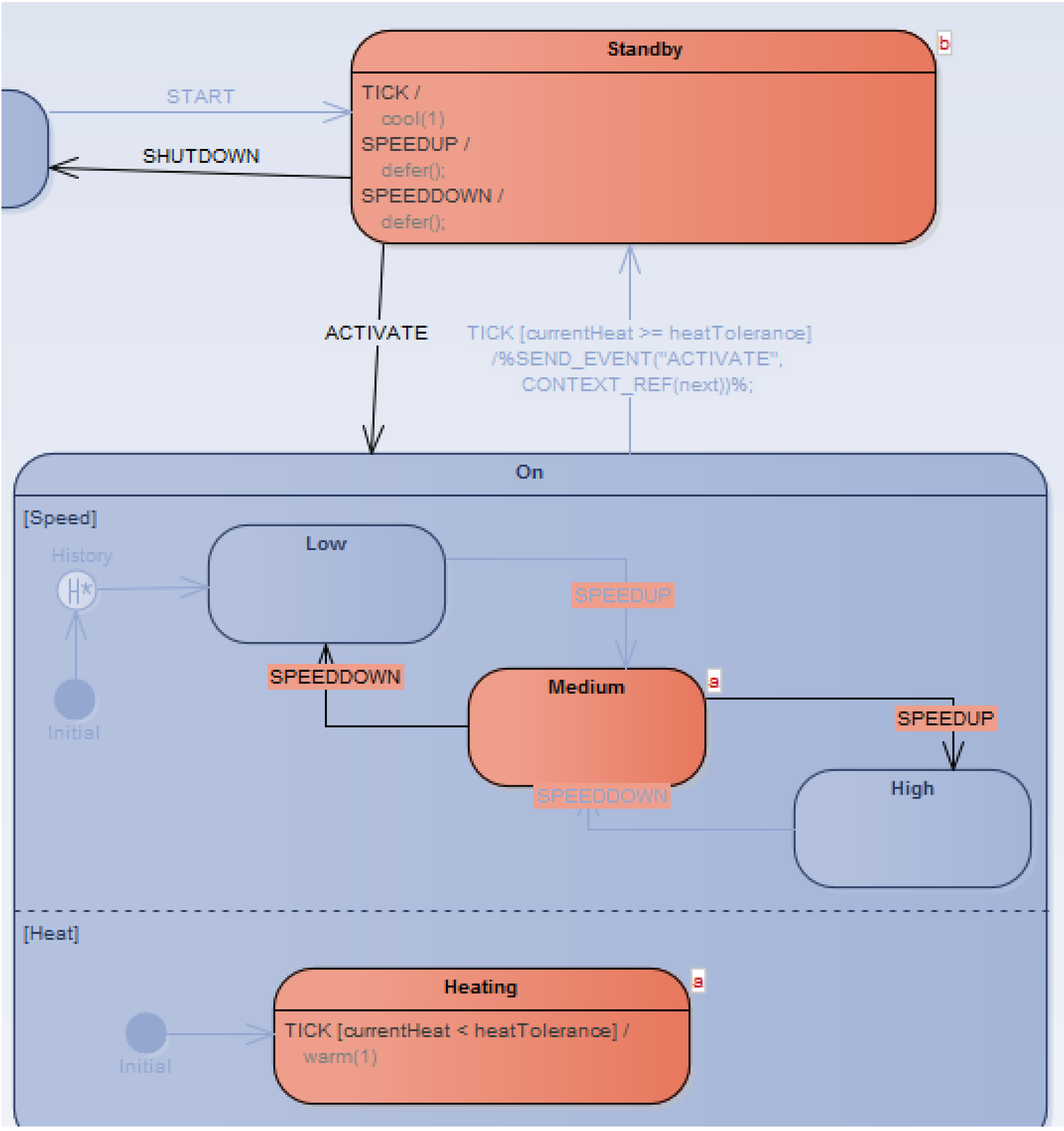




注记如何为每个属性设置属性值，元素之间的链接标识类模型中存在的关系。

仿真结果

运行模拟时， Enterprise Architect将突出显示任何状态机中的当前活动状态。在一个状态机的多个实例存在的情况下，它还将显示该状态下每个实例的状态。


Standby
TICK /
cool(1)
SPEEDUP /
defer();
SPEEDDOWN /
defer();
START
SHUTDOWN
ACTIVATE
TICK [currentHeat >= heatTolerance]
/%SEND_EVENT("ACTIVATE",
CONTEXT_REF(next))%;
On
[Speed]
History
Low
SPEEDUP
SPEEDDOWN
Medium
SPEEDUP
Initial
SPEEDDOWN
High
[Heat]
Heating
TICK [currentHeat < heatTolerance] /
warm(1)
Initial

示例：仿真命令

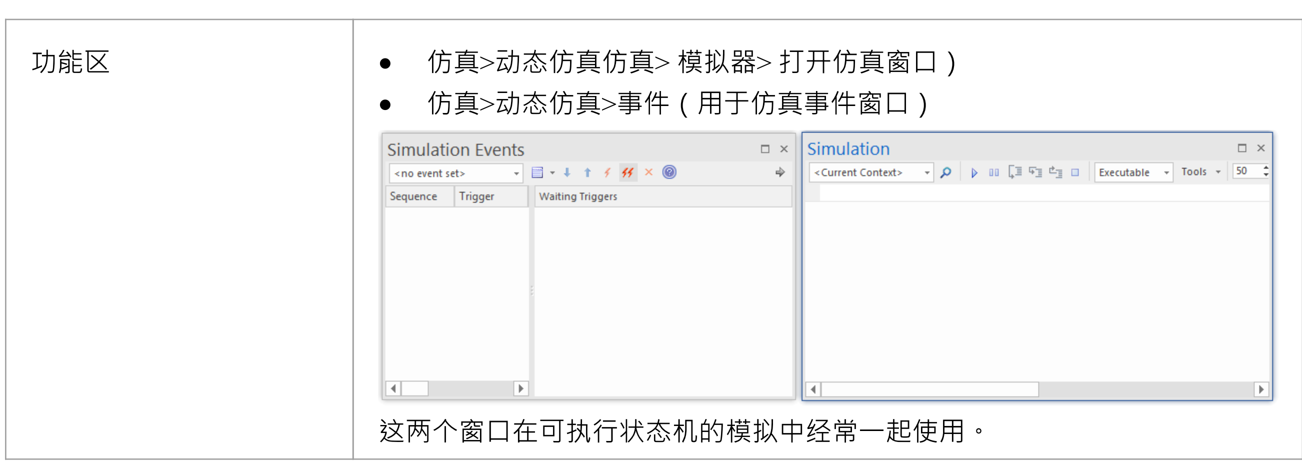
这个例子演示了我们如何使用仿真窗口来观察跟踪消息或发送命令来控制状态机。通过示例，您可以检查：

- 一个上下文的属性——类中定义的成员变量，即状态机的上下文；这些属性在上下文范围内携带值，用于所有状态行为和过渡效果，以访问和修改
- 一个信号的每个属性——信号中定义的成员变量，被一个事件引用，可以作为一个事件参数；每个信号事件出现可能有一个信号的不同实例
- 使用'Eval'命令查询上下文属性的运行时值
- 使用 转储"命令转储当前状态的活动计数；它还可以转储池中延迟的当前事件

此示例取自 EAExample模型：

示例模型.模型仿真.可执行状态机.仿真命令

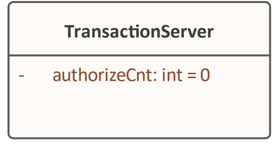
访问

功能区	• 仿真>动态仿真仿真> 模拟器> 打开仿真窗口） • 仿真>动态仿真>事件（用于仿真事件窗口）  这两个窗口在可执行状态机的模拟中经常一起使用。

创建上下文和状态机

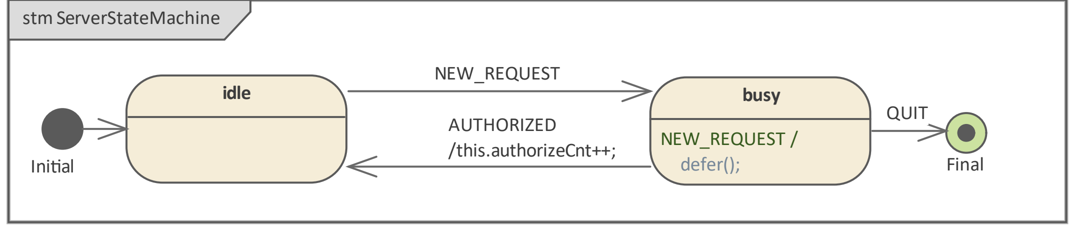
在本节中，我们将创建一个称为 TransactionServer 的类，它将状态机定义为其行为。然后我们创建一个可执行状态机工件模拟环境。

创建状态机的Context



1. 创建一个名为*TransactionServer*的类元素。
2. 在这个类中，创建一个名为*authorizeCnt*的属性，初始值为 0。
3. 在浏览器窗口中，右键单击*TransactionServer*并选择 添加|状态机"选项。

创建状态机



1. 创建一个名为*Initial*的 Initial 伪状态。
2. 转移
   到了一种状态空闲的状态。
3. 转移
   到一个称为忙的状态，触发状态。
4. 转移
   ：
   - 到一个称为终点的终点伪状态，使用触发器 QUIT
   - 回到空闲状态，触发 AUTHORIZED，影响'影响++;'

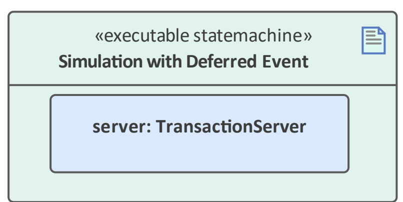
为忙状态创建一个延迟事件

1. 为忙碌划自我过渡。
2. 更改向 内部"过渡的 种类"。
3. 简单地触发器要延迟的事件。
4. 在 影响"字段中，输入 defer();"。

创建信号和属性

1. 创建一个称为*RequestSignal*的信号元素。
2. 创建一个名为*requestType*的属性，类型为 "int "。
3. 配置事件配置以引用*RequestSignal*。

创造可执行状态机工件



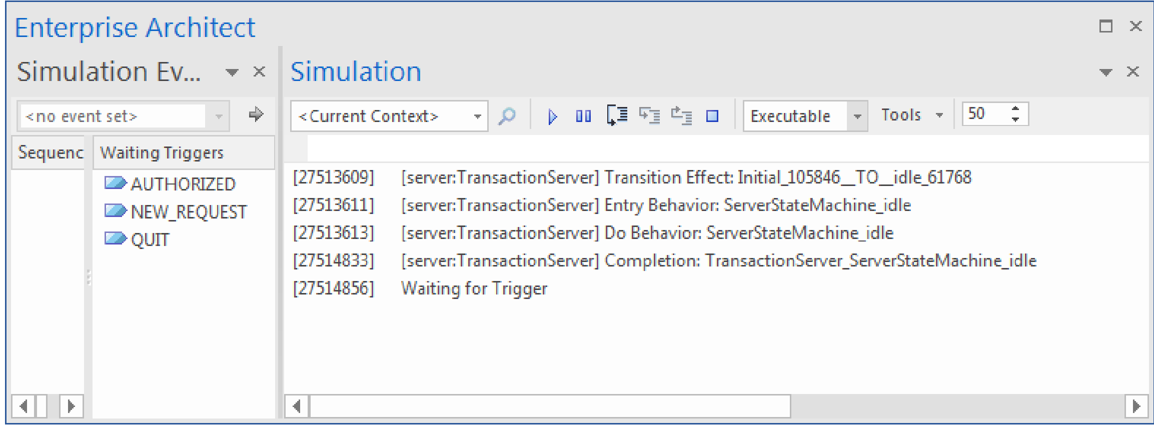
1. 在工具箱的页面中，将 无素仿真可执行状态机"图标图表图表上，并调用仿真事件。
2. Ctrl 从浏览器窗口拖动事务元素并将其工件到作为属性服务器的名称服务器上。
3. 将设备的JavaScript工件例如；在生产中，您还可以使用 C、C++、C# 或Java ，它们也支持可执行状态机）。
4. 点击工件仿真> 执行状态 > 执行状态 > 状态机生成,编译和运行功能区选项

仿真窗口和命令

模拟开始时， *idle*是当前状态。

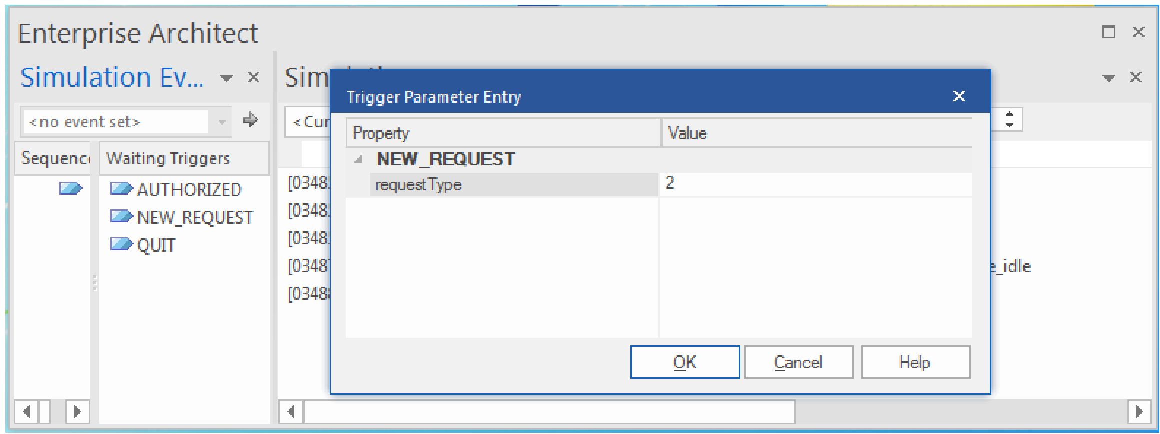
仿真窗口显示，转移

状态空闲的影响、进入和执行行为已完成，状态机正在等待触发。



通过信号属性值的事件数据

对于简单的信号事件触发器， 触发器参数Entry”对话框将显示用 提示在信号请求信号中定义的列出的属性的值，由信号引用。



类型值 2”并单击确定按钮。然 将信号属性值传递给调用的方法，例如状态的行为和转移的影响。

这些消息输出到仿真窗口：

[触发器] 等待简单

[03611358] 命令：广播 **NEW_REQUEST.RequestSignal(2)**

[03611362] [服务器：TransactionServer]事件排队：NEW_REQUEST.RequestSignal（requestType：2）

[03611367] [服务器：TransactionServer]事件调度：NEW_REQUEST.RequestSignal（requestType：2）

[03611371] [server:TransactionServer] 退出行为：ServerStateMachine_idle

[03611381] [服务器：TransactionServer]转移

影响：影响

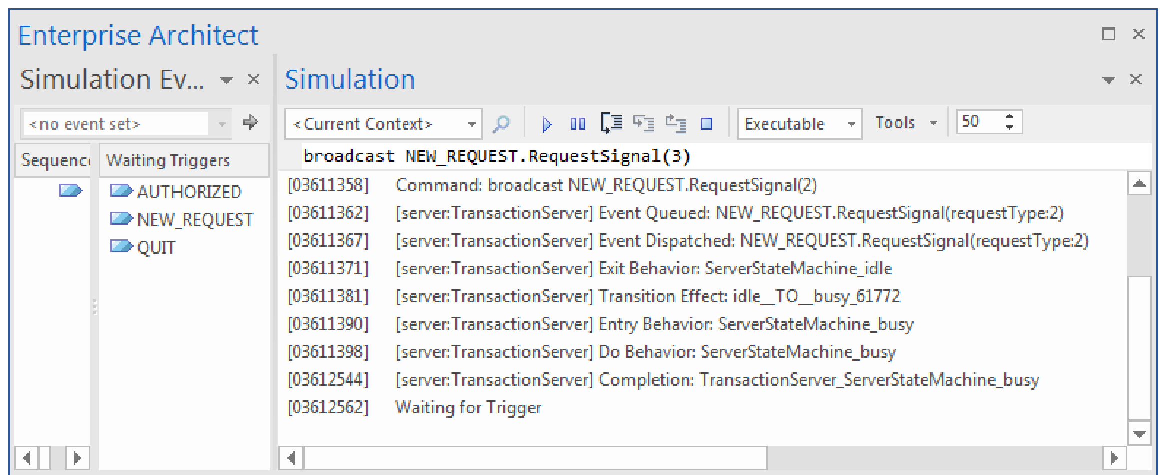
[03611390] [server:TransactionServer] 入口行为：ServerStateMachine_busy

[03611398] [server:TransactionServer]行为：ServerStateMachine_busy

[03612544] [服务器：TransactionServer] 完成：TransactionServer_ServerStateMachine_busy

[触发器] 等待简单

我们可以通过双击仿真事件窗口中列出的项目来播放事件。或者，我们可以在仿真窗口（工具栏下方）的文本字段中键入命令string 。



[触发器] 等待简单

[04460226] 命令：**广播 NEW_REQUEST.RequestSignal(3)**

[04460233] [服务器：TransactionServer]事件排队：NEW_REQUEST.RequestSignal（requestType：3）

[触发器] 等待简单

仿真事件消息表明事件发生被推迟（排队，但未调度）。我们可以使用文本字段运行更多命令：

[触发器] 等待简单

[04664057] 命令：**广播 NEW_REQUEST.RequestSignal(6)**

[04664066] [服务器：TransactionServer]事件排队：NEW_REQUEST.RequestSignal（requestType：6）

[触发器] 等待简单

[04669659] 命令：**广播 NEW_REQUEST.RequestSignal(5)**

[04669667] [服务器：TransactionServer]事件排队：NEW_REQUEST.RequestSignal（requestType：5）

[触发器] 等待简单

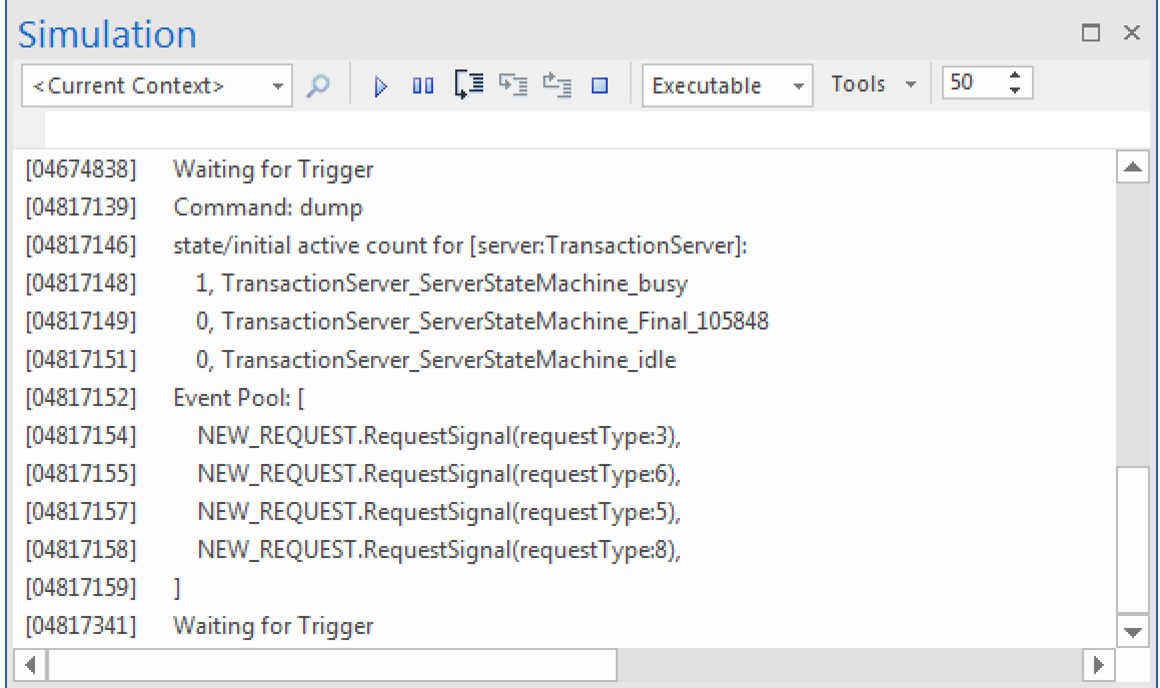
[04674196] 命令：**广播 NEW_REQUEST.RequestSignal(8)**

[04674204] [服务器：TransactionServer]事件排队：NEW_REQUEST.RequestSignal（requestType：8）

[触发器] 等待简单

dump:查询状态和事件的 活动泳池”

文本字段中的类型转储；这些结果显示：



从 活动计数"部分，我们可以看到忙碌是活动状态（活动计数为1 ）。

提示：对于复合状态，活动计数为1 （对自身）加上活动区域的数量。

从 事件泳池"部分，我们可以看到事件队列中有四个事件发生。信号的每个实例都携带不同的数据。池中事件的顺序是它们被广播的顺序。

eval：查询运行时间Context的值

触发器授权，

[触发器] 等待简单

[05494672] 命令：广播已授权

[05494678] [服务器：TransactionServer]事件排队：已授权

[05494680] [服务器：TransactionServer]事件调度：授权

[05494686] [server:TransactionServer] 退出行为：ServerStateMachine_busy

[05494686] [服务器：TransactionServer]转移

影响：忙影响

[05494687] [server:TransactionServer] 入口行为：ServerStateMachine_idle

[05494688] [server:TransactionServer]行为：ServerStateMachine_idle

[05495835] [服务器：TransactionServer] 完成：TransactionServer_ServerStateMachine_idle

[05495842] [服务器：TransactionServer]事件调度：**NEW_REQUEST.RequestSignal（requestType：3）**

[05495844] [server:TransactionServer] 退出行为：ServerStateMachine_idle

[05495846] [服务器：TransactionServer]转移

影响：影响

[05495847] [server:TransactionServer] 入口行为：ServerStateMachine_busy

[05495850] [server:TransactionServer]行为：ServerStateMachine_busy

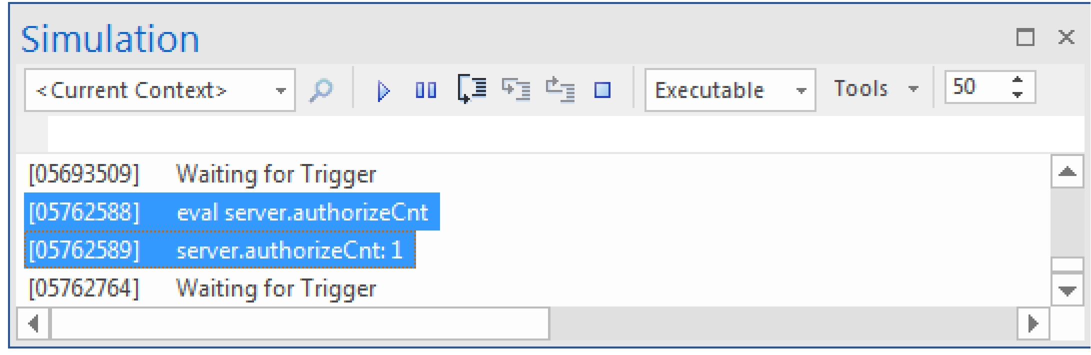
[05496349] [服务器：TransactionServer] 完成：TransactionServer_ServerStateMachine_busy

[触发器] 等待简单

- 从忙到空闲的转换已经完成，所以我们期望执行效果
- 空闲完成时从池中回调一个事件并调度，使忙碌状态变为活动状态
- 类型转储，注意池中还剩下三个事件；第一个被召回和发送

[泳池]事件泳池：[
[05693349] NEW_REQUEST.RequestSignal(requestType:6),
[05693351] NEW_REQUEST.RequestSignal(requestType:5),
[05693352] NEW_REQUEST.RequestSignal(requestType:8),
[05693354]]

文本字段中的类型*eval server.authorizeCnt* 。该图表示 '运行' 的运行时间值为1 。



触发器再次授权。当状态机稳定在*busy*时，池中会剩下两个事件。再次运行*eval server.suthorizeCnt* ；该值为 2。

从状态行为和转移

访问上下文的成员变量转移

影响

Enterprise Architect的可执行状态机，支持模拟C、C++、C# Java和JavaScript 。

对于 C 和 C++，访问上下文的成员变量的语法与 C#、 Java和JavaScript不同。 C 和 C++ 使用指针 '->' 而其他的只是使用 '.'；但是，您始终可以使用*this.variableName*来访问变量。 Enterprise Architect将把它翻译成 this->variableName 用于 C 和 C++。

因此，对于所有语言，只需使用以下格式进行模拟：

this.variableName

例子：

在过渡效果中：

this.authorizeCnt++;

在某些状态的进入、做或退出行为：

this.foo += this.bar;

注记：默认情况下，对于 C 和 C++， Enterprise Architect仅将 'this->' 替换为 'this'；例如：

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

将被翻译成：

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

支持的命令A完全列表

由于多个可执行状态机工件一起模拟一些上下文，因此可以指定一个实例名称。

运行状态机：

由于每个上下文可以有多个状态机， 运行"命令可以指定一个状态机的机器。

- 运行instance.statemachine
- 运行
- 运行实例
- 运行
- 运行

例如：

运行

运行

运行服务器

运行

广播和发送事件：

- 广播事件字符串
- 将 EventString 发送到实例
- 发送EventString（相当于广播EventString）

例如：

广播事件1

将 Event1 发送给客户端

转储命令：

- 倾倒
- 转储实例

例如：

倾倒

转储服务器

转储客户端

评估命令：

- eval instance.variableName

例如：

评估 client.requestCnt

评估 server.responseCnt

退出命令：

- 出口

EventString 的格式：

- EventName.SignalName（参数列表）

注记：参数列表应与信号中按顺序定义的属性相匹配。

例如，如果信号定义了两个属性：

- 富
- 酒吧

那么这些 EventStrings 是有效的：

- Event1.Signal1(10, 5) --------- foo = 10;酒吧 = 5
- Event1.Signal1(10,) --------- foo = 10;酒吧未定义
- Event1.Signal1(,5) --------- bar = 10; foo未定义
- Event1.Signal1(,) --------- foo 和 bar 都没有定义

如果信号不包含任何属性，我们可以将 EventString 简化为：

- 事件名称

示例：在 HTML 中使用JavaScript进行仿真

我们已经知道用户可以模型模拟可执行状态机并在Enterprise Architect中使用生成的代码进行模拟。使用*CD* 播放器和正则表达式解析器这两个示例，我们现在将现在如何将生成的代码与实际项目集成。

Enterprise Architect为客户端代码使用状态机提供了两种不同的机制：

- 状态- 客户端可以查询当前活动状态，然后根据查询结果 切换"逻辑
- Runtime Variable Based - 客户端不作用于当前活动状态，但作用于包含状态机的类中定义的变量的运行时值

在*CD*播放器的例子中，GUI上的状态很少，按钮很多，所以基于Active状态的例子很容易实现；我们还将查询当前轨道的运行时值。

Load Random CD

Number Of Tracks	Current Track	Track Length	Time Elapsed
13	2	0:20	0:10

在正则表达式解析器示例中，状态机处理所有事情，并且成员变量*bMatch*在状态更改时更改其运行时值。客户端不会注册有多少状态或当前处于活动状态的状态。

Regular Expression: **(a|b)*abb**

Input string to see if it match: aaababb

在这些主题中，我们将逐步演示如何为指定的正则表达式模型、模拟和集成 CD 播放器和解析器：

- CD Player
- Regular Expression Parser

激光唱机

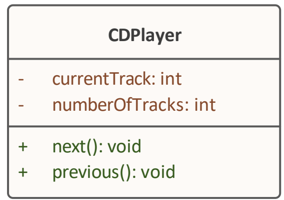
CD 播放器应用程序的行为可能看起来很直观；但是，有许多规则与按钮何时启用和禁用、窗口的文本字段中显示的内容以及向应用程序提供事件时发生的情况有关。

假设我们的示例 CD 播放器具有以下特征：

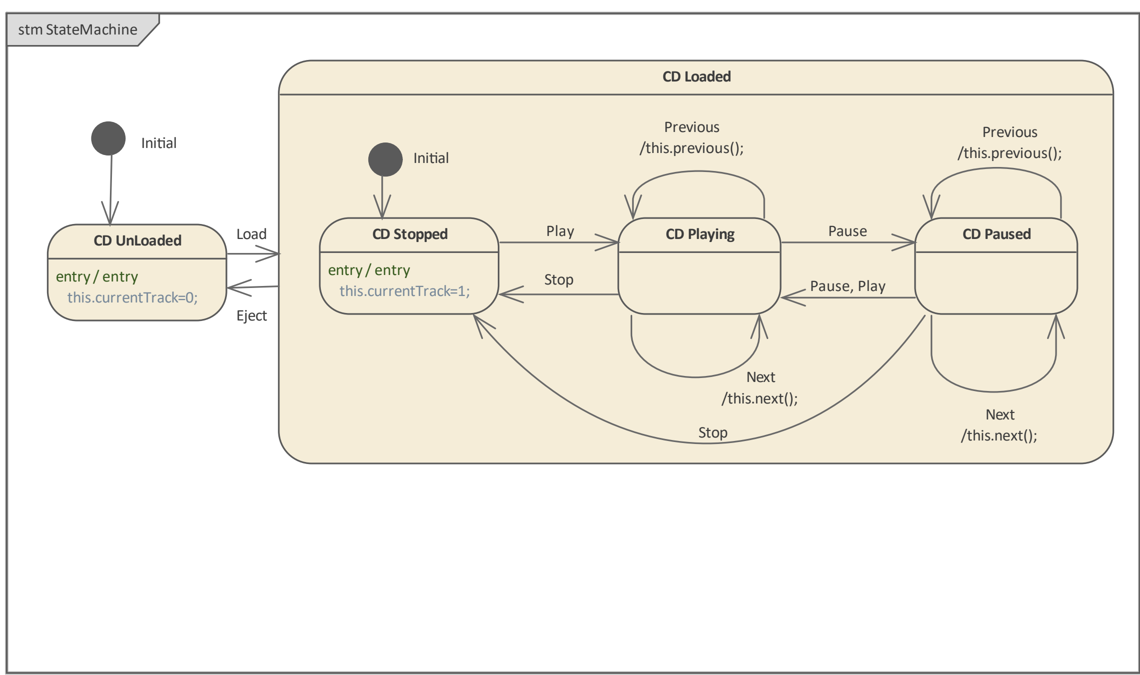
- 按钮 - 加载随机 CD、播放、暂停、停止、上一曲目、下一曲目和弹出
- 显示 - 曲目数、当前曲目、曲目长度和已播放时间

CD播放器状态机

A类*CDPlayer*定义了两个属性： *currentTrack*和*numberOfTracks* 。

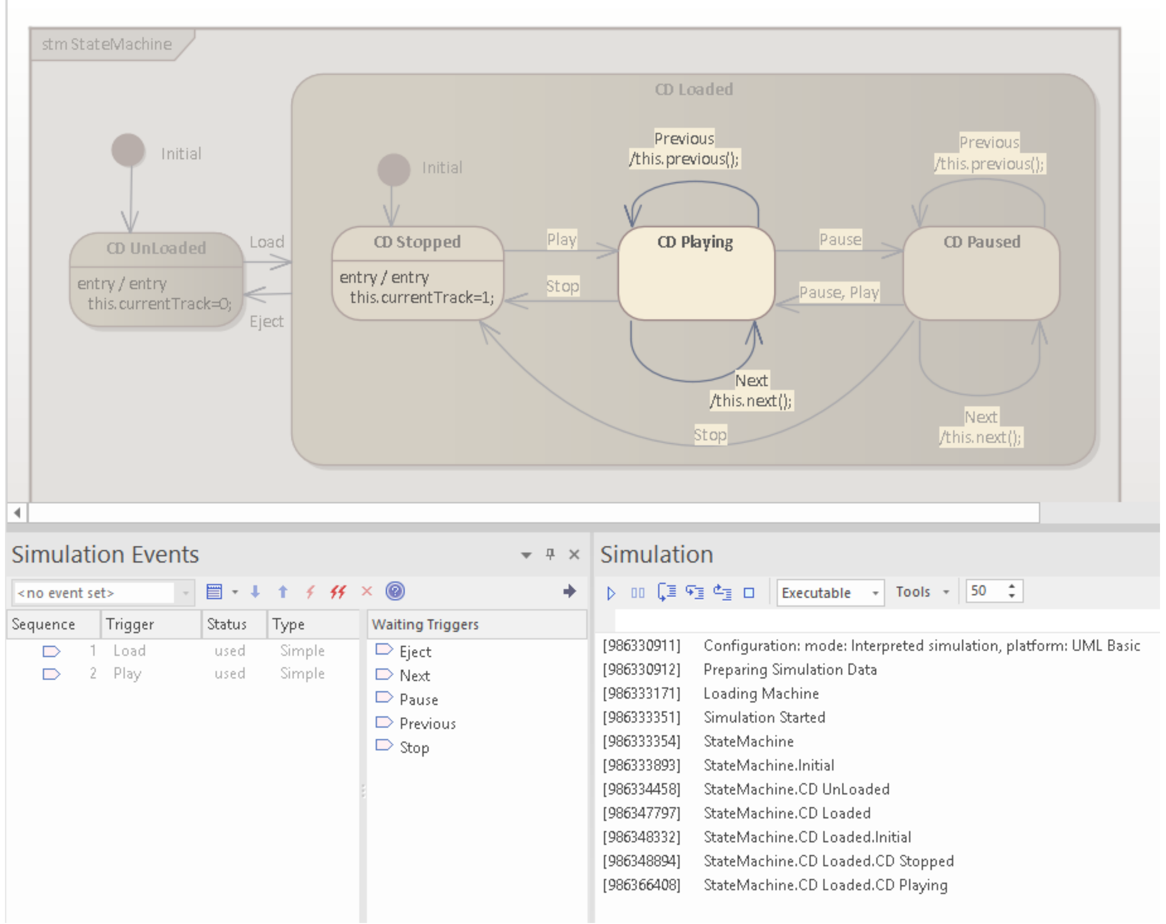


状态机用于描述 CD 播放器A状态：



- 在更高的层次上，状态机有两种状态： *CD UnLoaded*和*CD Loaded*
- *CD Loaded*可以由三个简单状态组成： *CD Stopped* 、 *CD Playing* 、 *CD Paused*
- 转换定义为事件 Load、Eject、Play、Pause、Stop、Previous 和 Next 的触发器
- 定义状态behaviors和效果状态定义的属性值；例如，'Previous' 事件将触发自转换（如果当前状态是*CD Playing*或*CD Paused* ）并执行影响，这将减少*currentTrack*的值或换行到最后一个轨道

我们可以创建一个状态状态机的属性，然后在Enterprise Architect可执行状态机中模拟该工件的类型，以确保该模型是正确的。



检查生成的代码

Enterprise Architect将在您指定的文件夹中生成这些文件：

- 后端代码：CDPlayer.js、ContextManager.js、EventProxy.js
- 客户端代码：ManagerWorker
- 前端代码：statemachineGUI.js、index.html
- 其它代码：SimulationManager.js

文件	描述
/CDPlayer.js	该文件定义了类CDPlayer 及其属性和操作。它还用状态行为和过渡效果定义了类的状态机。
/ContextManager.js	该文件是上下文的抽象管理器。该文件定义了与实际上下文无关的内容，这些内容是在*ContextManager*的泛化中定义的，例如*SimulationManager*和*ManagerWorker* 。 模拟（可执行状态机）可以涉及多个工件；例如，在网球比赛模拟中，将有一个裁判输入类*Umpire*，两个玩家——*playerA*和*playerB*——输入类。类和*ClassPlayer*类定义自己的状态机。
/EventProxy.js	该文件定义了仿真中使用的事件和信号。 如果我们用参数引发一个事件，我们模型事件为一个信号事件，它指定一个信号类；然后我们为信号类定义属性。每个事件发生都有一个信号实例，带

	有为属性指定的运行信号。
/SimulationManager.js	该文件用于Enterprise Architect中的模拟。
/html/ManagerWorker.js	该文件充当前端和后端之间的中间层。 • 前端发消息向ManagerWorker请求信息 • 由于ManagerWorker是从ContextManager泛化而来的，所以它可以完全访问所有上下文，例如查询当前活动状态和查询变量的运行时值 • ManagerWorker 将向前端发送一条消息，其中包含从后端检索到的数据
/html/statemachineGUI.js	该文件通过定义*stateMachineWorker*来建立前端和 ManagerWorker 之间的通信。它： • 定义函数*startStateMachineWebWorker*和*stopStateMachineWebWorker* • 使用占位符代码定义函数*onActiveStateResponse*和*onRuntimeValueResponse*： //to do: 写用户的逻辑 您可以简单地用您的逻辑替换此评论，本主题稍后将演示
/html/index.html	这定义了 HTML用户接口，例如用于引发事件或显示信息的按钮和输入。您可以在此文件中定义 CSS 和JavaScript 。

自定义 index.html 和 statemachineGUI.js

对生成的文件进行以下更改：

- 创建按钮和显示
- 创建 CSS 样式以格式化显示并启用/禁用按钮图像
- 创建一个 ElapseTimeWorker.js 以每秒刷新一次显示
- 创建一个TimeElapsed函数，当经过的时间达到轨道长度时设置为Next Track
- 创建JavaScript作为按钮 'onclick' 事件处理程序
- 广播事件后，请求状态的活动状态和运行时值
- 初始化时，请求活动状态

在 statemachineGUI.js 中找到函数*onActiveStateResponse_cdPlayer*

- 在 CDPlayer_StateMachine_CDUnLoaded 中，禁用所有按钮并启用 btnLoad
- 在 CDPlayer_StateMachine_CDLoaded_CDStopped 中，禁用所有按钮并启用 btnEject 和 btnPlay
- 在 CDPlayer_StateMachine_CDLoaded_CDPlaying 中，启用所有按钮并禁用 btnLoad 和 btnPlay
- 在 CDPlayer_StateMachine_CDLoaded_CDPaused 中，启用所有按钮并禁用 btnLoad

在 statemachineGUI.js 中找到函数*onRuntimeValueResponse*

- 在*cdPlayer.currentTrack*中，我们更新当前曲目和曲目长度的显示

完全的示例

通过单击此链接，可以从Sparx Systems网站的 资源"页面访问该示例：

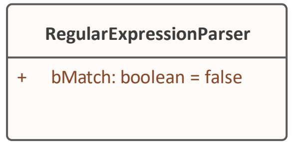
CD播放器仿真

单击加载随机 CD 按钮，然后单击开始仿真按钮。

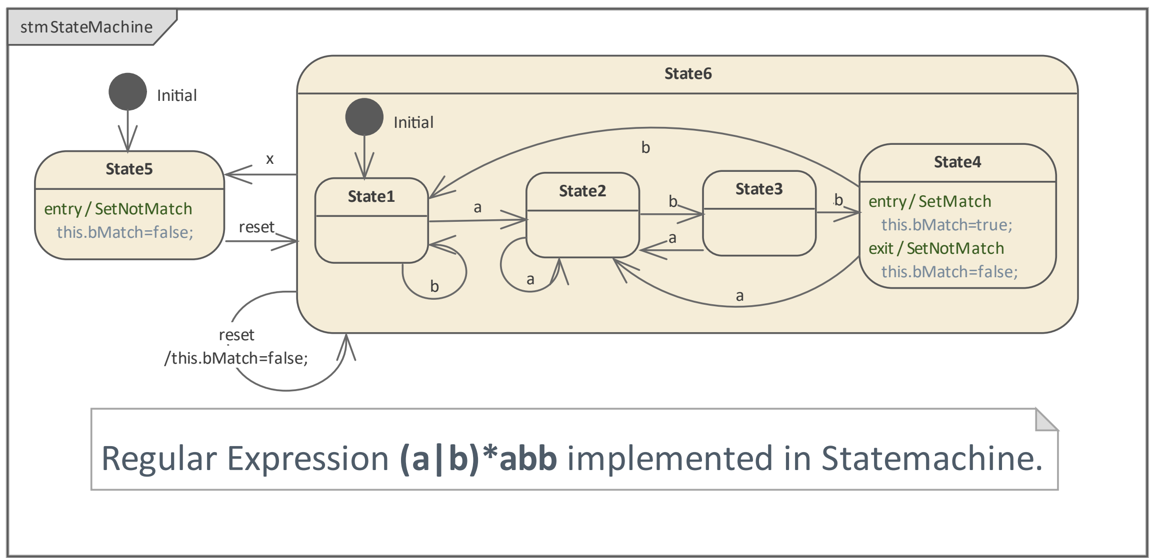
正则表达式解析器

正则表达式状态机解析器

类*RegularExpressionParser*定义了一个属性：bMatch。



状态机用于描述正则表达式 (A |b)*abb



- 转换触发器被指定为事件*a* 、 *b* 、 *x*和*reset*
- 在进入 State4 时，bMatch 设置为True ；从 State4 退出时，bMatch 设置为False
- 在进入 State5 时，bMatch 设置为False
- 在 State6 的自我转换中，bMatch 设置为False

自定义 **index.html** 和 **statemachineGUI.js**

对生成的文件进行以下更改：

- 创建一个 HTML 输入字段和一个图像来指示结果
- 创建JavaScript作为字段的*oninput*事件处理程序
- 创建函数 "SetResult "以切换通过/失败图像
- 创建函数 "getEventStr"，它将在 "a" 上返回 "*a* "，在 "b" 上返回 "*b* "，但在任何其他字符上返回 "x"
- 初始化时，广播 "重置"
- 在广播事件上，请求运行时变量 "regxParser.bMatch"

在*statemachineGUI.js*中，找到函数'onRuntimeValueResponse'。

- 在 "regxParser.bMatch" 中，我们将收到 "True "或 "False "并将其传递给 "SetResult" 以更新图像

完全的示例

通过单击此链接，可以从Sparx Systems网站的 资源"页面访问该示例：

正则表达式解析器仿真

示例：进入状态

进入状态的状态取决于状态的状态和进入的方式。

在所有情况下，状态的进入行为在状态时执行（如果已定义），但仅在与传入转移

相关联的任何效果行为之后转移

完成了。此外，如果为状态定义了行为，则该行为在执行该条目行为之后立即开始执行。

对于定义了一个或多个区域的复合状态，每个区域存在许多替代方案：

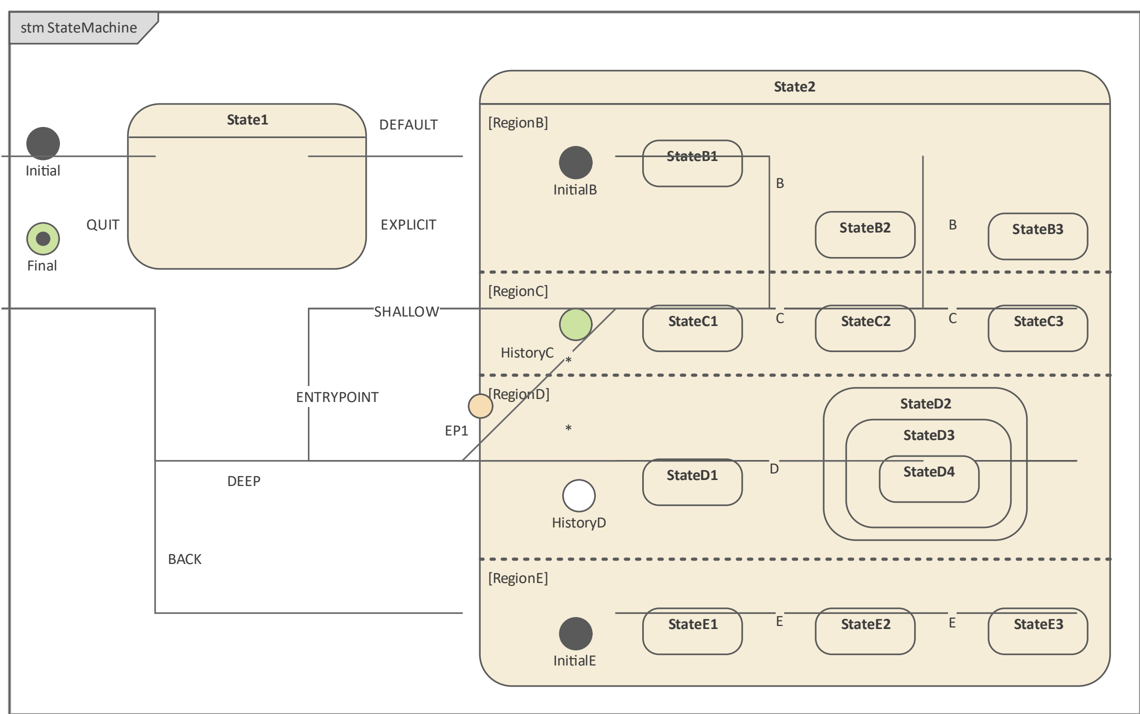
- 默认条目：当拥有复合状态是转移
  的直接目标时会发生这种情况转移
  ;在执行 entry行为并分叉一个可能的行为执行之后，状态entry 通过它的输出转移
  从一个初始的 Pseudostate 继续转移
  （称为默认转移
  的状态），如果它在区域中定义
  如果没有定义初始 Pseudostate，则此区域将不会处于活动状态
- 显式入口：如果传入转移
  或者它的延续终止于拥有复合状态的直接包含的子态状态，然后该子态状态变为活动状态，并且在包含复合状态的行为执行之后执行其条目行为
  如果转移
  ，则此规则递归适用转移
  以间接（深度嵌套）子态终止
- 浅历史入口：如果传入转移
  终止于该区域的一个区域Pseudostate，激活的子态状态成为在该进入之前最近处于活动状态的子态状态（除了 FinalState），除非这是进入该状态的第一个条目；如果它是第一个进入这个状态的条目，之前的条目已经到达了一个终点或一个默认的浅历史转移
  如果定义了就取，否则应用默认状态入口
- 深度历史条目：这种情况的规则与浅层历史的规则相同，只是目标伪状态的类型为 deepHistory 并且该规则递归地应用于此之下的活动状态配置中的所有级别
- 入口点入口：如果一个转移
  通过一个状态Pseudostate进入拥有复合状态，然后向外转移
  取自入口点，渗透到该区域内的状态；如果从入口点有更多的传出转换，则每个转移
  必须定位到不同的区域并且所有区域同时激活

对于具有多个区域的正交状态，如果转移

显式输入一个或多个区域（在分叉或入口点的情况下），这些区域显式输入，其他区域默认输入。

在这个例子中，我们演示了一个具有正交状态的所有这些入口行为的模型。

建模状态机



状态机的上下文

1. 创建一个名为*MyClass*的类元素，作为状态机的上下文。
2. 在浏览器窗口中右键单击*MyClass*并选择 添加|状态机"选项。

状态机

1. 在图中添加一个*Initial*节点、一个名为状态的状态、一个名为 *State2*的状态和一个名为 final 的终点元素。
2. 放大图上的*State2* ，右键单击它并选择'高级|定义 Concurrent Substates 的选项，定义*RegionB*、*RegionC*、*RegionD*和*RegionE*。
3. 右键单击*State2*并选择 'New子元素|入口"选项以创建入口*EP1* 。
4. 在*RegionB*中，创建元素*InitialB* ，过渡到*StateB1* ，过渡到*StateB2* ，过渡到*StateB3* ；由事件触发的所有转换 *B*
5. 在*RegionC*中，创建元素 shallow *HistoryC* （右键单击 History 节点 |高级| Deep History | 取消选中），过渡到 *StateC1* ，过渡到*StateC2* ，过渡到*StateC3* ；事件*C*触发的所有转换。
6. 在*RegionD*中，创建元素 deep *HistoryD* （右键单击 History 节点 |高级| Deep History | 勾选），过渡到*StateD1* ，创建*StateD2*作为 StateD3 的父级，它是*StateD4*的父级；从*StateD1*到*StateD4*的转换；由事件*D*触发。
7. 在*RegionE*中，创建元素*InitialE* ，过渡到*StateE1* ，过渡到*StateE2* ，过渡到*StateE3* ；由事件触发的所有转换 *E*
8. 从入口*EP1*到*StateC1*和划的划线过渡。

不同条目类型的划转换：

1. 默认条目： *State1*到*State2* ；由事件DEFAULT 触发。
2. 显式条目： *State1*到*StateB2* ；由事件EXPLICIT 触发。
3. 浅历史条目：从*State1*到*HistoryC* ；由事件SHALLOW 触发。
4. 深度历史条目： *State1*到*HistoryD* ；由事件DEEP 触发。
5. 入口条目： *State1*到*EP1* ；由事件ENTRYPOINT 触发。

其它转换：

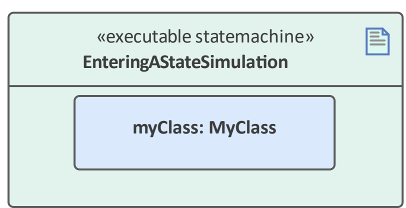
1. 复合状态Exit：从*State2*到*State1* ；由事件BACK 触发。
2. *State1*到终点，由事件QUIT 触发。

仿真

工件

Enterprise Architect支持 C、C++、C#、 Java和JavaScript 。我们在此示例中使用JavaScript ，因为我们不需要安装编译器。（对于其他语言，需要 Visual Studio 或 JDK。）

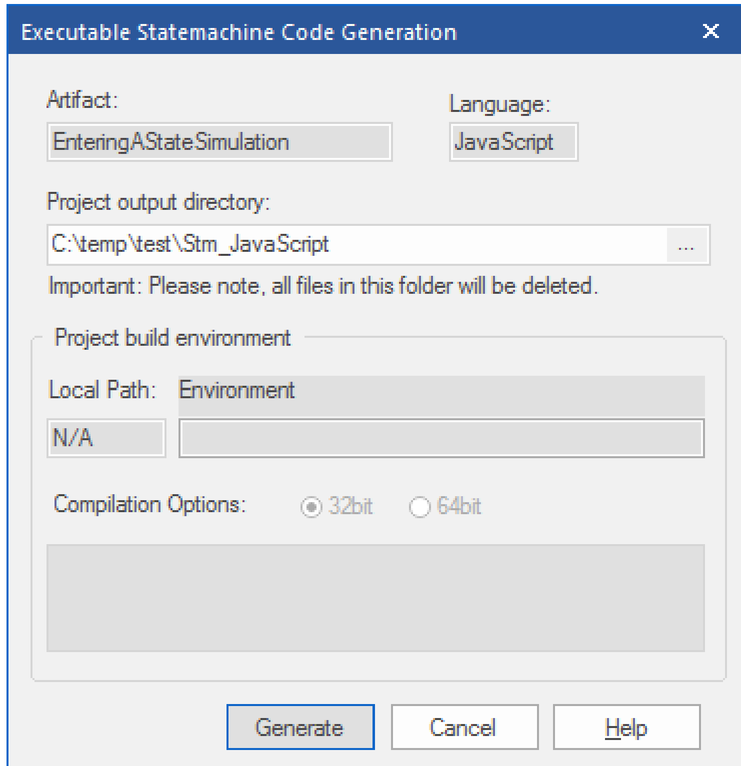
1. 在工具图表的 仿真工具"页面上，将 输入和可执行状态机"的图表工具箱一个工件 状态模拟"的仿真工具上。将语言设置为JavaScript 。
2. Ctrl+ 将*MyClass*从元素浏览器窗口输入到工件属性浏览器中，选择 粘贴作为属性"选项并将名称命名为属性。



代码生成

1. 单击*EnteringAStateSimulation*并选择 仿真> 可执行状态 > 状态机 >生成、编译和运行"功能区选项。
2. 为生成的源代码指定一个目录。

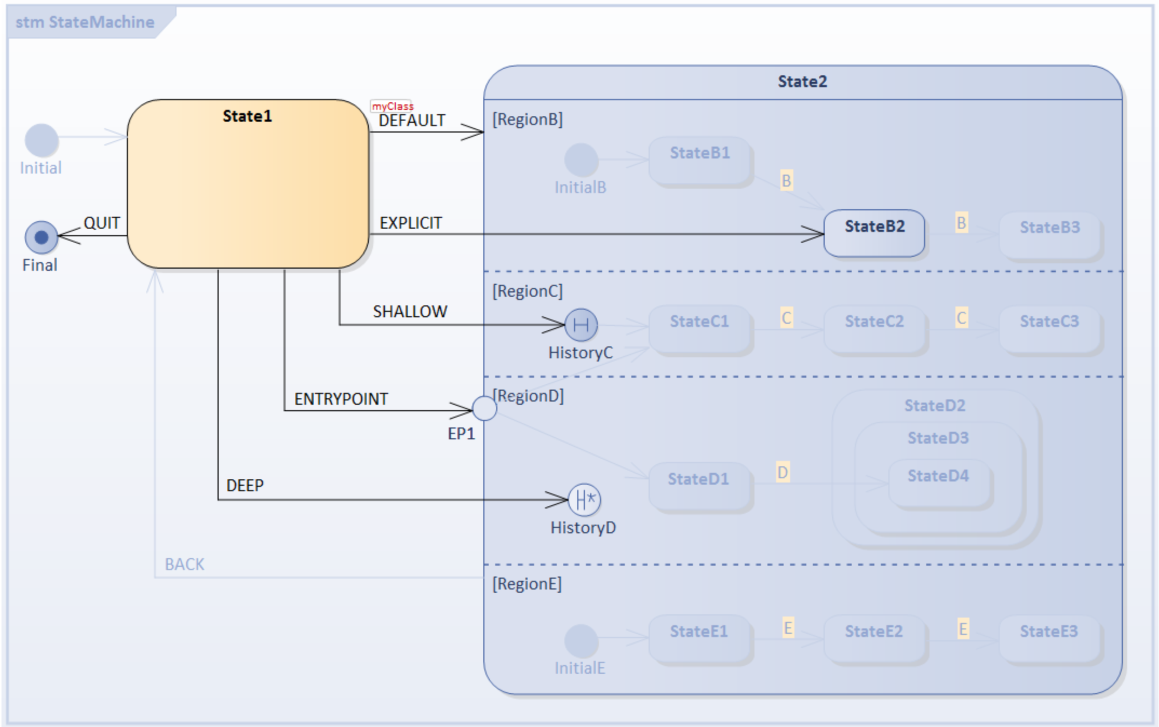
注记：该目录的内容在生成前会被清除；确保您指定的目录仅用于状态机模拟目的。



运行仿真

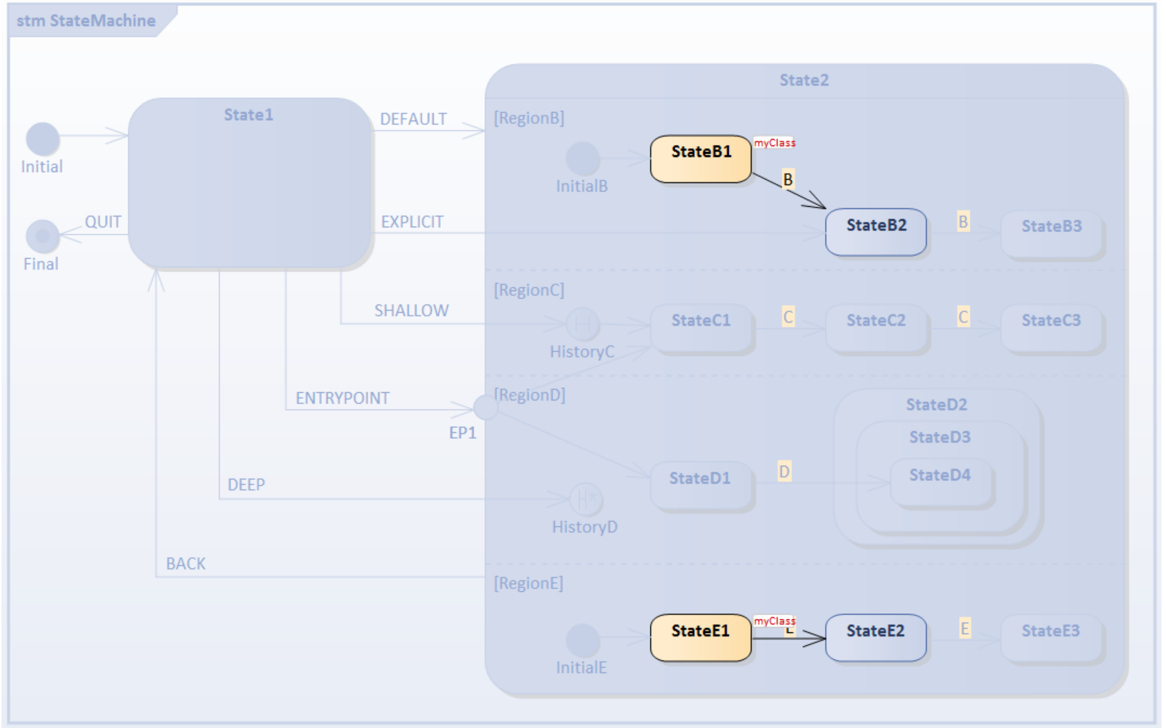
温馨提示：您可以在仿真窗口中查看执行序列，您可以通过选择 仿真>动态仿真仿真>模拟器>打开仿真窗口"功能区选项打开该窗口

模拟开始时， *State1*处于活动状态，状态机正在等待事件。



使用 "仿真动态仿真>事件"功能区选项打开仿真事件（触发器）窗口。

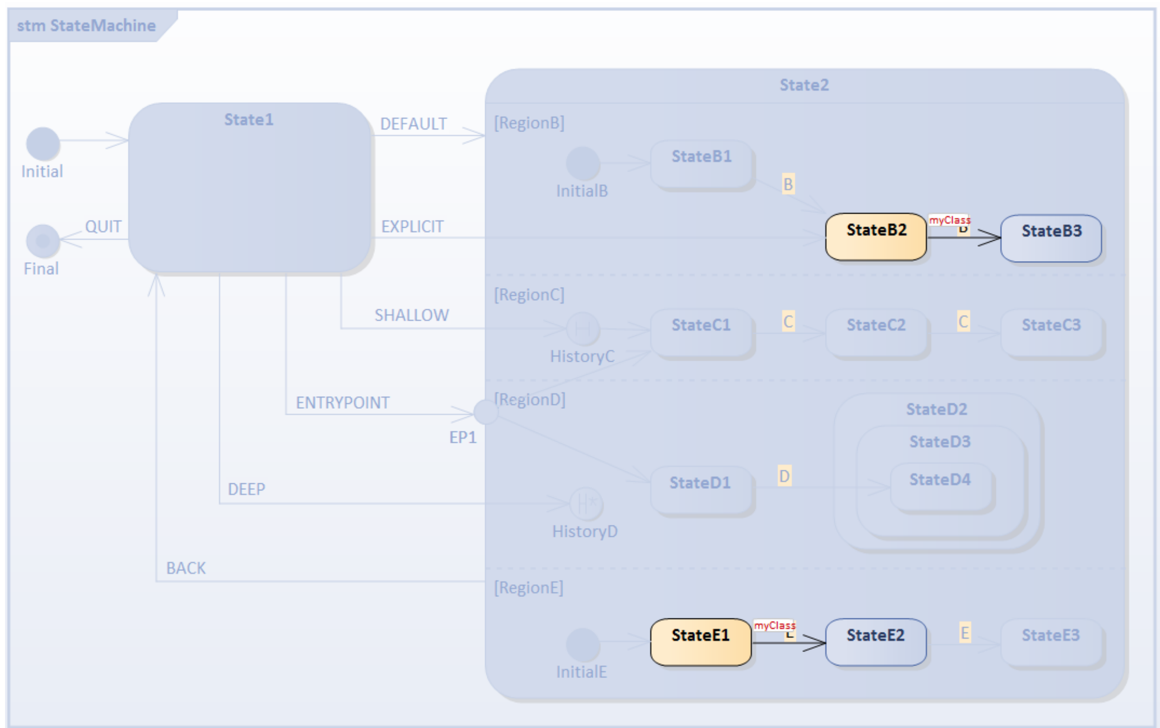
1）选择默认条目：触发器序列[DEFAULT]。



- *RegionB*被激活，因为它定义了*InitialB*；从它发出的转换将被执行， *StateB1*是活动状态
- *RegionE*被激活，因为它定义了*InitialE*；从它发出的转换将被执行， *StateE1*是活动状态
- *RegionC*和*RegionD*处于非活动状态，因为没有定义初始伪状态

选择触发器的[BACK]来重置。

2) 选择显式条目：触发器序列[EXPLICIT]。

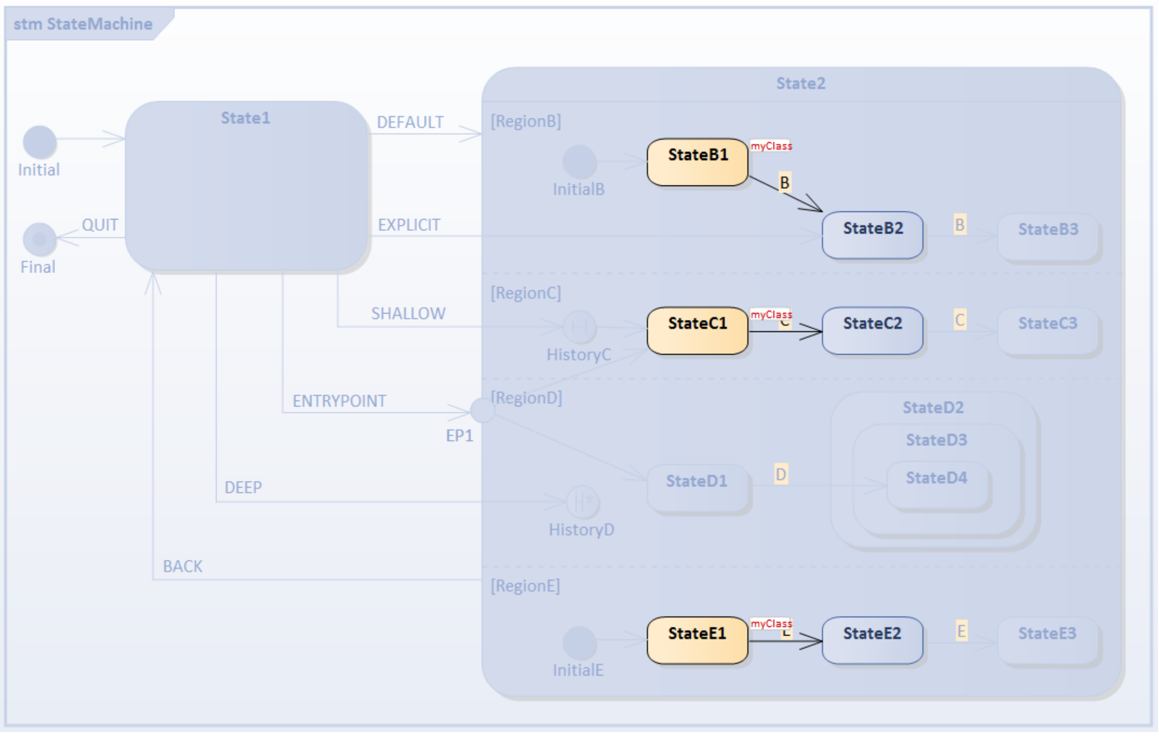


- *RegionB*被激活，因为过渡以包含的顶点*StateB2*为目标
- *RegionE*被激活，因为它定义了*InitialE* ；从它发出的转换将被执行， *StateE1*是活动状态
- *RegionC*和*RegionD*处于非活动状态，因为没有定义初始伪状态

选择触发器的[BACK]来重置。

3) 选择默认历史转移

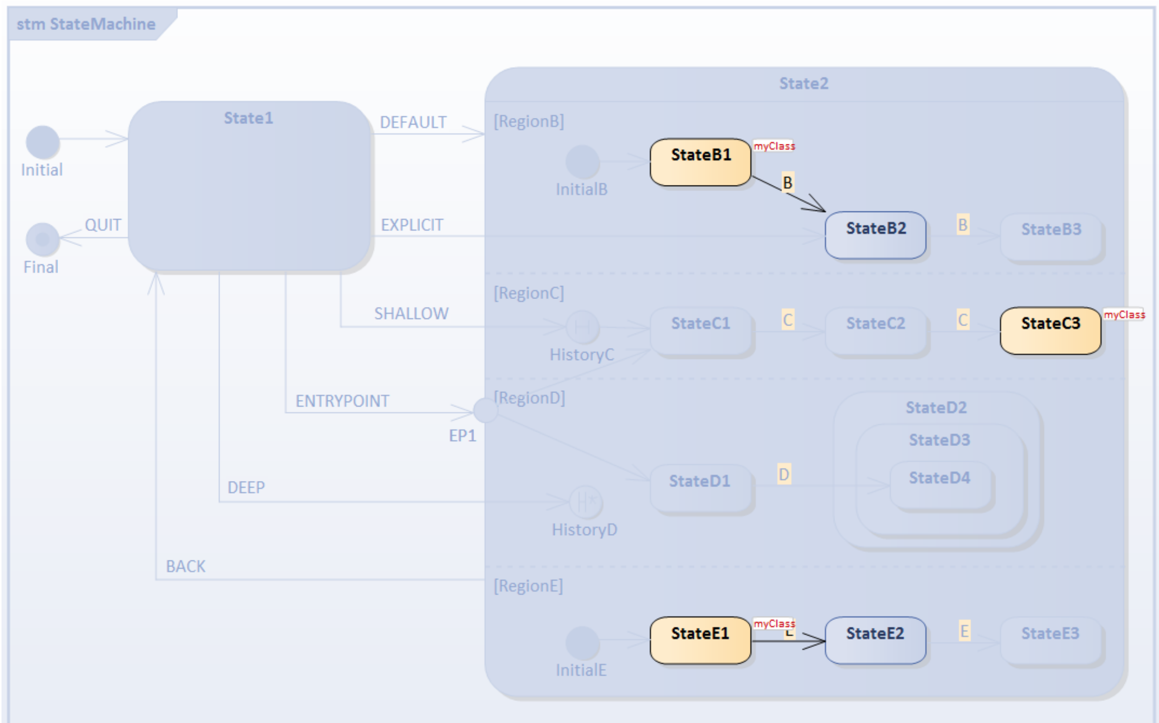
:简单的序列[触发器]



- *RegionC*被激活，因为过渡以包含的顶点*HistoryC*为目标；因为这个区域是第一次进入（并且历史伪状态没有什么要 记住"的），所以从历史C到状态*C1*的转换被执行

- *RegionB*被激活，因为它定义了*InitialB* ；从它发出的转换将被执行， *StateB1*是活动状态
- *RegionE*被激活，因为它定义了*InitialE* ；从它发出的转换将被执行， *StateE1*是活动状态
- *RegionD*处于非活动状态，因为未定义初始伪状态

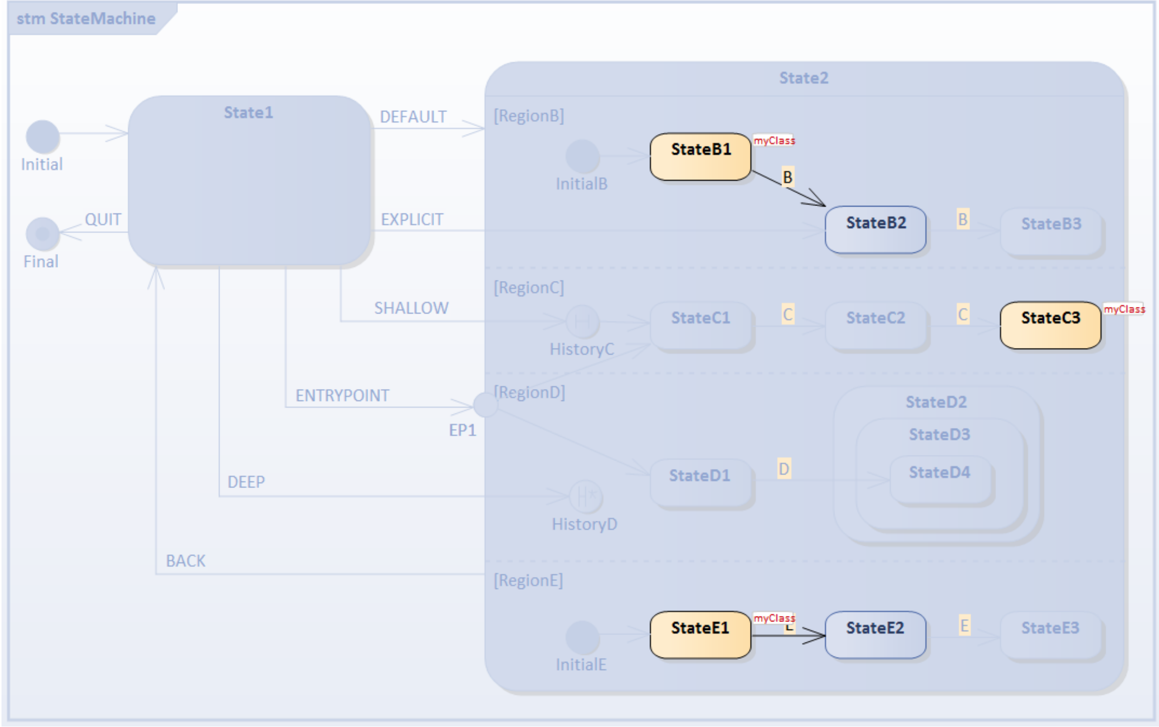
4) 准备测试Shallow History Entry：触发器序列[C, C]。



- 我们假设浅历史伪状态*HistoryC*可以记住*StateC3*

选择触发器的[BACK]来重置。

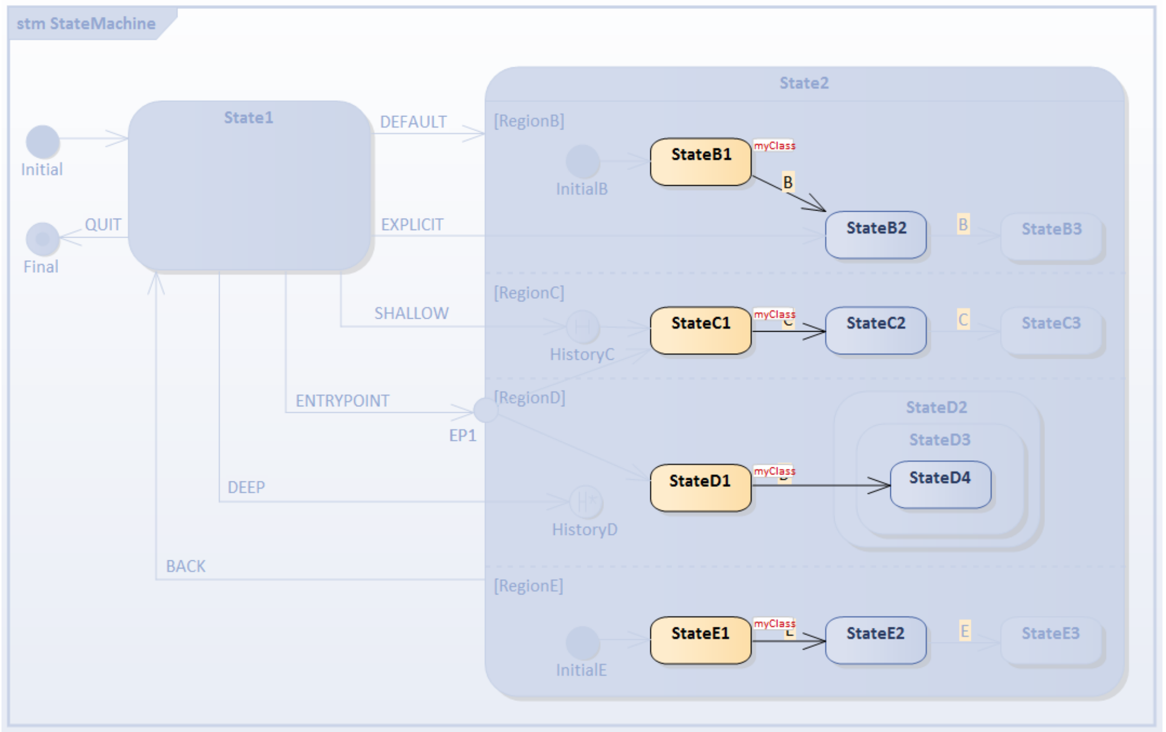
5) 选择Shallow History Entry：简单序列[触发器]。



- 对于*RegionC* ，直接激活*StateC3*

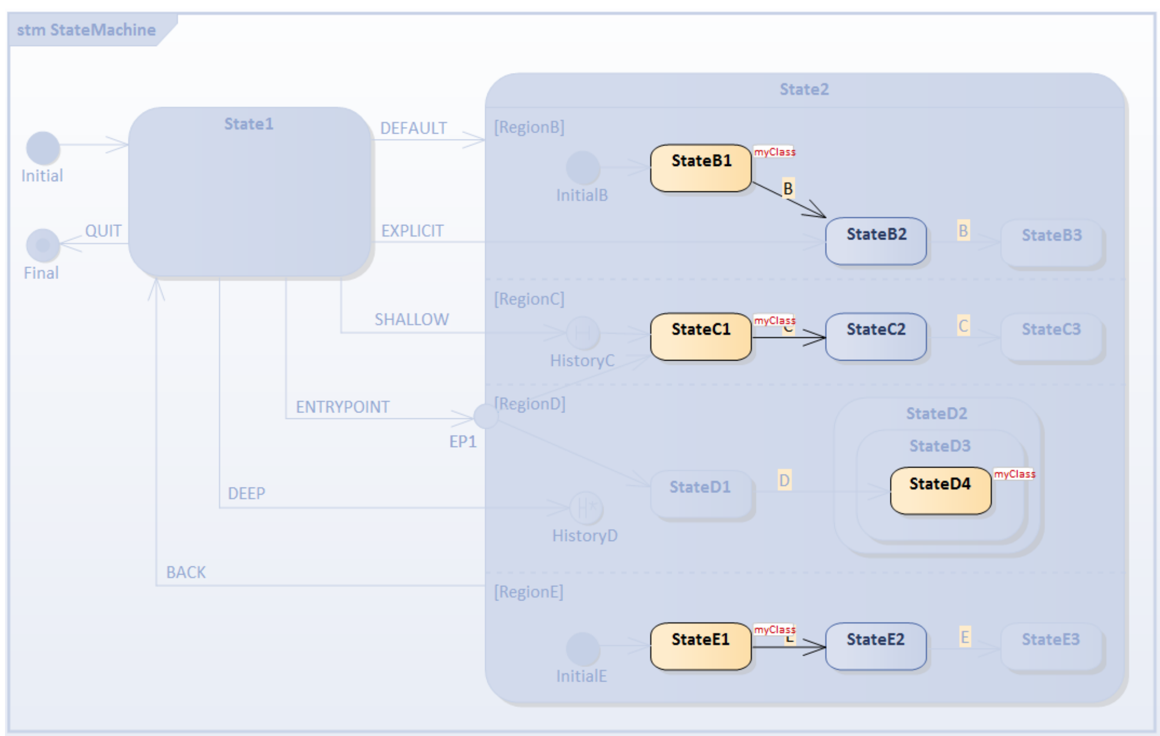
选择触发器的[BACK]来重置。

6) 选择入口条目：简单的序列[触发器]。



- *RegionC*被激活，因为从*EP1*的转换以包含的*StateC1*为目标
- *RegionD*被激活，因为从*EP1*的转换以包含的*StateD1*为目标
- *RegionB*被激活，因为它定义了*InitialB*；从它发出的转换将被执行， *StateB1*是活动状态
- *RegionE*被激活，因为它定义了*InitialE*；从它发出的转换将被执行， *StateE1*是活动状态

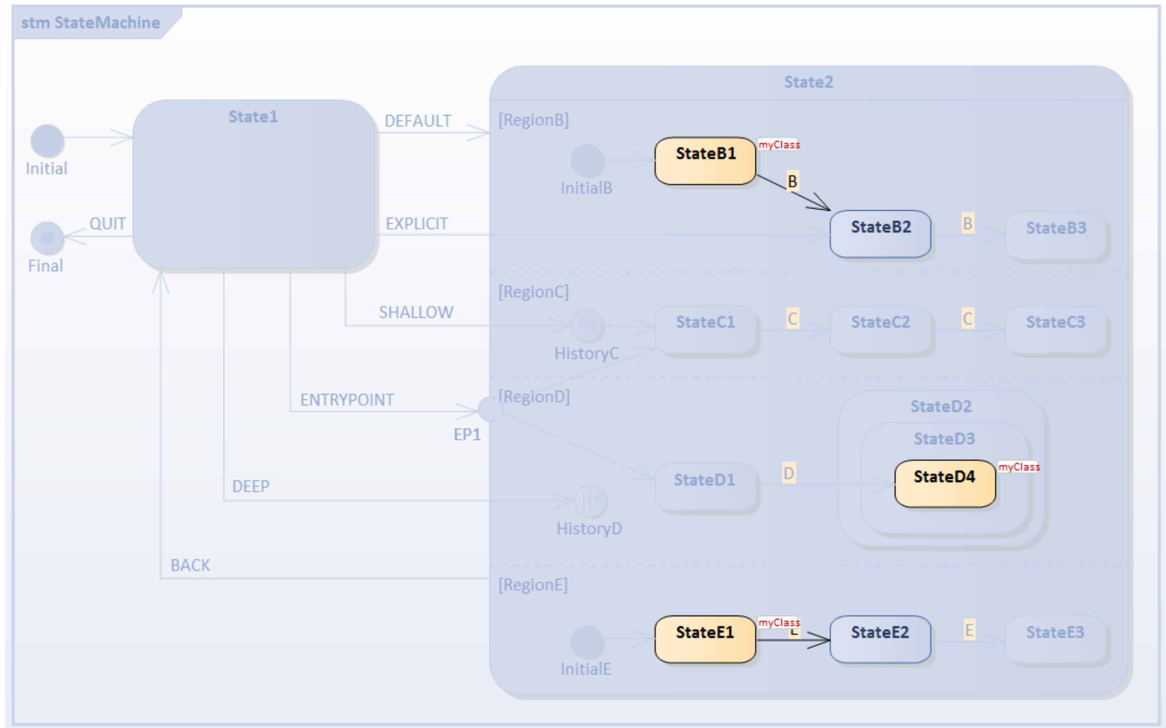
7) 准备测试Deep History：触发器序列[D]。



- 我们假设深度历史伪状态*HistoryD*可以记住*StateD2*、 *StateD3*和*StateD4*

选择触发器的[BACK]来重置。

8) 选择深度历史条目：触发器序列[DEEP]。



- 对于*RegionD*，输入*StateD2*、*StateD3*和*StateD4*；痕迹是：
  - myClass[MyClass].StateMachine_State1 退出

- myClass[MyClass]影响

- myClass[MyClass].StateMachine_State2 ENTRY

- myClass[MyClass].StateMachine_State2 DO

- myClass[MyClass].InitialE_105787__TO__StateE1_61746影响

- myClass[MyClass].StateMachine_State2_StateE1 条目

- myClass[MyClass].StateMachine_State2_StateE1 DO

- myClass[MyClass].InitialB_105785__TO__StateB1_61753影响

- myClass[MyClass].StateMachine_State2_StateB1 条目

- myClass[MyClass].StateMachine_State2_StateB1 DO

- myClass[MyClass].StateMachine_State2_StateD2 条目

- myClass[MyClass].StateMachine_State2_StateD2_StateD3 条目

- myClass[MyClass].StateMachine_State2_StateD2_StateD3_StateD4 条目

示例：分叉和汇合

分叉伪状态分裂一个传入的转移

成两个或多个Transitions，终止于一个复合状态的正交区域中的Vertices。从分叉伪状态传出的转换不能有保护或触发器，并且各个传出转换的影响行为至少在概念上是同时执行的。

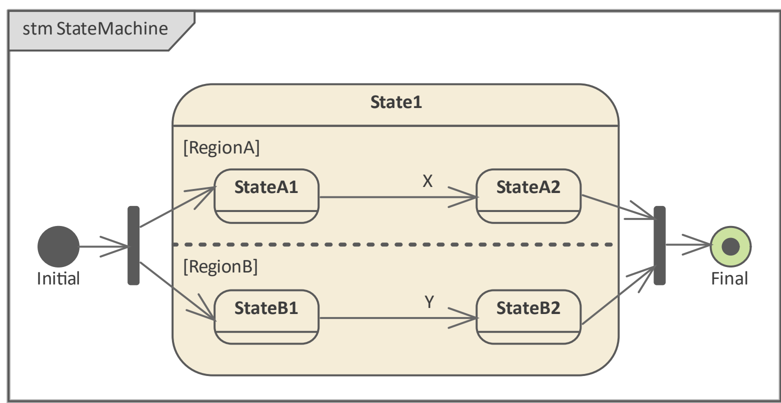
汇合函数合伪状态是来自不同正交区域中的顶点的两个或多个转换的公共目标顶点。汇合伪状态执行同步，因此所有传入的转换必须完成才能通过传出的转移

继续执行转移

.

在这个例子中，我们演示了具有分叉和汇合伪状态的状态机的行为。

建模状态机



状态机的上下文

- 创建一个名为*MyClass*的类元素，作为状态机的上下文
- 在浏览器窗口中右键单击*MyClass*并选择"添加|状态机"选项

状态机

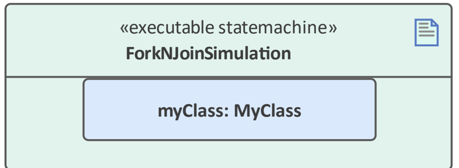
- 在图中添加一个初始节点、一个分叉、一个名为*State*的状态、一个汇合和一个终点
- 放大*State1* ，右键单击它 在图表上选择'高级|定义并发子状态 |定义'选项并定义*RegionA*和*RegionB*
- 在*RegionA*中，定义*StateA1* ，转换到*StateA2* ，由事件*X*触发
- 在*RegionB*中，定义*StateB1* ，转换到*StateB2* ，由事件*Y*触发
- 绘制其他转换：初始到分叉；分叉到划和*StateB1* ； *StateA2* 和 *StateB2*到汇合；汇合到终点

仿真

工件

Enterprise Architect支持 C、C++、C#、 Java和JavaScript ；我们将在此示例中使用JavaScript ，因为我们不需要安装编译器（对于其他语言，需要 Visual Studio 或 JDK ）。

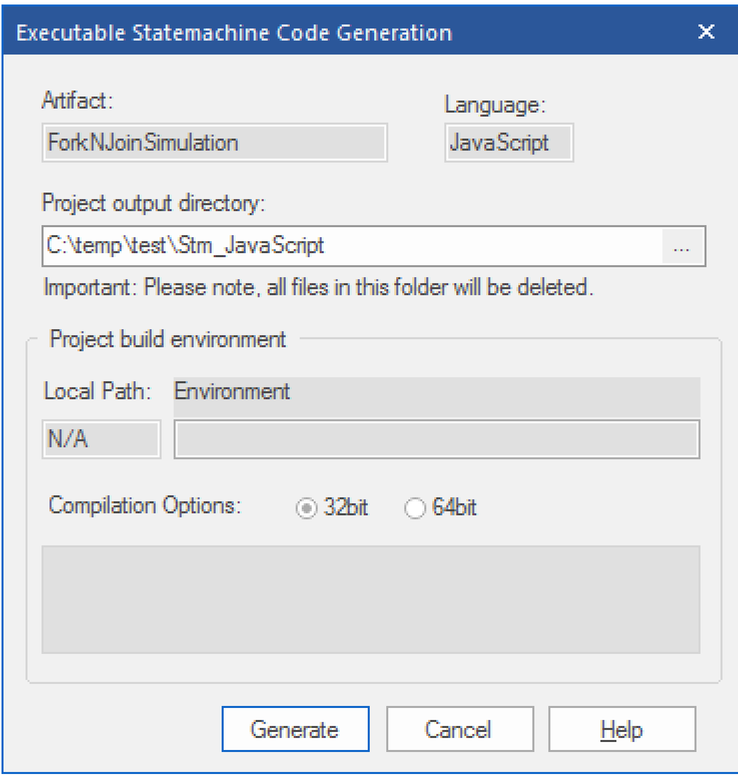
- 从图表中选择 "工具箱仿真页面，然后将绘图工具拖到可执行状态机上，创建一个工件；将其命名为*ForkNJoinSimulation*并将其 "语言"字段设置为 "JavaScript "
- 从浏览器窗口 Ctrl+ 将*MyClass*拖放到属性*ForkNJoinSimulation D*工件；将其命名为*myClass*



代码生成

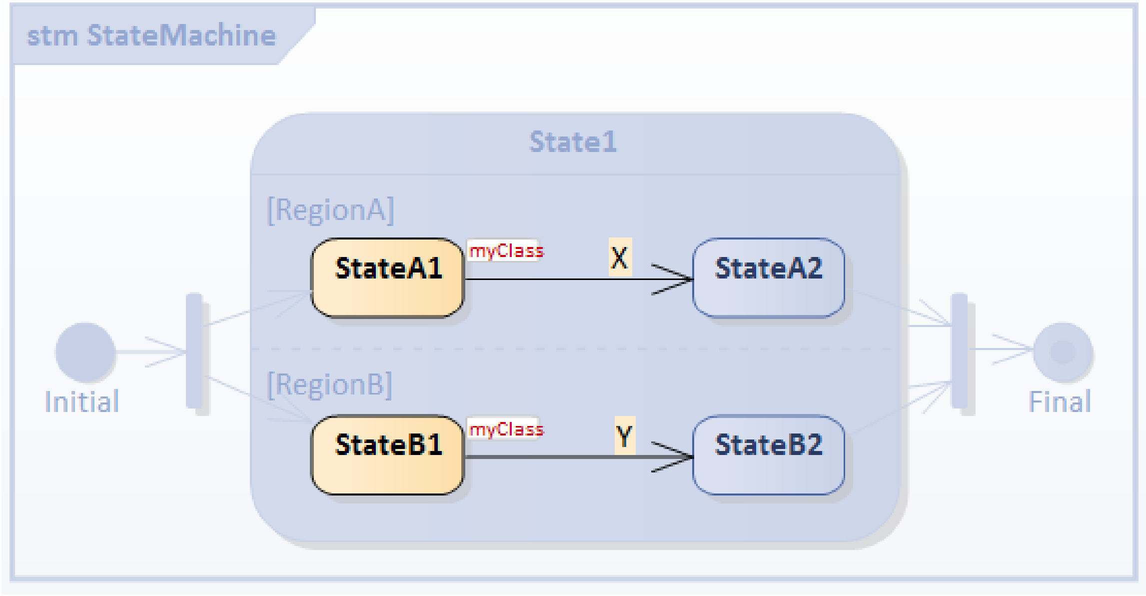
- 单击*ForkNJoinSimulation*并选择 仿真> 可执行状态 > 状态机 >生成、编译和运行"功能区选项
- 为生成的源代码指定目录

注记：该目录的内容在生成前会被清除；确保您指向的目录仅用于状态机模拟目的。



运行仿真

当仿真开始时， *State1* 、 *StateA1*和*StateB1*处于活动状态，状态机正在等待事件。



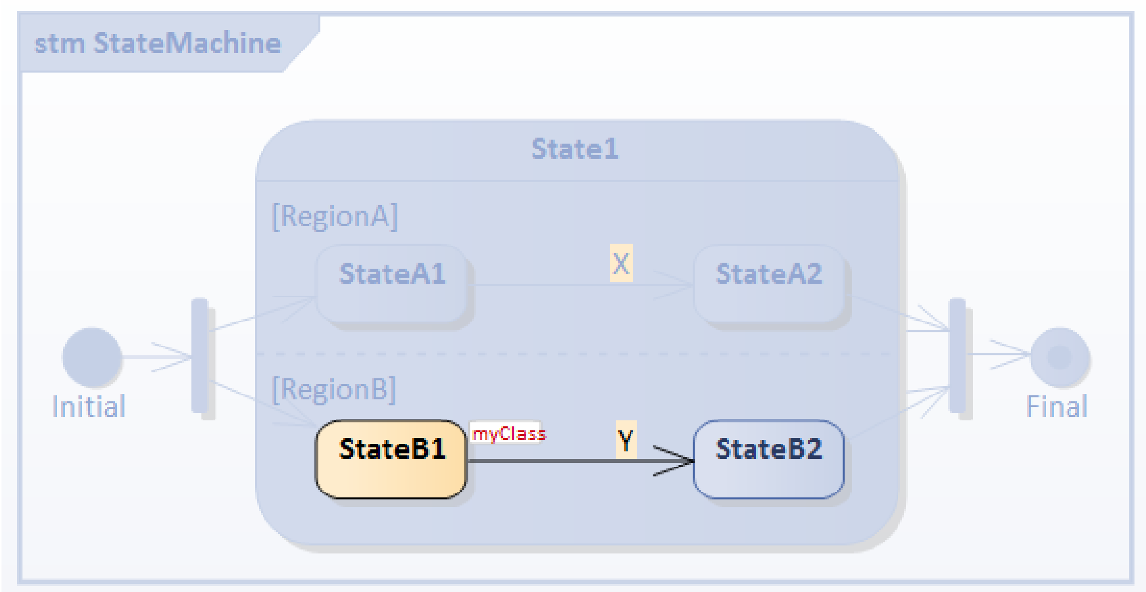
选择 仿真>动态仿真>事件”功能区选项，显示仿真事件窗口。

简单来说，在事件*X*中，触发器会退出并进入*StateA2* ；在 entry 和 doActivity 行为运行后，运行的完成事件被调度和回调。然后启用并遍历从*StateA2*到汇合伪状态的转换。

注记：汇合必须等待所有传入的转换完成，然后才能通过传出的转移

继续执行转移

.由于*RegionB*的分支没有完成（因为*StateB1*仍然处于活动状态，等待触发），此时不会执行从汇合终端到终点的转换。



简单来说，在事件*Y*时，触发器将退出并进入*StateB2* ；在 entry 和 doActivity 行为运行后，运行的完成事件被调度和回调。然后启用并遍历从*StateB2*到汇合伪状态的转换。满足所有传入的过渡都已完成的条件，则执行从汇合到汇合终点的过渡。仿真结束。

提示：您可以从仿真窗口（'仿真>动态仿真仿真>模拟器>打开仿真窗口'功能区选项）查看执行轨迹序列。

myClass[MyClass].Initial_82285__TO__fork_82286_82286_61745影响

myClass[MyClass].StateMachine_State1 条目

myClass[MyClass].StateMachine_State1 DO
myClass[MyClass]影响
myClass[MyClass].StateMachine_State1_StateA1 条目
myClass[MyClass].StateMachine_State1_StateA1 DO
myClass[MyClass]影响
myClass[MyClass].StateMachine_State1_StateB1 条目
myClass[MyClass].StateMachine_State1_StateB1 DO
触发器X
myClass[MyClass].StateMachine_State1_StateA1 退出
myClass[MyClass]影响
myClass[MyClass].StateMachine_State1_StateA2 条目
myClass[MyClass].StateMachine_State1_StateA2 DO
myClass[MyClass].StateMachine_State1_StateA2 退出
myClass[MyClass]影响
触发器
myClass[MyClass].StateMachine_State1_StateB1 退出
myClass[MyClass]影响
myClass[MyClass].StateMachine_State1_StateB2 条目
myClass[MyClass].StateMachine_State1_StateB2 DO
myClass[MyClass].StateMachine_State1_StateB2 退出
myClass[MyClass]影响
myClass[MyClass].StateMachine_State1 退出
myClass[MyClass]影响

示例：延迟事件模式

Enterprise Architect支持延迟事件模式。

在一个状态下创建一个延迟事件：

1. 为状态创建一个自我转换。
2. 更改向 内部"过渡的 种类"。
3. 简单地触发器要延迟的事件。
4. 在 影响"字段中，输入 defer();"。

仿真：

1. 选择 仿真>动态仿真仿真>模拟器>打开仿真窗口"。同时选择 仿真>动态仿真>事件"打开仿真事件窗口。
2. 模拟器事件窗口帮助您触发事件；双击 等待触发器"列中的触发器。
3. 仿真窗口以文本形式显示执行过程。您可以在 Simulator 命令行中键入 dump"以显示队列中延迟了多少事件；输出可能类似于：
   泳池]事件泳池：[NEW,NEW,NEW,NEW,NEW,]

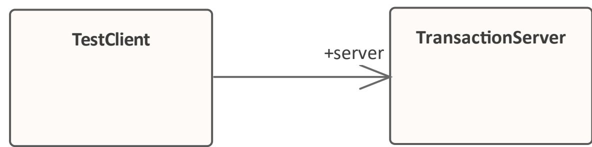
延迟事件示例

此示例显示了使用延迟事件的模型，以及显示所有可用事件的仿真事件窗口。

我们首先设置上下文（包含状态机的类元素），在一个简单的上下文模拟它们并从外部引发事件；然后用上下文事件机制模拟一个客户端-服务器时间。

创建上下文和状态机

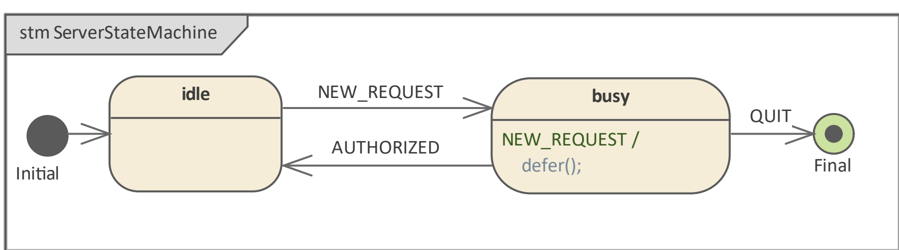
创建服务器上下文



创建类图并：

1. A类元素*TransactionServer* ，你添加一个状态机*ServerStateMachine* 。
2. A类元素*TestClient* ，向其中添加状态机*ClientStateMachine* 。
3. 从*TestClient*到*TransactionServer*的关联，目标角色名为*server*。

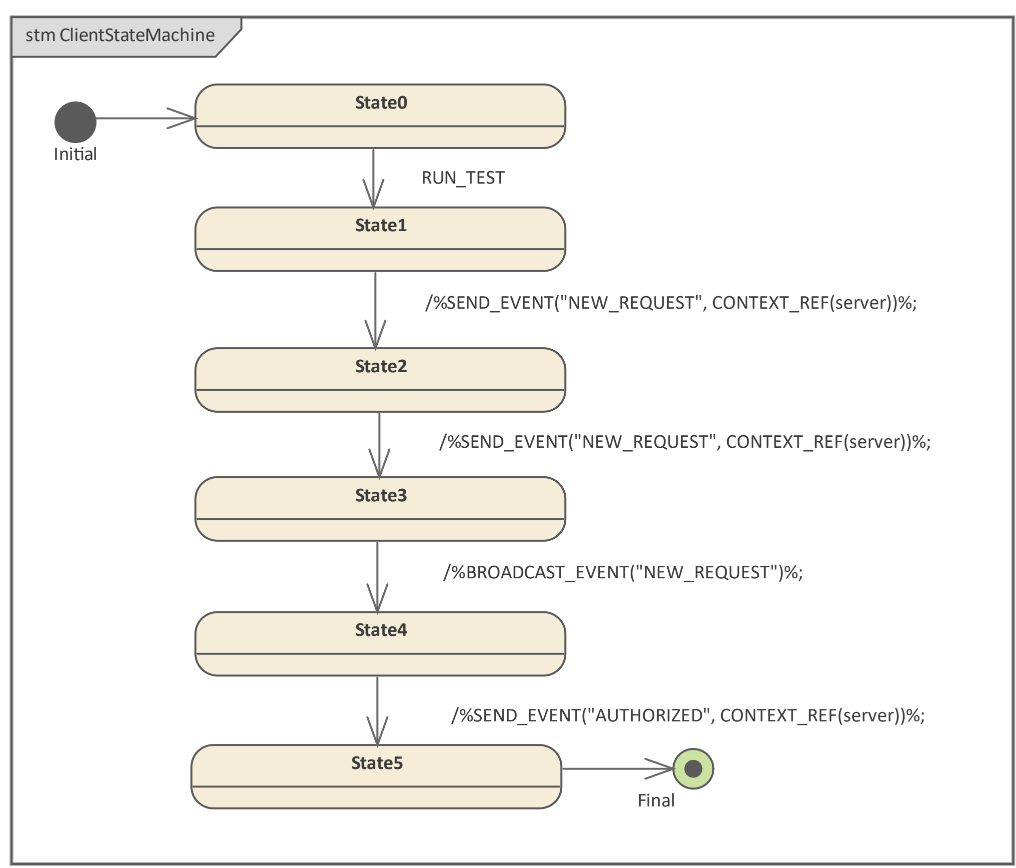
***ServerStateMachine*建模**



1. 在状态机图中添加一个 Initial节点*Initial* ，并转换到一个状态*idle* 。

2. 转移
   （以事件触发器为简单）到状态忙。
3. 转移
   （以事件为触发器）终点状态终点
4. 转移
   （与事件 AUTHORIZED 一样触发器）到*idle* 。
5. 转移
   （以事件触发器作为简单和*defer();*作为效果）到忙

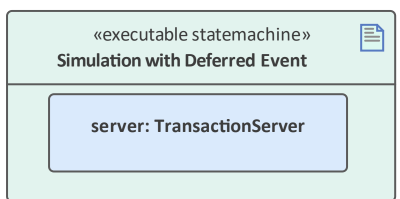
***ClientStateMachine*建模**



1. 在状态机图中添加一个 Initial节点*Initial* ，并转换到一个状态。
2. 转移
   （以事件*RUN_TEST* 作为触发器）到 *stateState1*状态
3. 转移
   （效果：%SEND_EVENT("NEW_REQUEST", CONTEXT_REF(server))%;) 到状态*State2*。
4. 转移
   （效果： %SEND_EVENT(" *NEW_REQUEST", CONTEXT_REF(server))%;)* 到 *stateState3*状态
5. 转移
   （效果： %BROADCAST_EVENT(" *NEW_REQUEST")%;)* 到 *stateState4*状态
6. 转移
   （效果： %SEND_EVENT("AUTHORIZED", CONTEXT_REF(server))%;)到 *stateState5*状态
7. 转移
   到终点状态终点

仿真在一个简单的上下文

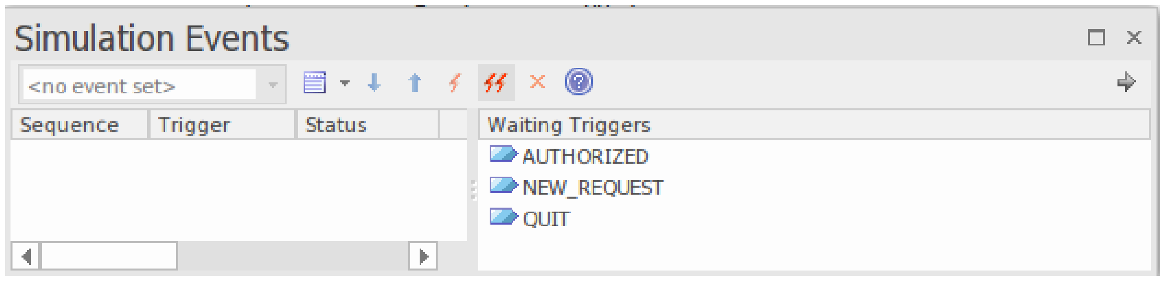
创造仿真工件



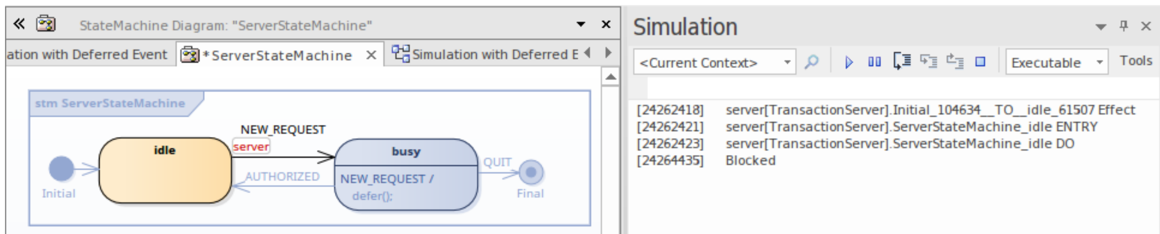
1. 工件and with the name仿真事件的可执行状态机将 语言"字段设置为*JavaScript*
2. 放大 Ctrl+元素将它拖到事务服务器上，然后将其作为属性与名称服务器工件粘贴。

运行仿真

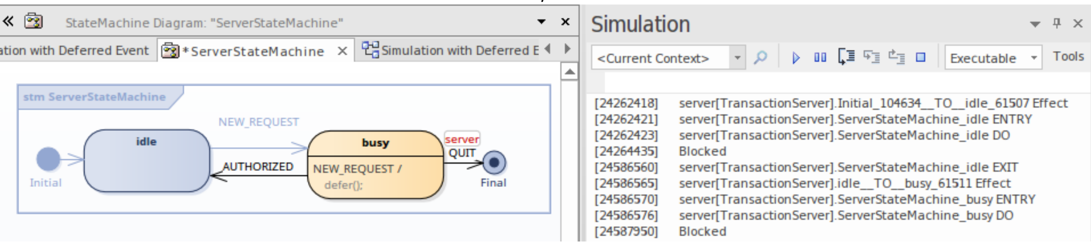
1. '选择工件编译仿真生成运行指定一个目录为你的（注记：目录中的所有文件将在模拟开始之前）。
2. 点击生成按钮。
3. 选择 仿真>动态仿真>事件"选项打开仿真事件窗口。



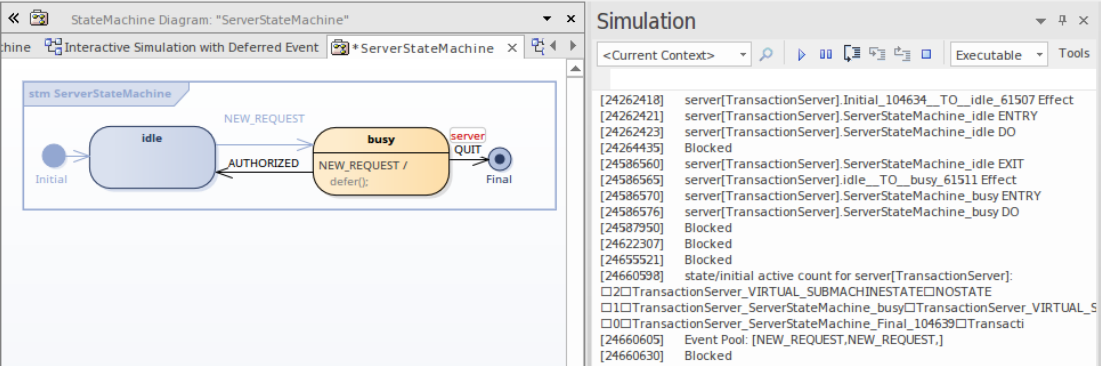
模拟开始时， *idle*将是活动状态。



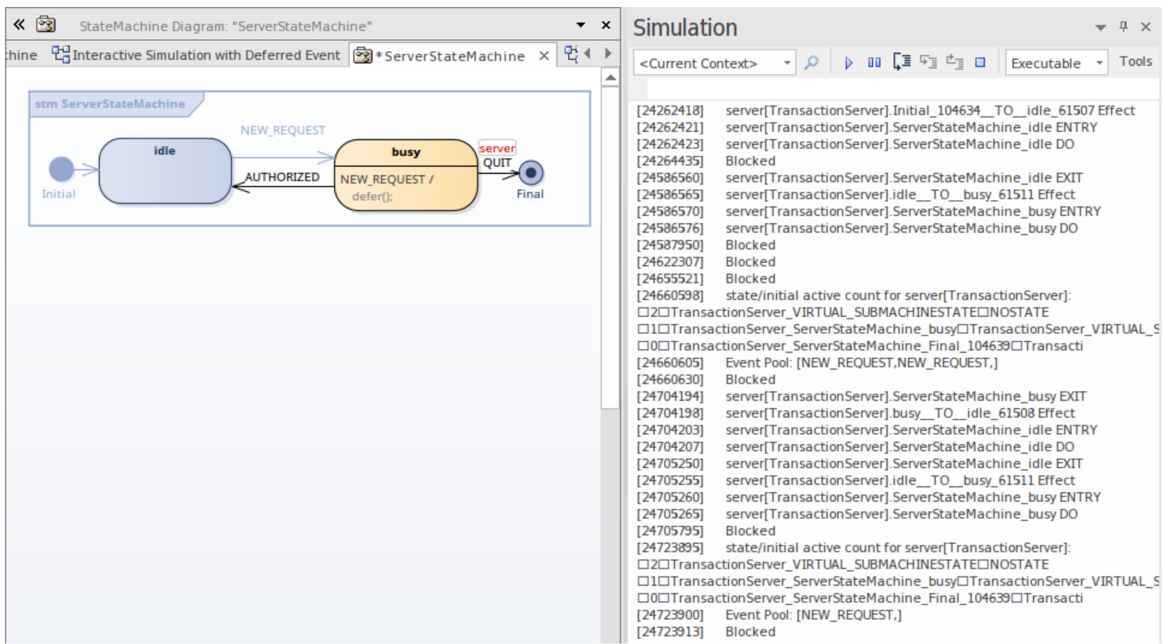
1. 双击触发器仿真将其作为事件简单的窗口执行；退出空闲并激活忙碌。



2. 双击仿真窗口中的仿真，再次像事件一样触发器； *busy*保持激活状态，并且泳池的实例被附加到事件池中。
3. 双击仿真事件窗口中的仿真，第三次执行触发器事件； *busy*保持激活状态，并且泳池的实例被附加到事件池中。
4. 仿真窗口命令行中的类型转储；请注意，事件池有两个 NEW_REQUEST 实例。

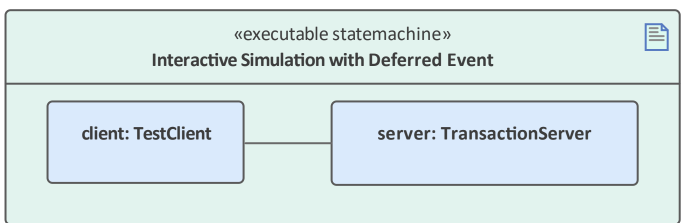


5. 双击触发器仿真中的AUTHORIZED，将其作为事件模拟窗口执行；这些行动发生：
 -*busy*退出， *idle*变为活动状态
 - 从池中检索 NEW_REQUEST 事件，退出空闲并且忙碌变为活动状态
6. 仿真窗口命令行中的类型转储；现在事件泳池只有一个现在实例。



通过发送/广播事件进行交互模拟

创造仿真工件

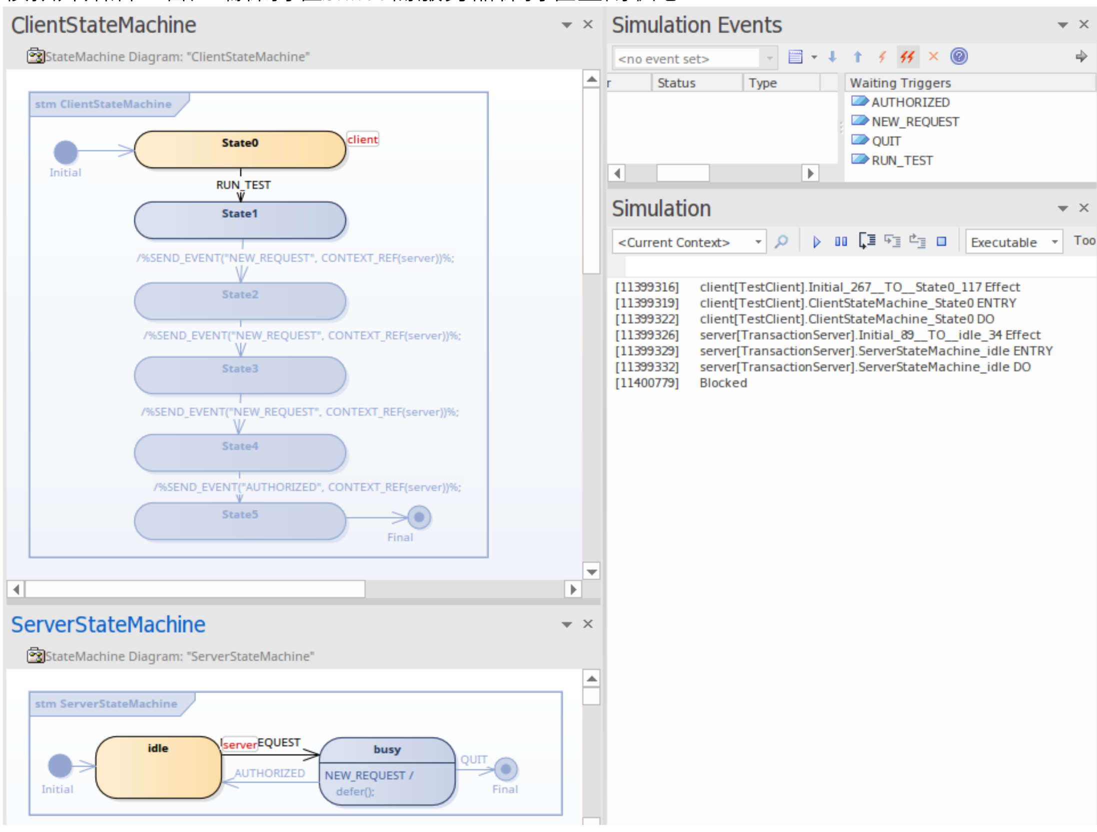


1. 使用名称创建事件仿真状态并可执行状态机到工件*JavaScript* 语言"字段集的名称放大元素。
2. Ctrl+D 将*TransactionServer*元素工件属性上，并将其粘贴为名称服务器。
3. Ctrl+工件将*TestClient*元素拖到属性上，并将其粘贴为名称*client* 。
4. 创建从客户端到服务器的连接器。
5. 单击连接器并按 Ctrl+L 以选择从*TestClient*元素到*TransactionServer*元素的关联。

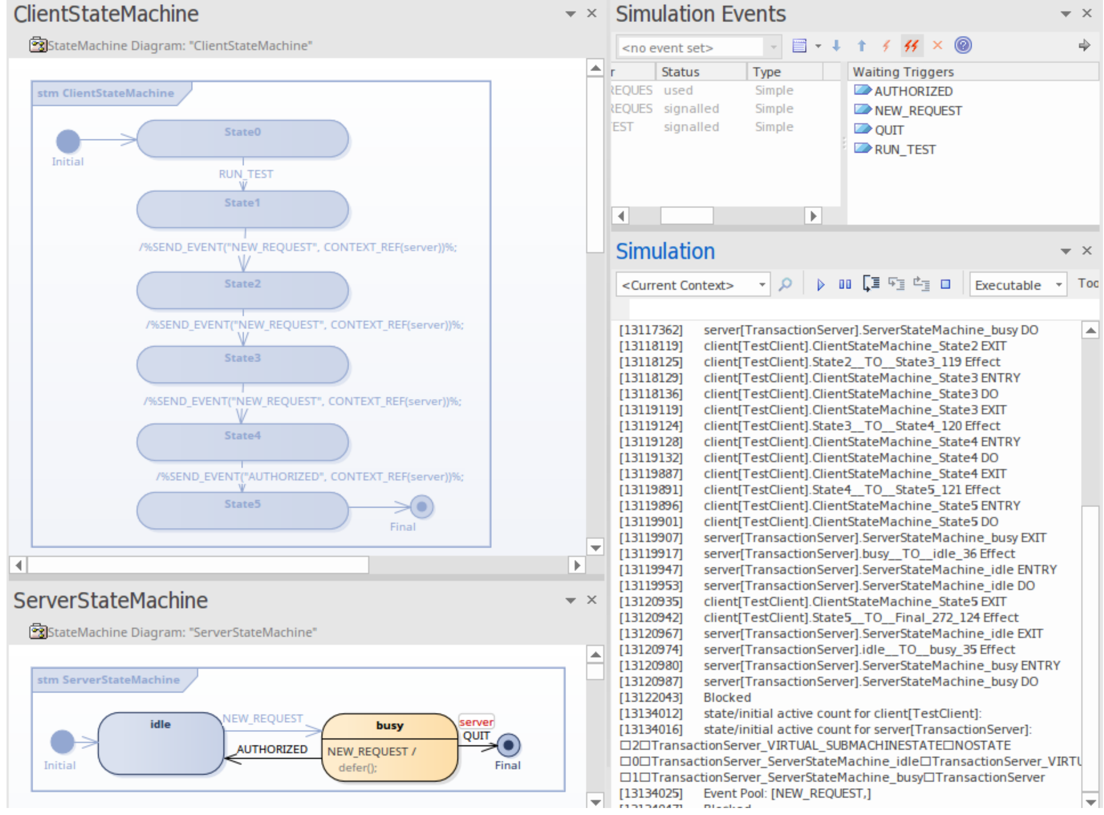
运行互动仿真

1. 以与简单上下文相同的方式启动模拟。

   模拟开始后，客户端保持在*State0*而服务器保持在空闲状态。



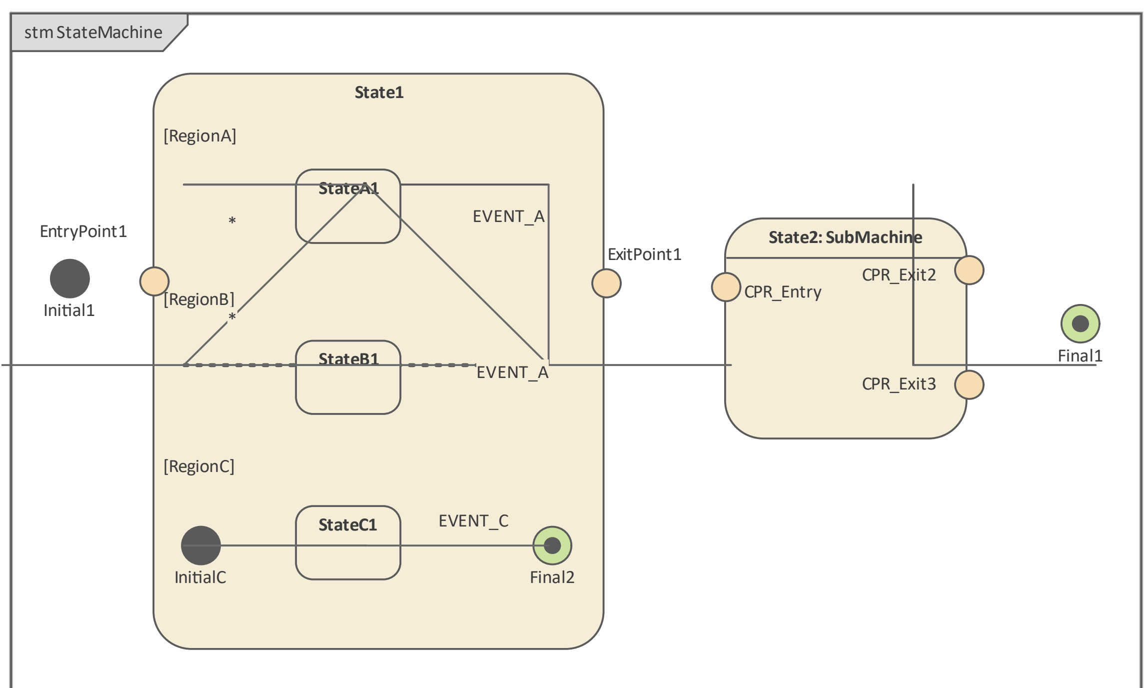
2. 双击仿真事件窗口中的 RUN_TEST 即可触发。事件 NEW_REQUEST 将被触发 3 次（由 SEND_EVENT 和 BROADCAST_EVENT），而 AUTHORIZED 将由 SEND_EVENT 触发一次。

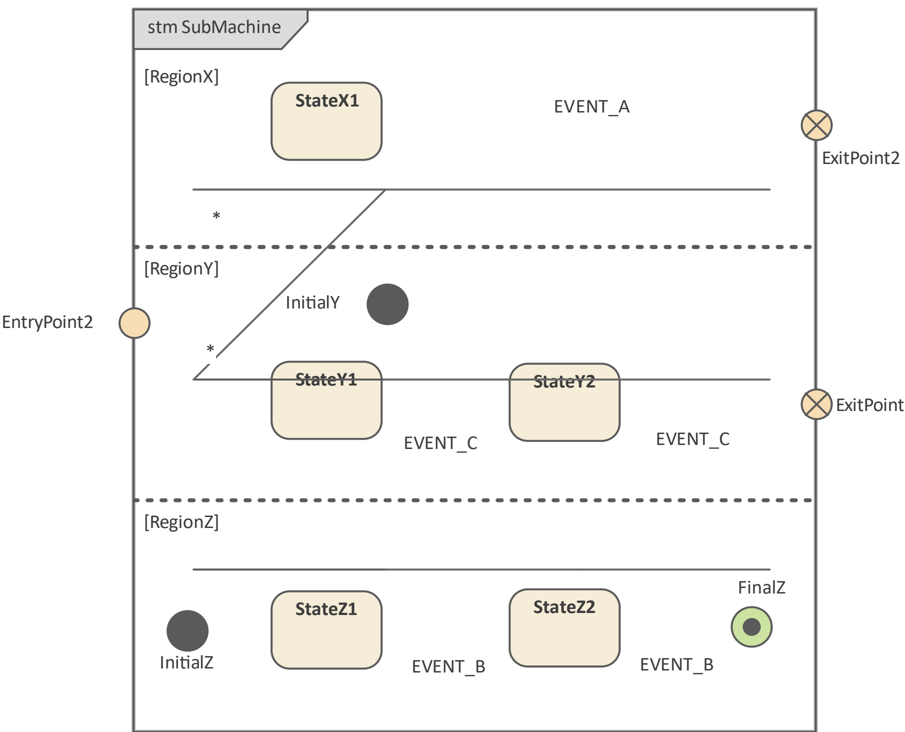


仿真窗口命令行中的类型转储，事件池中只剩下一个泳池实例。结果与我们的手动触发测试相匹配。

示例：入口和出口点（连接点参考）

Enterprise Architect提供对入口和出口点以及连接点引用的支持。在这个例子中，我们为*MyClass* 定义了两个状态机——状态机状态机*SubMachine*。





- *State1*是一个复合状态（也称为正交状态，因为它有多个区域），有三个区域： *RegionA* 、 *RegionB*和 *RegionC*
- *State2*是调用*SubMachine*的 SubMachine状态，它有三个区域： *RegionX* 、 *RegionY*和*RegionZ*
- 在*State1*上定义*EntryPoint1* ，激活三个区域中的两个；在*SubMachine*上定义*EntryPoint2*以激活三个区域中的

两个

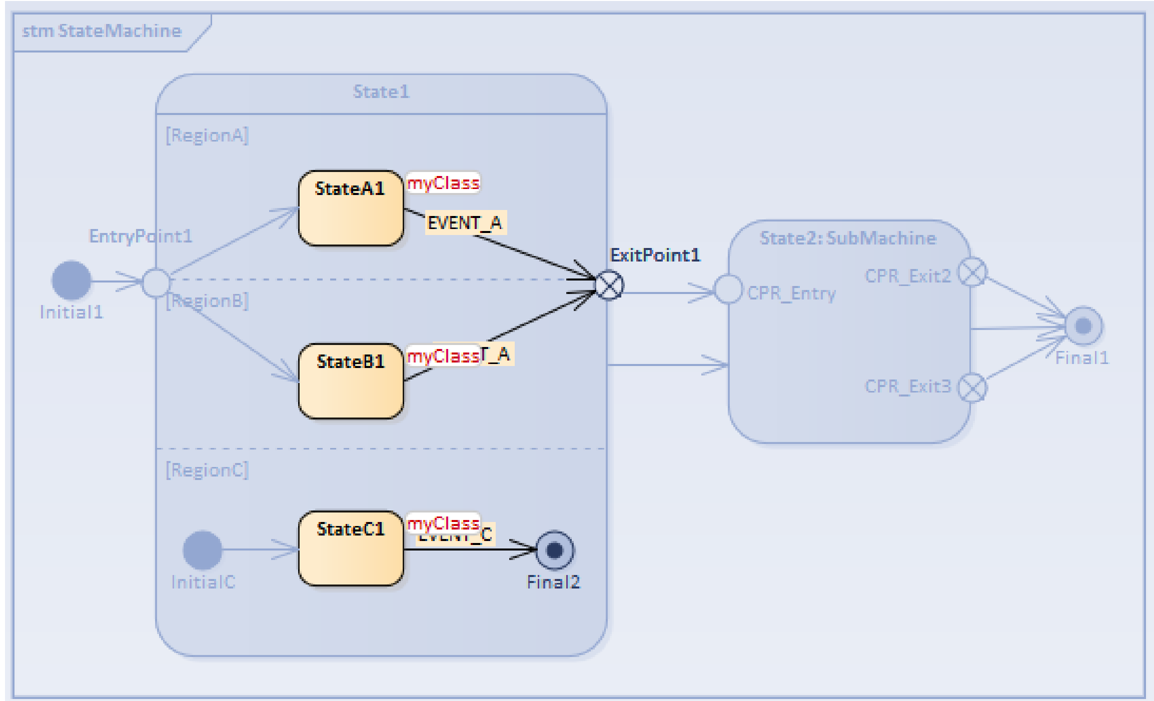
- *ExitPoint1*在*State1*上定义；在*SubMachine*上定义了两个出口点*ExitPoint2*和*ExitPoint3*
- 连接点引用在*State2*上定义并绑定到输入 SubMachine 的入口/出口点
- 定义初始节点以演示区域的默认区域

进入状态：入口进入

状态1 上的入口点 1

当一个转移

以*EntryPoint1*为目标已启用， *State1*被激活，然后是包含的区域。



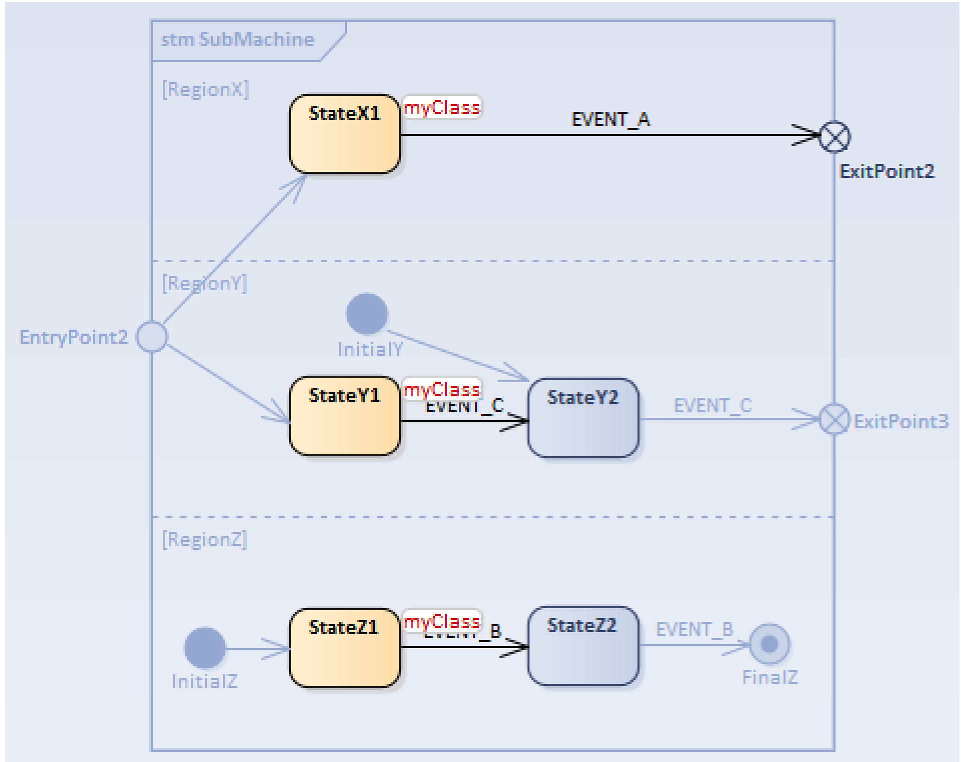
- *RegionA*和*RegionB*发生显式激活，因为它们中的每一个都由转移
  输入转移
  终止于区域包含的顶点之一
- *RegionC*发生默认激活，因为它定义了一个 Initial 伪状态*InitialC*和转移
  从*InitialC*到*StateC1*开始执行

SubMachine*上的*EntryPoint2

要模拟的序列是：[简单的触发器，EVENT_A]。

当一个转移

针对连接点参考*CPR_Entry* on *State2*被启用， *State2*被激活，随后通过绑定入口点激活SubMachine。



- 显式激活*RegionX*和*RegionY*，因为它们中的每一个都由转移
  输入转移
  终止于区域包含的顶点之一 - *RegionX*中的*StateX1*， *RegionY*中的*StateY1*
- *RegionZ*发生默认激活，因为它定义了 Initial 伪状态 *InitialZ*和转移
  从*InitialZ*到*StateZ1*开始执行

进入状态：默认进入

当复合状态是转移

的直接目标时，就会出现这种情况转移

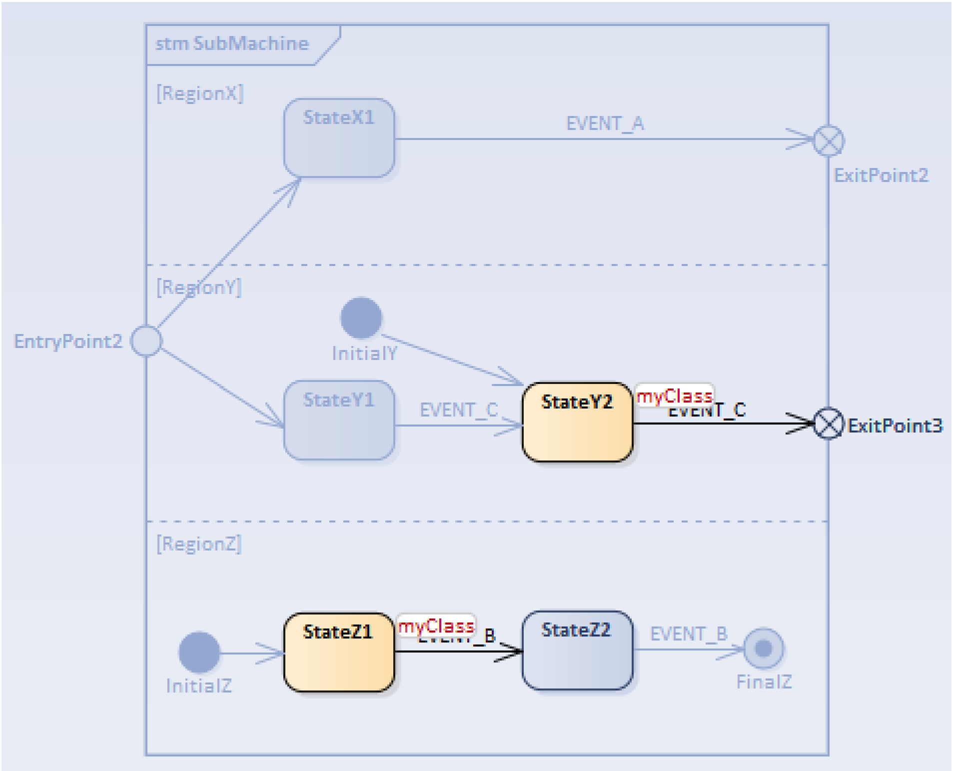
.

***State2*的默认入口**

要模拟的序列是：[简单的触发器，EVENTC]。

当一个转移

直接针对*State2*启用， *State2*被激活，随后默认激活区域的所有区域。



- *RegionX*的状态是不活跃的，因为它没有定义一个Initial节点
- *RegionY*通过*InitialY*和转移
  激活转移
  到*StateY2*被执行
- *RegionZ*通过*InitialZ*和转移
  激活转移
  到*StateZ1*被执行

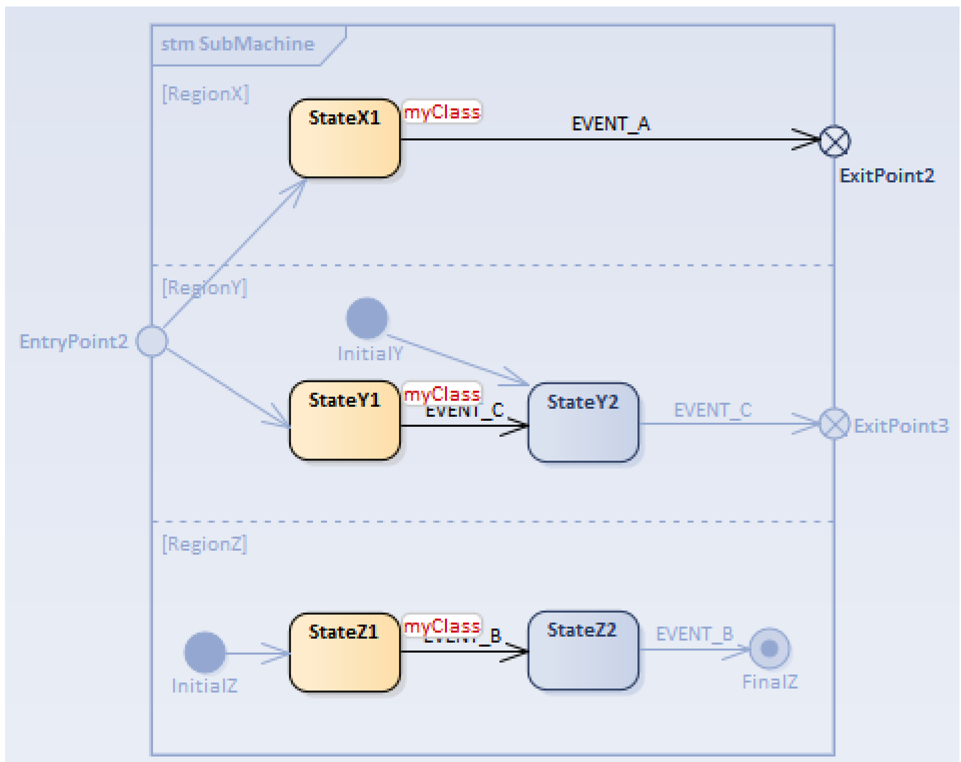
状态

State1 出口

- 触发器序列[EVENT_C, EVENT_A]：先去激活RegionC，再去激活*RegionA*和*RegionB* ； *State1*的退出行为执行后，转移
  从*ExitPoint1*传出已启用
- 触发器序列[EVENT_A, EVENT_C]：先去激活*RegionA*和*RegionB* ，再去激活*RegionC* ； *State1*的退出行为执行后，转移
  直接从*State1*传出已启用

State2 退出

触发器序列[EVENT_C, EVENT_A]，所以当前状态类似这样：



- 简单的触发器[EVENT_A,序列, EVENT_C, EVENT_B, EVENT_B]：先去激活RegionX，再去*RegionY*，最后去*RegionZ*；*State2*的退出行为执行后，转移
直接从*State2*传出已启用
- 简单的触发器[EVENT_A,序列, EVENT_B, EVENT_C, EVENT_C]：先去激活RegionX，再去*RegionZ*，最后去*RegionY*；*State2*的退出行为执行后，转移
启用从*CPR_Exit3*传出（*SubMachine*上的*ExitPoint3*绑定到*State2*的*CPR_Exit3*）
- 简单的触发器[EVENT_C,序列, EVENT_B, EVENT_B, EVENT_A]：先去激活*RegionY*，然后去*RegionZ*，最后去RegionX；*State2*的退出行为执行后，转移
从*CPR_Exit2*传出已启用（*SubMachine*上的*ExitPoint2*绑定到*State2*的*CPR_Exit2*）

示例：历史伪状态

状态历史是一个与复合状态区域相关联的方便概念，其中区域跟踪状态上次退出时所处的配置。如有必要，当区域下一个变为活动状态时（例如，从处理中断返回后），或者如果存在本地转移

，这允许轻松返回到该状态配置转移

回到它的历史。

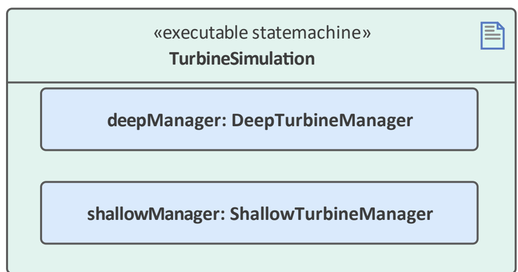
Enterprise Architect支持两种类型的历史伪状态：

- Deep History - 代表最近访问包含区域的完整状态配置；效果和转移
  一样转移
  终止于状态Pseudostate 相反，终止于保留状态配置的最内层状态，包括执行沿途遇到的所有入口行为
- 浅历史 - 表示仅返回最近状态配置的最顶层子状态，使用默认输入规则输入

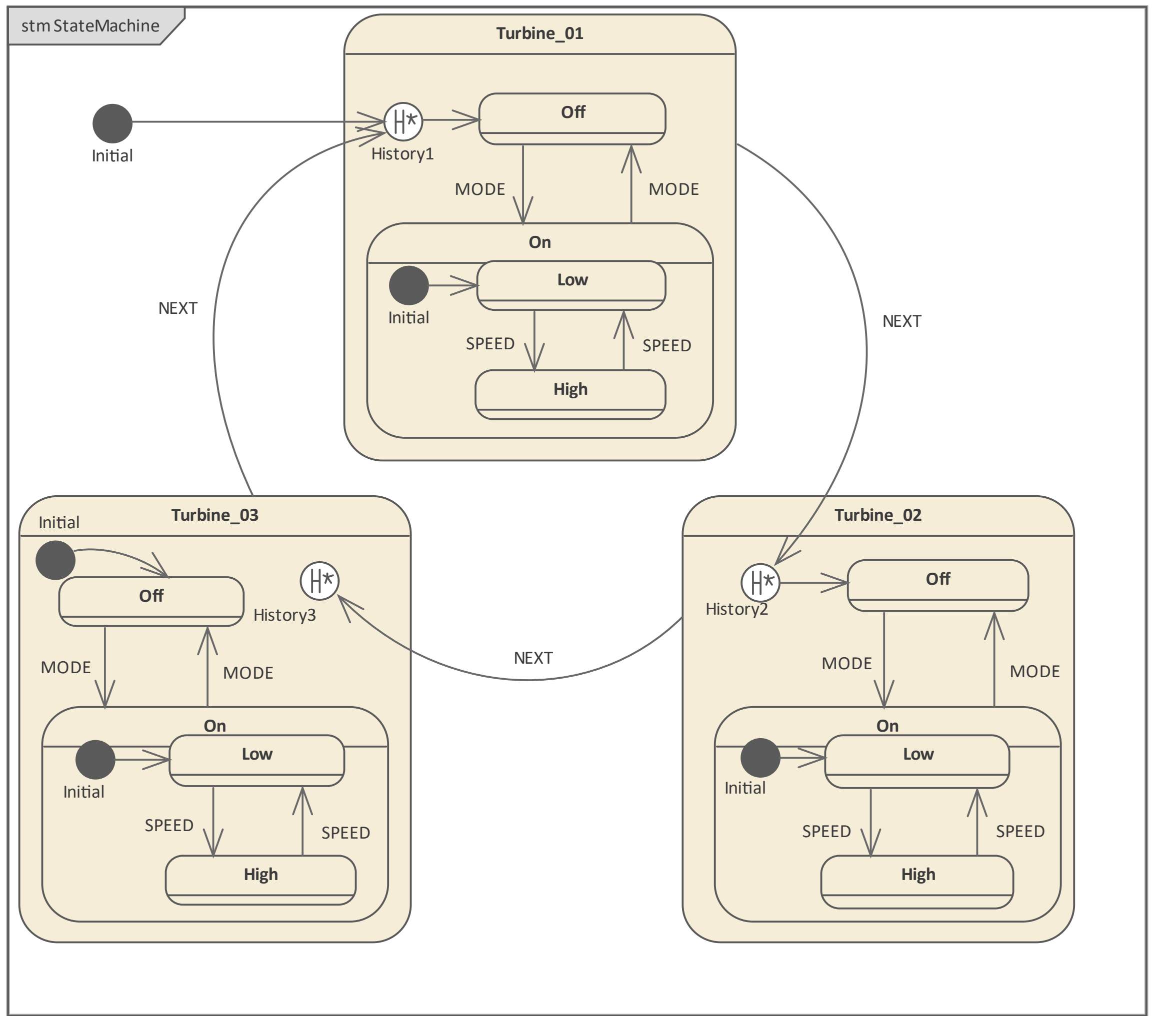
在此示例中， *DeepTurbineManager*和*ShallowTurbineManager*类完全相同，只是第一个包含的状态机具有 deepHistory Pseudostate，而第二个包含的状态机具有 shallowHistory Pseudostate。

两个状态机都有三个复合状态： *Turbine_01* 、 *Turbine_02*和*Turbine_03* ，每个状态机在其区域中都有*Off*和*On*状态以及 History Pseudostate。

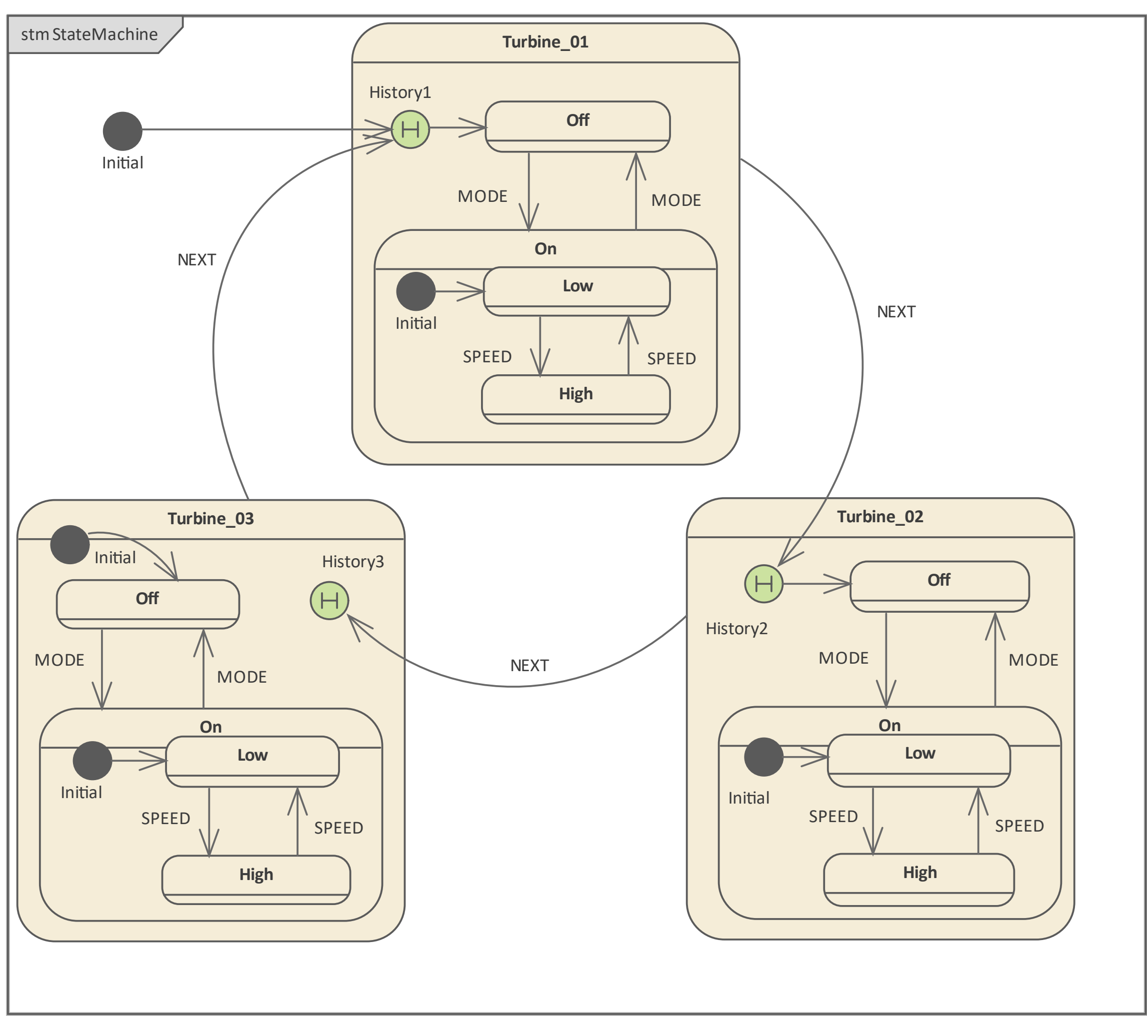
为了更好地观察 Deep History 和 Shallow History 的区别，我们在一次模拟中执行了两个状态机。



*DeepTurbineManager*中的状态机如下图所示：



*ShallowTurbineManager*中的状态机如下图所示：

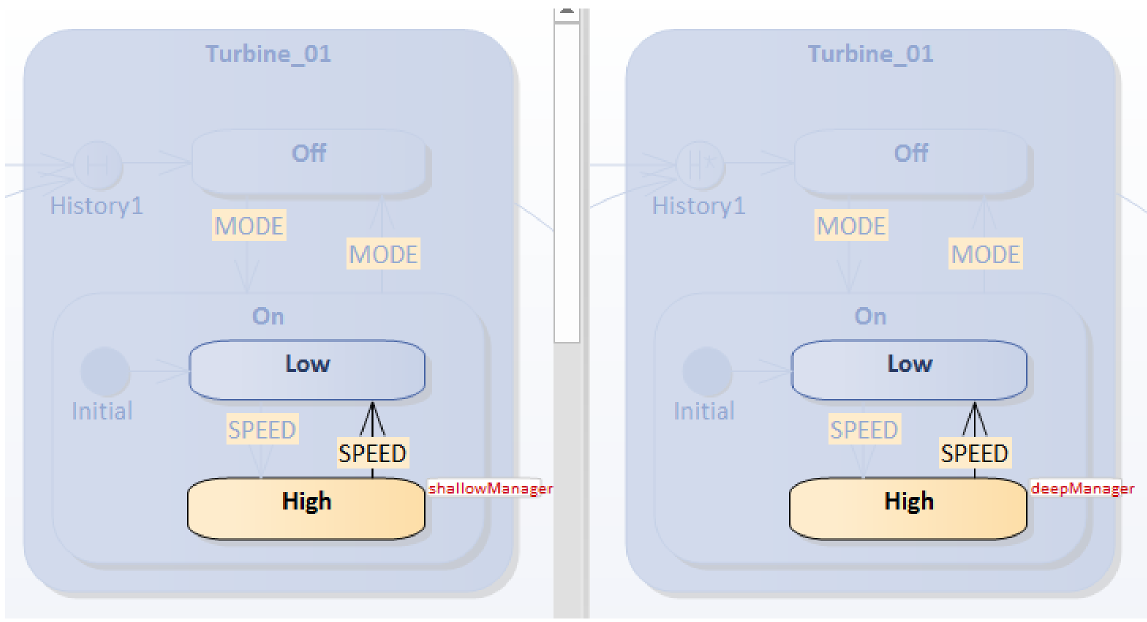


提示：如果您右键单击图表上的历史节点并选择'高级|深度历史选项，您可以在浅层和深层之间切换历史伪状态的类型。

第一时间激活状态

仿真开始后，*Turbine_01*及其子状态*Off*被激活。

触发器：[MODE,序列]

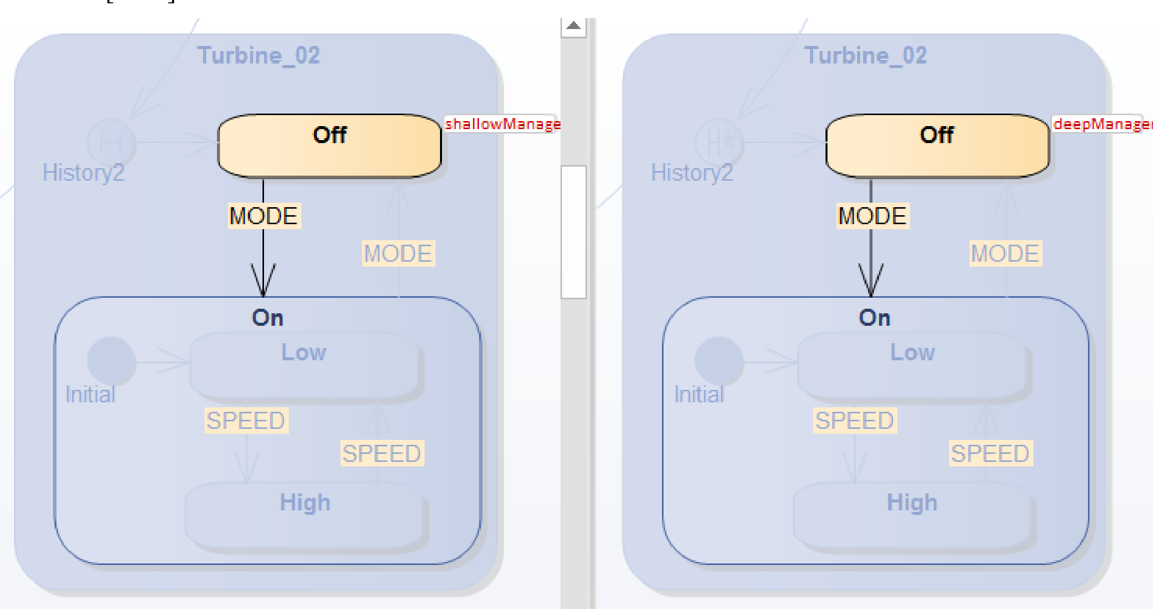


那么活动状态配置包括：

- 涡轮_01
- Turbine_01.On
- Turbine_01.On.High

这适用于*deepManager*和*shallowManager*。

触发器：[序列]



从仿真窗口（仿真>动态仿真仿真>模拟器>打开仿真窗口）可以观察到这条序列：

01 shallowManager[ShallowTurbineManager].StateMachine_Turbine_01_On_High 退出
02 shallowManager[ShallowTurbineManager].StateMachine_Turbine_01_On EXIT
03 shallowManager[ShallowTurbineManager].StateMachine_Turbine_01 退出

04 shallowManager[ShallowTurbineManager].Turbine_01__TO__History2_105720_61730影响

05 shallowManager[ShallowTurbineManager].StateMachine_Turbine_02 ENTRY

06 shallowManager[ShallowTurbineManager].StateMachine_Turbine_02 DO

07 shallowManager[ShallowTurbineManager].History2_105720__TO__Off_61731影响

08 shallowManager[ShallowTurbineManager].StateMachine_Turbine_02_Off ENTRY

09 shallowManager[ShallowTurbineManager].StateMachine_Turbine_02_Off DO

注记：由于*deepManager*和 shallowManager 的 trace 完全一样，所以*deepManager*的*trace*是从这个序列中过滤掉的。

我们可以了解到：

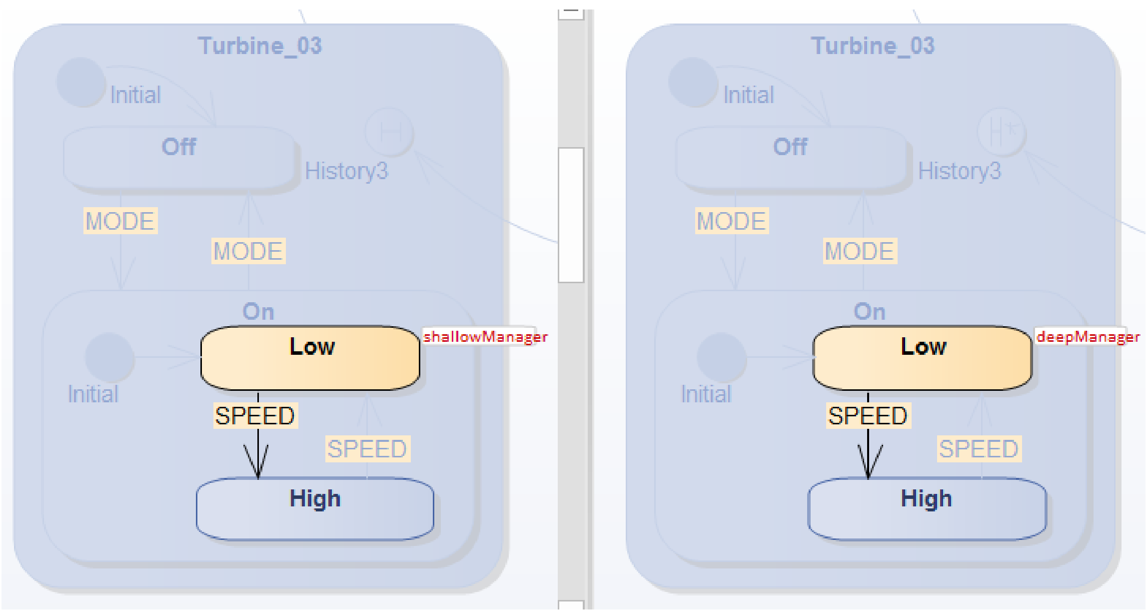
- 退出复合状态从活动状态配置中的最状态开始（参见跟踪序列中的第 01 - 03 行）
- 默认历史转移
  仅当执行导致历史节点（参见第 04 行）并且该状态之前从未处于活动状态（参见第 07 行）时才会执行

那么活动状态配置包括：

- 涡轮_02
- Turbine_02.Off

这适用于*deepManager*和*shallowManager*。

触发器：[序列,MODE]



从仿真窗口可以观察到这个序列：

触发器[下一个]

01 shallowManager[ShallowTurbineManager].StateMachine_Turbine_02_Off 退出

02 shallowManager[ShallowTurbineManager].StateMachine_Turbine_02 退出

03 shallowManager[ShallowTurbineManager].Turbine_02__TO__History3_105713_61725影响

04 shallowManager[ShallowTurbineManager].StateMachine_Turbine_03 ENTRY

05 shallowManager[ShallowTurbineManager].StateMachine_Turbine_03 DO

06 shallowManager[ShallowTurbineManager].Initial_105706__TO__Off_61718影响

07 shallowManager[ShallowTurbineManager].StateMachine_Turbine_03_Off ENTRY

08 shallowManager[ShallowTurbineManager].StateMachine_Turbine_03_Off DO

触发器[模式]

信息省略...

注记：由于*deepManager*和 shallowManager 的 trace 完全一样，所以*deepManager*的*trace*是从这个序列中过滤掉的。

我们可以了解到：

- 由于没有默认历史转移
  为*History3*定义，执行状态的标准默认输入；在区域所包含的区域中发现了一个 Initial 节点，因此转移源自*Initial*已启用（参见第 06 行）

那么活动状态配置包括：

- 涡轮_03
- Turbine_03.On
- Turbine_03.On.Low

这适用于*deepManager*和*shallowManager*。

国家的历史条目

作为参考，我们展示了每个涡轮机在第一次激活后的深度历史快照：

涡轮_01

- Turbine_01.On
- Turbine_01.On.High

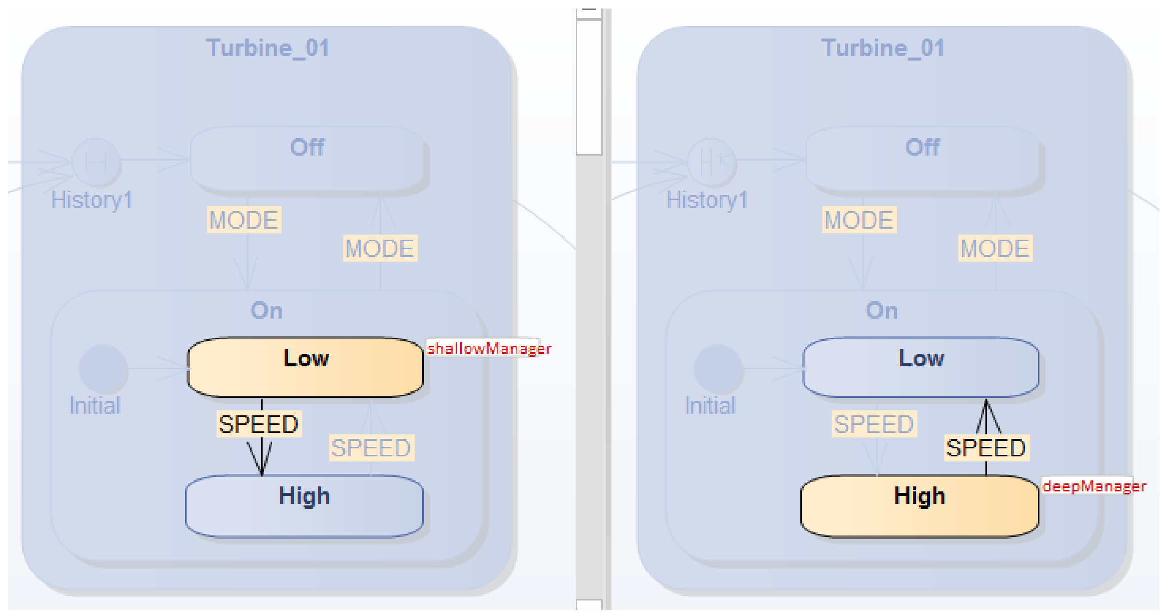
涡轮_02

- Turbine_02.Off

涡轮_03

- Turbine_03.On
- Turbine_03.On.Low

当我们进一步简单地 NEXT 时，触发器将再次被激活。



从仿真窗口可以观察到这个序列：

对于浅层管理器：

触发器[下一个]

01 shallowManager[ShallowTurbineManager].StateMachine_Turbine_03_On_Low 退出

02 shallowManager[ShallowTurbineManager].StateMachine_Turbine_03_On EXIT

03 shallowManager[ShallowTurbineManager].StateMachine_Turbine_03 退出

04 shallowManager[ShallowTurbineManager].Turbine_03__TO__History1_105711_61732影响

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影响

10 shallowManager[ShallowTurbineManager].StateMachine_Turbine_01_On_Low ENTRY

11 shallowManager[ShallowTurbineManager].StateMachine_Turbine_01_On_Low DO

我们可以了解到：

- shallowHistory 节点将 Turbine_01 恢复到 Turbine_01.On
- 那么复合状态区域所包含的区域将被*Initial*节点激活，在*Low*激活

对于 deepManager：

触发器[下一个]

01 deepManager[DeepTurbineManager].StateMachine_Turbine_03_On_Low 退出

02 deepManager[DeepTurbineManager].StateMachine_Turbine_03_On EXIT

03 deepManager[DeepTurbineManager].StateMachine_Turbine_03 退出

04 deepManager[DeepTurbineManager].Turbine_03__TO__History1_105679_61708影响

05 deepManager[DeepTurbineManager].StateMachine_Turbine_01 条目

06 deepManager[DeepTurbineManager].StateMachine_Turbine_01 DO

07 deepManager[DeepTurbineManager].StateMachine_Turbine_01_On ENTRY

08 deepManager[DeepTurbineManager].StateMachine_Turbine_01_On_High ENTRY

我们可以了解到：

- *deepHistory*节点将 Turbine_01 恢复到 Turbine_01.On.High

触发器[NEXT] 退出 Turbine_01 并激活 Turbine_02

*shallowManager*和*deepManager*都激活了 Turbine_02.Off，这是它们退出时的 History 快照。

触发器[NEXT]退出Turbine_02并激活Turbine_03

*shallowManager*和*deepManager*都激活 Turbine_03.On.Low。但是， *shallowManager*和*deepManager*的顺序是不同的。

对于*shallowManager* ， *shallowHistory*只能恢复到 Turbine_03.On。由于在 Turbine_03.On 中定义了*Initial*节点，因此转移

源自*Initial*的将被启用并达到 Turbine_03.On.Low。

01 shallowManager[ShallowTurbineManager].StateMachine_Turbine_02_Off 退出

02 shallowManager[ShallowTurbineManager].StateMachine_Turbine_02 退出

03 shallowManager[ShallowTurbineManager].Turbine_02__TO__History3_105713_61725影响

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影响

09 shallowManager[ShallowTurbineManager].StateMachine_Turbine_03_On_Low ENTRY

10 shallowManager[ShallowTurbineManager].StateMachine_Turbine_03_On_Low DO

对于*deepManager* ， *deephistory*可以直接恢复到 Turbine_03.On.Low。

01 deepManager[DeepTurbineManager].StateMachine_Turbine_02_Off 退出

02 deepManager[DeepTurbineManager].StateMachine_Turbine_02 退出

03 deepManager[DeepTurbineManager].Turbine_02__TO__History3_105680_61701影响

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

事件宏：EVENT_PARAMETER

EVENT_PARAMETER 是一个函数宏，用于访问信号实例的属性，在状态的行为中，转移的守护和效果。该宏将根据仿真语言扩展为可执行代码。

访问默认宏定义

功能区|开发 |源代码 |选项 |编辑代码模板|语言 | Stm事件参数

用途格式

%EVENT_PARAMETER（信号类型，信号属性名称）%

例如： A信号'MySignal'有两个属性，'foo: int '和'bar: int '；这些用例是有效的：

- 转移
  的效果：%EVENT_PARAMETER(MySignal, foo)%
- 状态的行为：%EVENT_PARAMETER(MySignal, bar)%
- 转移
  的守卫：%EVENT_PARAMETER(MySignal, bar)% > 10
- 状态的行为：%EVENT_PARAMETER(MySignal, bar)%++
- 跟踪值到仿真窗口：%TRACE(EVENT_PARAMETER(MySignal, 富))%

宏扩展示例

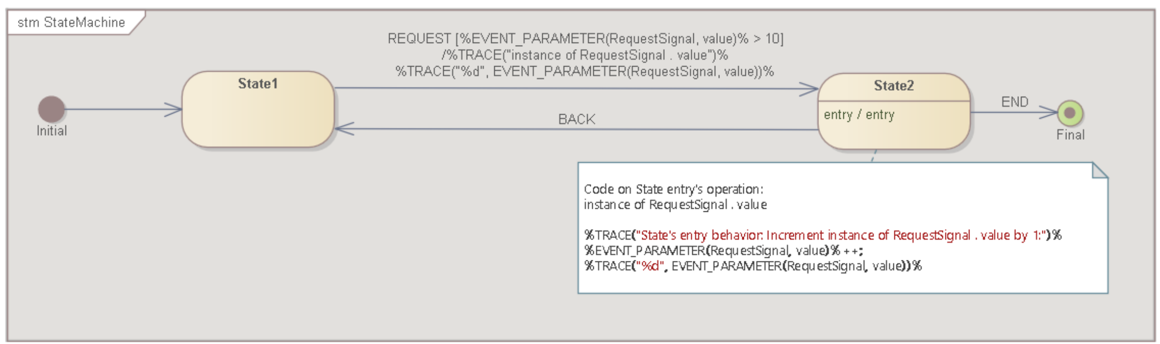
对于带有属性 "value"的信号 "MySignal"，转移
的宏扩展示例转移
带信号触发：%EVENT_PARAMETER(MySignal, value)%

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

示例

这个例子演示了如何在转移

中使用 EVENT_MACRO转移
的效果、守卫和状态的行为。



在运行模拟时，
(1) 触发 REQUEST 并为属性值指定数字1 。
由于守卫的条件为false ，因此活动状态将保持为 State1。
(2) 触发 REQUEST 并为属性值指定数字 11。
由于守卫的条件为真，活动状态将从State1变为State2；
转移
的效果被执行。在这里，我们将信号属性的运行时值跟踪到模拟窗口。
State2 的行为被执行。在这里，我们增加了信号属性的运行时间值并将其跟踪到模拟窗口。

[32608107] [Part1:TransactionServer]转移
影响：影响
[32608118] [Part1:TransactionServer] 入口行为：StateMachine_State1
[32608124] [Part1:TransactionServer]行为：StateMachine_State1
[32608877] [第 1 部分：TransactionServer] 完成：TransactionServer_StateMachine_State1
[触发器] 等待简单
[32613165] 命令：广播 **REQUEST.RequestSignal(1)**
[32613214] [Part1：TransactionServer]事件排队：REQUEST.RequestSignal（值： 1 ）
[32613242] [Part1:TransactionServer]事件调度：REQUEST.RequestSignal（值： 1 ）
[触发器] 等待简单
[32619541] 命令：广播 **REQUEST.RequestSignal(11)**
[32619546] [Part1：TransactionServer]事件排队：REQUEST.RequestSignal（值：11）
[32619551] [Part1：TransactionServer]事件调度：REQUEST.RequestSignal（值：11）
[32619557] [Part1:TransactionServer] 退出行为：StateMachine_State1
[32619562] [Part1:TransactionServer]转移
影响：影响
[32619567] **RequestSignal** 的实例。价值
[32619571] **11**
[32619576] [Part1:TransactionServer] 入口行为：StateMachine_State2
[32619584]状态的进入行为：增加 **RequestSignal** 的实例。**值1** ：
[32619590] **12**
[32619594] [Part1:TransactionServer]行为：StateMachine_State2

[32620168] [第 1 部分：TransactionServer] 完成：TransactionServer_StateMachine_State2

[触发器] 等待简单

[32622266] 命令：广播结束

[32622272] [Part1：TransactionServer]事件排队：END

[32622310] [Part1：TransactionServer]事件调度：END

[32622349] [Part1:TransactionServer] 退出行为：StateMachine_State2

[32622359] [Part1:TransactionServer]转移

影响：影响

[32622896] [第 1 部分：TransactionServer] 完成：TransactionServer_VIRTUAL_SUBMACHINESTATE

限制和解决方法

由于宏扩展涉及类型转换，因此用户有责任确保类型转换有效。

以下是模型中类型转换会遇到问题的一些常见情况的解决方法。

- 一个转移
  具有不同信号类型的多个触发器

例如，一个转移

有triggerA（指定SignalA）和triggerB（指定SignalB）； A这个转移

时，宏 %EVENT_PARAMETER(SignalA, attributeOfA)% 将不起作用转移

由 triggerB 触发。

我们建议创建两个转换，一个使用 triggerA（指定 SignalA），另一个使用 triggerB（SignalB）。

- 一个状态是不同信号触发的多个**Transition**的目标

例如，TransitionA 和 TransitionB 都针对 MyState，TransitionA 由 SignalA 触发，TransitionB 由 SignalB 触发。

如果在状态的行为代码中使用了 EVENT_PARAMETER，则一个宏将无法同时适用于两种情况。

我们建议将处理信号属性的逻辑从状态的行为转移到转移

的效果。

- 自定义模板和生成的代码

用户也可以更改默认模板，在类型转换前添加运行时间类型标识（RTTI）。用户也可以在生成后修改生成的代码，也可以满足编译后的模拟目的。