Statechart to C++ Mapping

Introduction

Goal
This Technology Accelerator™ illustrates the use of Codagen Architect to map UML statechart diagrams into ANSI C++ source code.

Compatibility
This Codagen Technology Accelerator™ requires Codagen Architect version 3.0, Service Release 1.

Overview
Reactive systems are best modeled with statecharts diagrams since the main focus of these diagrams is on the possible events that the system can respond to, and how the response is carried out. With these diagrams, we can visualize the behavior that the system will exhibit in response to a particular event according to its current state.

An interesting challenge is the transformation of these statecharts into manageable source code. Several patterns for mapping these statecharts into actual code have been published. In this Technology Accelerator™, we will use the “State” design pattern (from Gamma et al.) to derive ANSI C++ code from the model’s class and state diagrams. The basic principle behind this pattern is that an object’s behavior will change according to its current state, giving the appearance that its class has changed.

For each class possessing state diagrams (hereafter called a state-based class), the following subsystem is produced:
- a “context class” representing the subsystem’s entry point, delegating all requests to the current “state class”
- an abstract “state interface” that defines the methods that each “state class” must implement
- a “state class” for each possible state that the state-based class can be in, encapsulating a specific behavior of this state-based class

To illustrate the mapping from statechart diagrams to ANSI C++ code, this Technology Accelerator™ features a temperature control system that is reactive to the following external events: user commands (from a control panel, for example) and changes in the ambient temperature. This system is modeled as a collection of classes containing state diagrams to illustrate their state dependent behavior.
Application Model

This section takes a black-box approach: it describes the project’s expected input and the produced output (the focus is on the “what”). In other words, it describes the nature of the platform-independent model (PIM) and the platform-dependent model (PSM).
Any model that possesses the following characteristics can be used as an input for the Technology Accelerator™ Architect project:

- Each association must have a role name for each navigable end.
- This role name must be used as the target name in the send events.
- Each send event to a specific target class must correspond to a method in the target class interface.
- Each trigger on a transition in a class statechart must correspond to a method in the class interface.
- Each state in a class statechart must have a transition for each possible trigger (all methods in the class interface).

### Architecture Specification

The following table lists the properties and the associated UML model elements, defined in the “HeatingSystem.csf” architecture specification file.

<table>
<thead>
<tr>
<th>Layer/Issue/Property</th>
<th>UML Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design Pattern</td>
<td></td>
<td>Generic Design Patterns</td>
</tr>
<tr>
<td>State</td>
<td></td>
<td>Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.</td>
</tr>
<tr>
<td>isStateBased</td>
<td>Class</td>
<td>Defines which class will have its behavior implemented by multiple state classes, one for each state that the class may be in.</td>
</tr>
</tbody>
</table>
Here are the main characteristics of the PSM produced from the given PIM:

For each class in the PIM (both state-based and non-state based):
- Create a new class with the same name as the PIM class, containing the attributes and operations of the PIM class. Also add an attribute for each of the PIM class’s outgoing associations.

For each state-based class in the PIM (classes with statecharts):
- Create the pattern’s State Interface ("ModelClass_StateInterface"). It contains the same attributes and operations as the PIM state-based class. The attributes are declared as static so that the context of the state-based class is shared among all of its state classes (described below). The operations are declared as virtual so that each state class can override it to implement its own specific behavior.
- Add a reference to the State Interface in the created state-based class, and for each of the operations in the state-based class interface, forward the request to the appropriate method in the State Interface class.

For each state in a PIM state-based class:
- Create a “state class” that inherits from the State Interface and add a reference to it in the state-based class.
- For each triggered outgoing transition, add a method to the state class (these methods should match the State Interface methods), along with support for the guard, action and send event of the transition.
- Create a control loop method to handle the logic (guards, actions and send events) for all un-triggered transitions.
- Finally, create a constructor in the state-based class that will instantiate all state classes and the logic required to manage the transition between them.

Code Generation Process
This section takes a white-box approach: it describes the project templates (the focus is on the “how”).

Code Generation Templates
This table contains the important templates for creating the ANSI C++ PSM from the given PIM:
<table>
<thead>
<tr>
<th><strong>Template Name</strong></th>
<th><strong>Type</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Create Business Class</td>
<td>Composite</td>
<td>This composite template creates a C++ class for every classifier in the model. The created class will contain the model class attributes and methods. It will also contain a reference attribute for each association target (for the model class outgoing associations).</td>
</tr>
<tr>
<td>Create State Enumeration Class</td>
<td>Class</td>
<td>For every state-based classifier, create an enumeration that contains an enumerator for each possible state in the classifier's state machines.</td>
</tr>
<tr>
<td>Create State Interface for state-based class</td>
<td>Composite</td>
<td>This composite template will create a State Interface for each state-based class containing the signatures of every method plus a &quot;control loop&quot; method.</td>
</tr>
<tr>
<td>Create Classes from States</td>
<td>Composite</td>
<td>For each state in a classifier's statecharts, create a class that inherits the State Interface.</td>
</tr>
<tr>
<td>Create State reference</td>
<td>Composite</td>
<td>Create references to the state classes within the state-based class.</td>
</tr>
<tr>
<td>Create State Table</td>
<td>Composite</td>
<td>Create a state table in the context classes large enough to hold the references to the state classes. This table will be used for state transitions.</td>
</tr>
<tr>
<td>State-Based Class Constructor</td>
<td>Method</td>
<td>This constructor will instantiate all possible state classes and inserts them into the state transition table.</td>
</tr>
<tr>
<td>Create Attributes Getters/Setters in State-Based Class</td>
<td>Composite</td>
<td>All getter/setters in the state-based class are forwarded to the current state.</td>
</tr>
<tr>
<td>Create method from state transition</td>
<td>Method</td>
<td>Called by &quot;Create Classes from States&quot;. Create a method from the trigger on a state's outgoing transition.</td>
</tr>
</tbody>
</table>

**Example Model**

The temperature control system in this example is composed of four entities: the thermostat, the furnace relay, the air conditioner relay and a thermometer.

Of these entities, it is the thermostat that is of particular interest. It is this entity that is the central "controller" of the system. It constantly monitors the current temperature (from the thermometer) and activates/deactivates the furnace and air conditioner relays in order to bring the ambient temperature up/down to the target temperature. The thermostat also listens for user commands which permit to start and stop the whole temperature control system and to change the target temperature.

Note: the system's air conditioner and furnace are directly linked to the thermometer since we choose not to model the ambient air, which is normally found between them.

**Generating the Example**

This section describes the additional steps required to generate the PSM model, compile the resulting code, and execute the example. To assist in these steps, the example provides a Rose model of the temperature control system and a “main” C++ file (TemperatureControlSystem.cpp) that mimics a control panel for controlling the temperature control system.

**Files provided**

Here is a list of files included in this Technology Accelerator™:
- Temperature Control System.pdf – this document
Technology Accelerator™: Statechart to C++ Mapping

- TemperatureControl.mdl – the Rose model for the temperature control system
- StatePattern.mdl – the Rose model that explains the project templates
- HeatingSystem.csf – Codagen Architect architecture specification file
- TemperatureControlSystem.gpcpp – Architect project templates
- src\TemperatureControlSystem.cpp – Main function of the test application

PSM generation
1. Open the provided Rose model: TemperatureControl.mdl
2. From the TemperatureControlSystem package in the model, invoke Codagen Architect – Implement
3. Generate code using every template (Generate All command)

Code to add manually
Add the following two lines of code (these lines are required for the heating/cooling elements to affect the “ambient air”):
1. In the code pocket of the “controlLoop” method in the file TemperatureControlSystem_AirConditionerRelay_State_On.cpp, add the following:
   ```cpp
   theThermometer->currentTemp--; 
   ```
2. In the code pocket of the “controlLoop” method in the file TemperatureControlSystem_FurnaceRelay_State_On.cpp, add the following:
   ```cpp
   theThermometer->currentTemp++; 
   ```

Code compilation
1. Using your favorite C++ compiler, create a project containing all the generated source code along with the provided “main” C++ file (TemperatureControlSystem.cpp).
2. Compile your new project

Example testing
The Temperature Control System is tested using a prompt-based application (the “main” function is provided in “TemperatureControlSystem.cpp”). This application displays the current and target temperatures, along with a short menu for controlling the system. With this menu, you can change the current and target temperatures, turn the system on or off and activate a cycle of the control loops. Each control loop cycle will affect the temperature by one degree, if required.

Copyright and Trademark Information
The software described in this document is furnished under a license agreement or non-disclosure agreement. The software may be used or copied only in accordance with the terms of those agreements. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means electronic or mechanical, including photocopying and recording for any purpose other than the purchaser’s personal use, without the prior written permission of Codagen Technologies Corp.

Codagen Technologies Corp.
2075 University St., Suite 1020
Montreal (Quebec)
Canada H3A 2L1

Codagen® and Generation Template® are registered trademarks of Codagen Technologies Corp. The Codagen logo and design and the terms White Box, Code Pocket, and Technology Accelerator are service marks or trademarks (™) of Codagen Technologies Corp.

ANSI is a registered trademark of the American National Standards Institute.