Running the Bookstore Models in BridgePoint

These examples will show you how to use the BridgePoint Model Verifier to run Executable UML models through various scenarios. Once you have run these examples, you should be able to try other scenarios and to add more features to the models.

We will use example scenarios from Chapters 10 and 15 of the book *Executable UML: A Foundation for Model-Driven Architecture*. These are based upon a simple model set consisting of only the Product, Order, ProductSelection, CreditCardCharge, and Shipment classes. These simple bookstore models are downloadable from www.executableumlbook.com/models/SimpleBookstore.

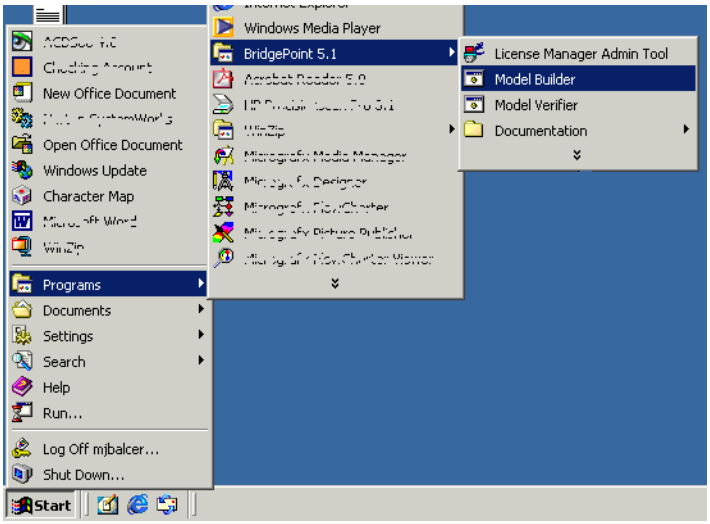
To run these models, first get a copy of BridgePoint from Project Technology.

After installing BridgePoint, you are ready to load and run the models.

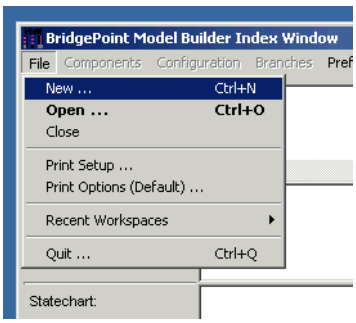
Load the models into a BridgePoint repository

This step loads the BridgePoint repository (the working database of the domain models) from the file you downloaded.

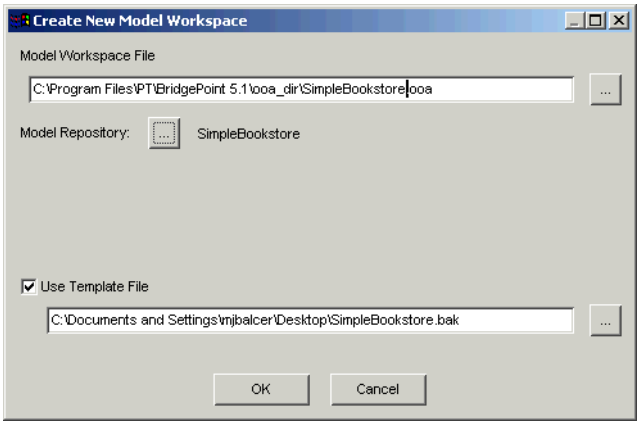
1. Start the Model Builder



2. Create a new model workspace.



3. Populate the workspace by importing the SimpleBookstore.bak file.



4. Browse the models

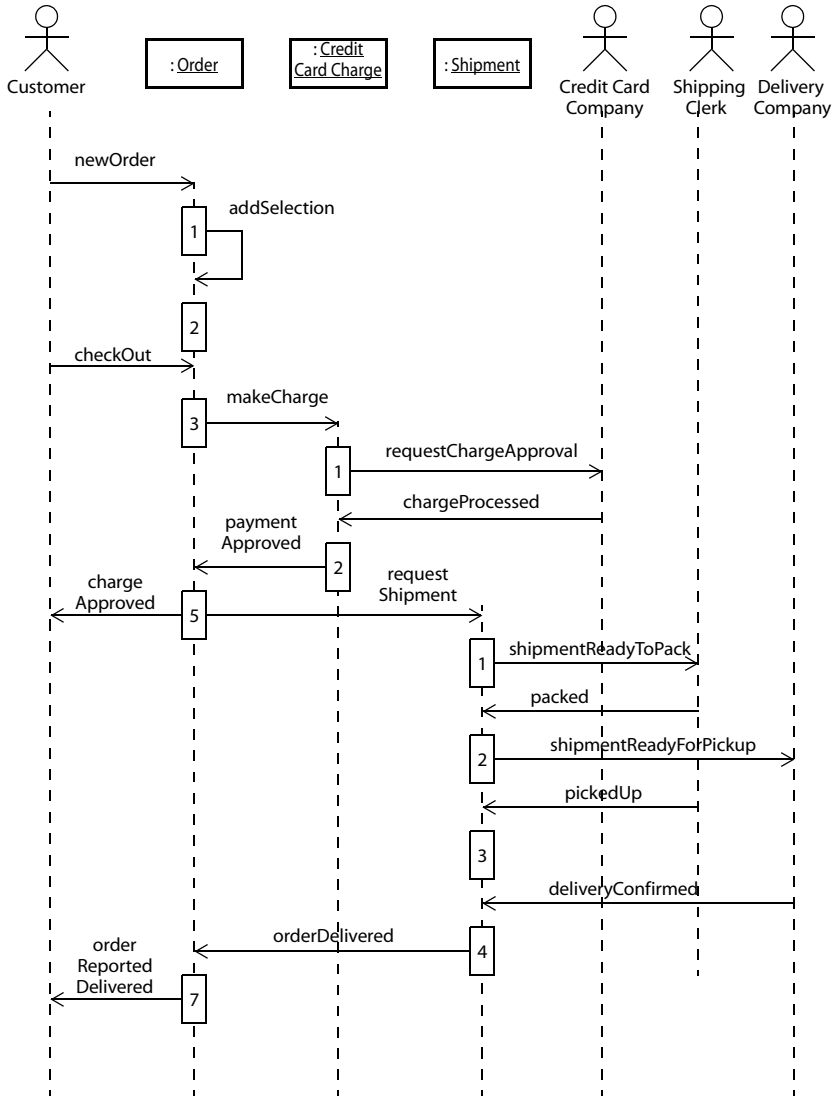
5. Print the model diagrams for reference while running the models

Some of the action language and the diagrams may differ from the model diagrams in the book. See the document “Using BridgePoint 5 for Executable UML” to understand the differences between the models in the book and the models in BridgePoint.

Single-Item Ordering Scenario

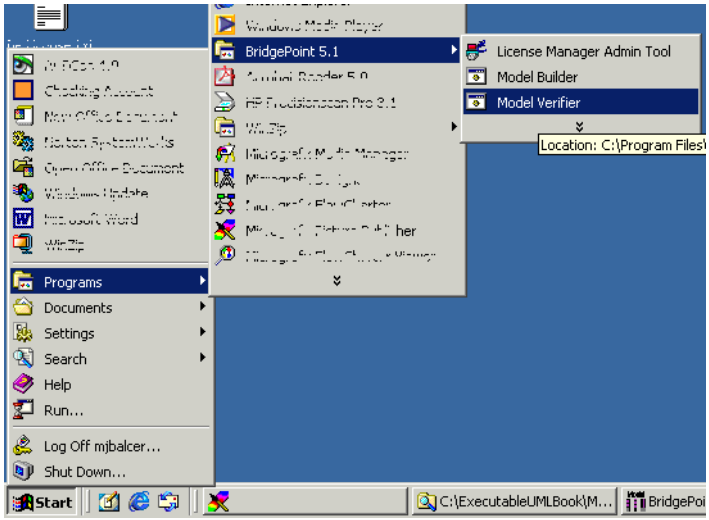
In this scenario, you will create an order with a single product selection, check out the order, approve the credit card charge, and prepare the shipment. You will learn how to create a simulation workspace, to create pre-existing instances, to send signals into the domain, and to step through state transitions.

The scenario is illustrated by the sequence diagram in Figure 10.9 of *Executable UML*, reprinted below for your convenience.

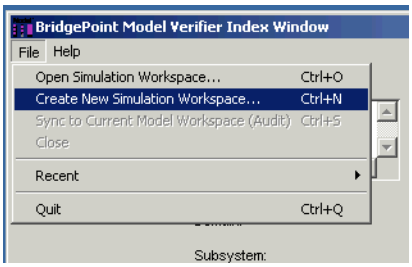


Start a simulation workspace

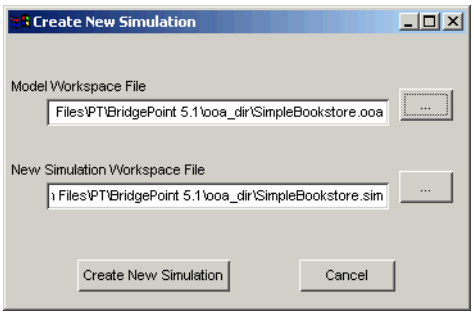
1. Start the Model Verifier



2. Create a new simulation workspace.

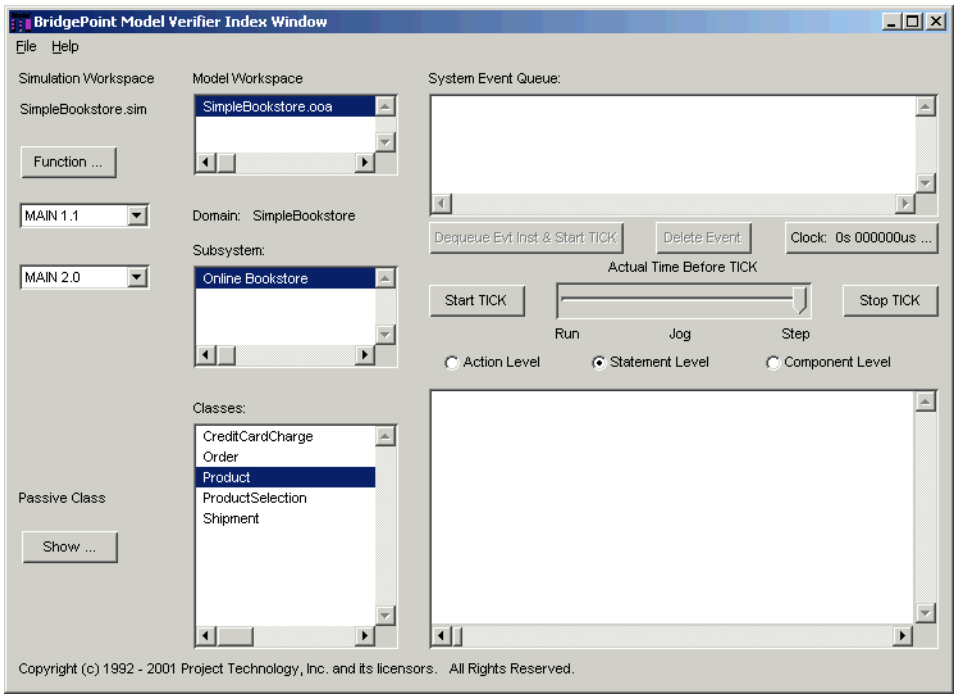


3. Base it upon the SimpleBookstore model repository.

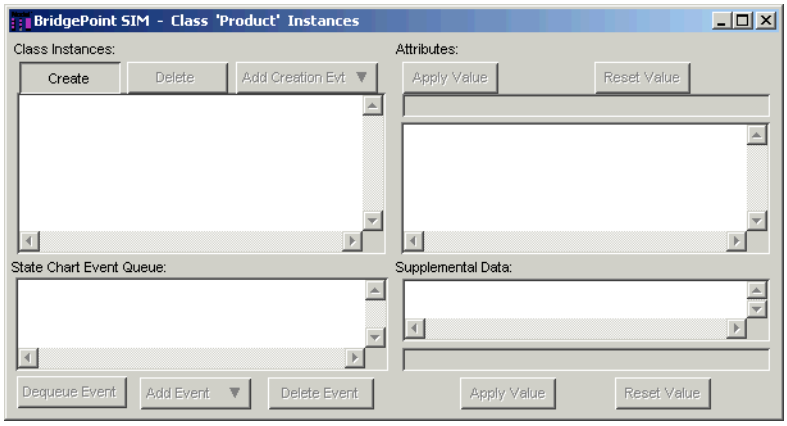


Create some instances for the test

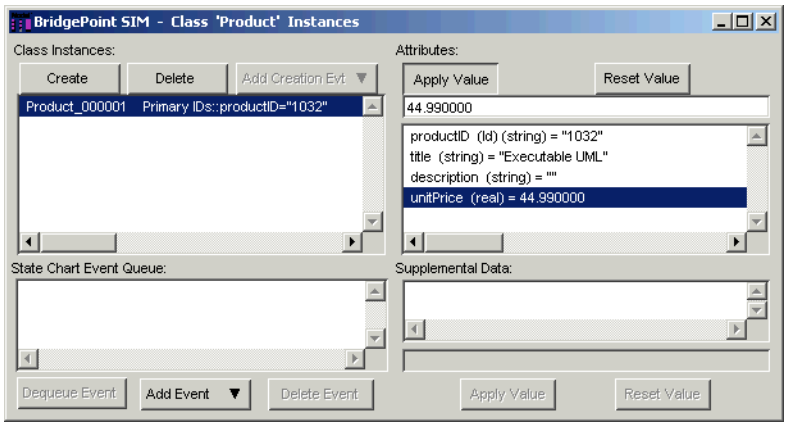
1. On the main simulator window, double-click the “Product” class.



2. This opens up the Product instance editor. Create a new Product object by pressing “Create.”



3. Set the attribute values by selecting the attribute, entering the value in the text box, and pressing "Apply Value."



4. Create additional instances by pressing Create and entering their attribute values.

5. Check the instances you have created by clicking the instance and checking the values of the attrrib-utes.

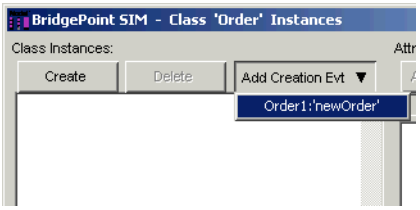
6. Close the Product window.

Start the scenario with an initiating event

The single-item scenario in Figure 10.9 begins when the customer (that's you!) sends a newOrder signal to create a new Order object and transition that state machine instance into its initial state.

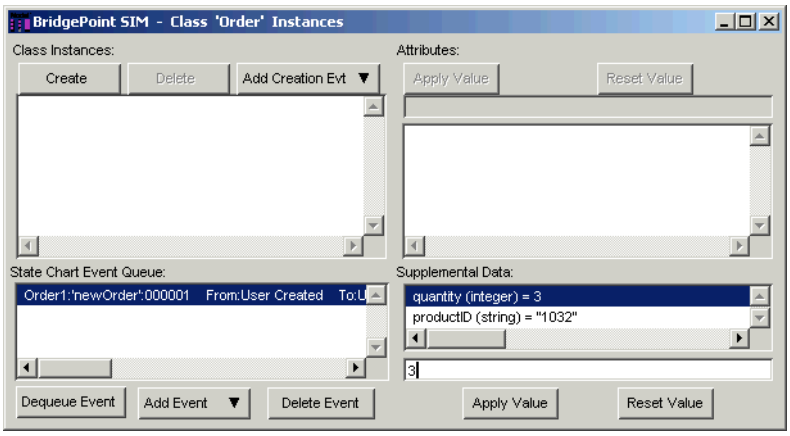
1. From the main simulator window, double-click the Order to open up the Order instance editor.

2. Press the "Add Creation Evt" button to select and send a new creation signal (a signal that causes a transition from the initial pseudostate)[1].



1. BridgePoint treats the creation of the instance in its initial pseudostate and its initial event as the same activity.

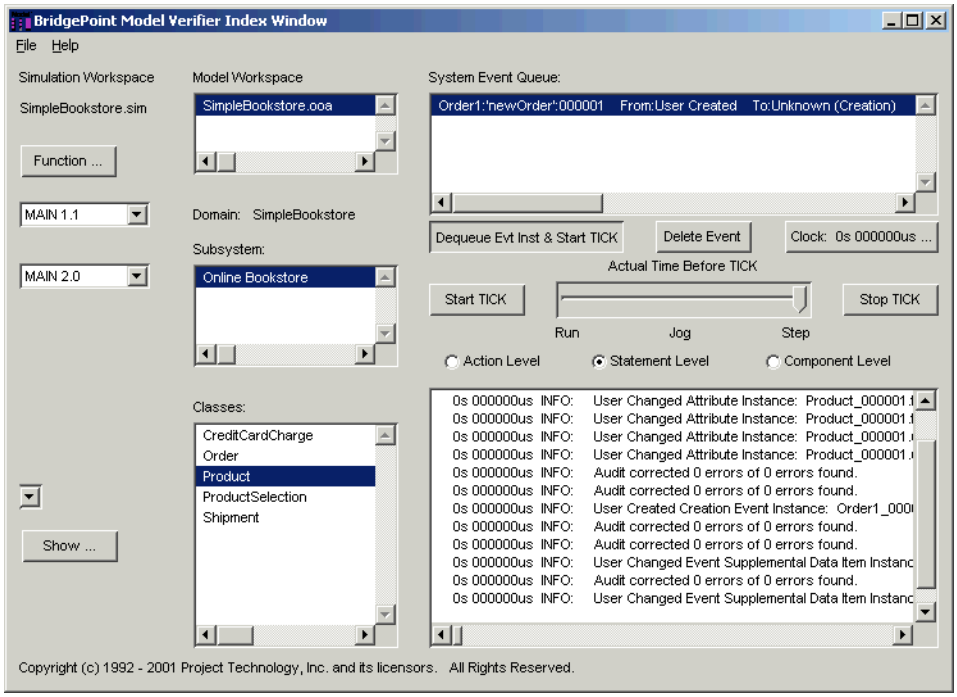
3. Select the event in the State Chart Event Queue box and edit each of the event parameters just as you edited attribute values in the last step. Click on the parameter, enter the value in the box, and press Apply Value. For this test, use the productID of a product you just created and a quantity of 3.



4. Note that the new signal appears in the pending event list on the main simulator window. You may want to position the main simulator window and the Order instance editor so that you can switch between them and, if possible, that you can see both at the same time.

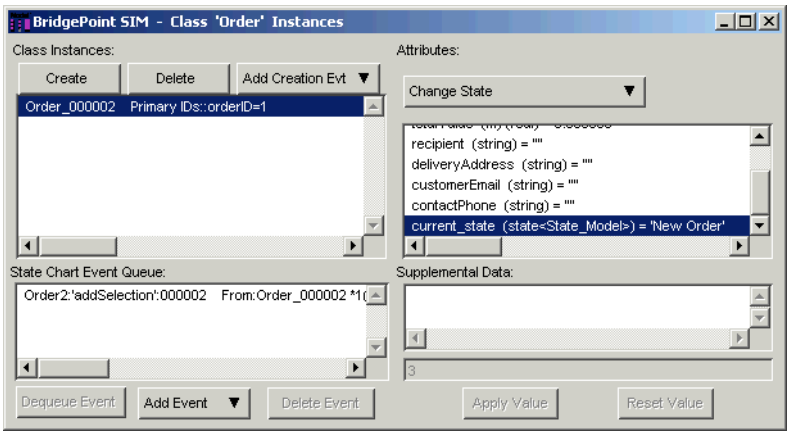
Process the first event

1. On the main simulator window, select the newOrder event from the System Event Queue. Then press "Dequeue event and start tick." This sends the newOrder signal to create the order.



2. The event log shows the actions of the initial state of the Order as they are executed. One of those actions is to signal addSelection to self. Note how that event now appears in the pending event list.

3. Also note how the Order instance editor window shows the newly-created Order in state “New Order.”



4. Each time a state procedure is executed, the Model Verifier audits the model to check for situations where the instances do not conform to the rules established by the models. In this case, the log notes that an Order exists with no ProductSelection. Do not worry about this, as the next events to be processed will create the ProductSelection. (See Rule 8 on page 191 of *Executable UML*.)

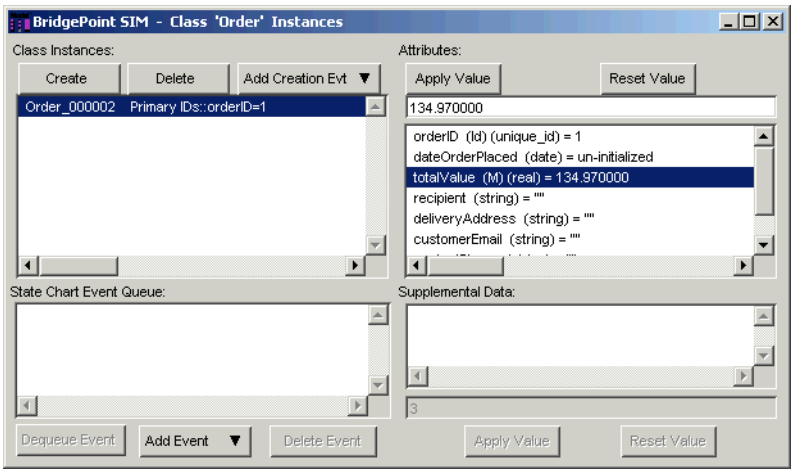
Process subsequent events to complete the initial selection

1. On the main simulator window, select the addSelection event and press “Dequeue...”

2. The Order should transition to state “Adding Selection to Order” and execute its state procedure. Check the Order’s state on the Order instance window.

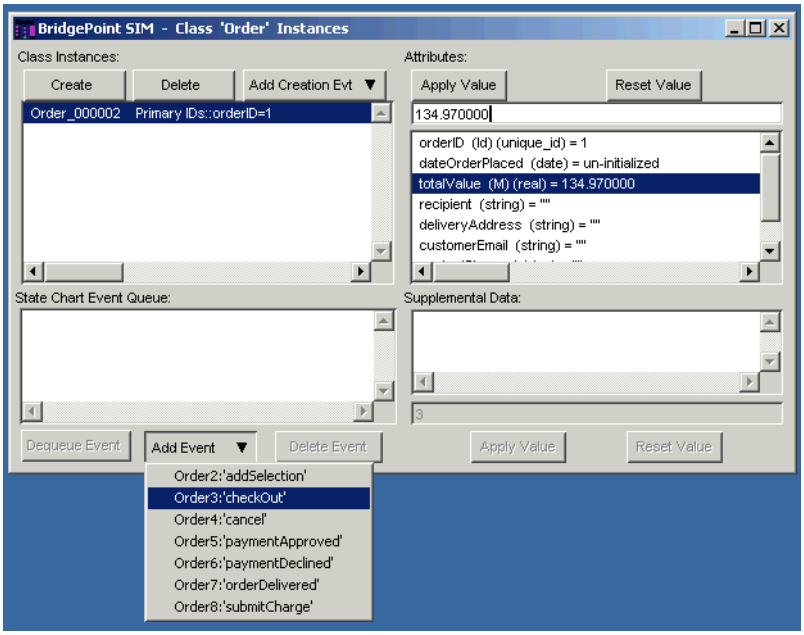
3. The Order also created a new instance of ProductSelection. Check this by opening the ProductSelection instance editor (double-click the ProductSelection on the main simulator window).

4. Finally, note that the derived attributes ProductSelection.selectionValue and Order.totalValue are correctly computed (3 times the unit price of the Product selected).



Check out the order

1. To check out the order, send the order a checkOut signal. On the Order instance editor, select the Order and press the Add Event button. (Note: checkOut is not a creation signal; don't press the "Add Creation Event" or you'll get the wrong event.)



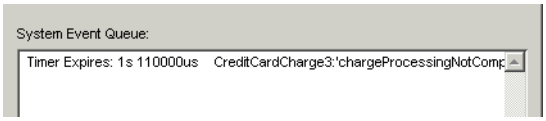
2. Select the checkOut event. and edit the checkOut parameter values. (Note that since the models don't "do" anything with those values, you can leave them blank and get on with verification fun!)

3. Return to the main simulator window and dequeue the checkOut event. This will transition the Order to state 3: Establishing Customer and Verifying Payment. The state procedure sends a makeCharge signal to create a CreditCardCharge.

4. Dequeue the makeCharge event.

Simulate the charge processing

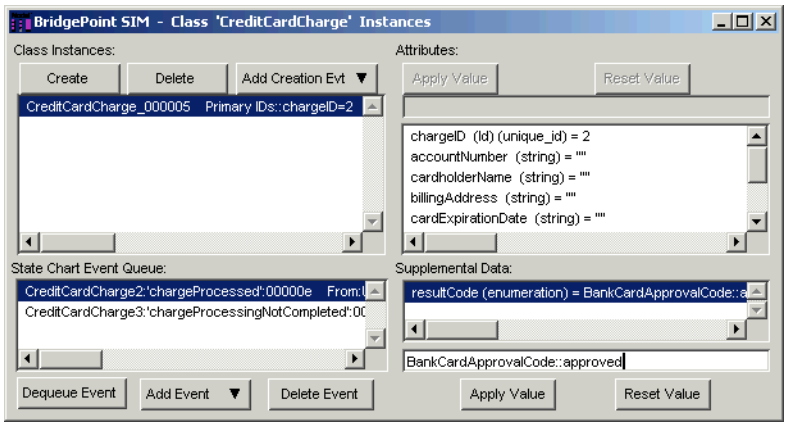
1. Note that the pending event list now contains the delayed event chargeProcessingNotCompleted. In a real system, this is intended to cause the charge to be declined if the credit card company does not respond within 60 seconds (60,000 microseconds)[1].



2. To signal the CreditCardCharge that a credit processing was completed, open the CreditCardCharge instance editor, select the charge instance, and add a chargeProcessed signal.

1. When stepping through models in the Verifier, you cannot get the delayed signal to be processed by simply waiting 60 seconds. You must specifically dequeue the event.

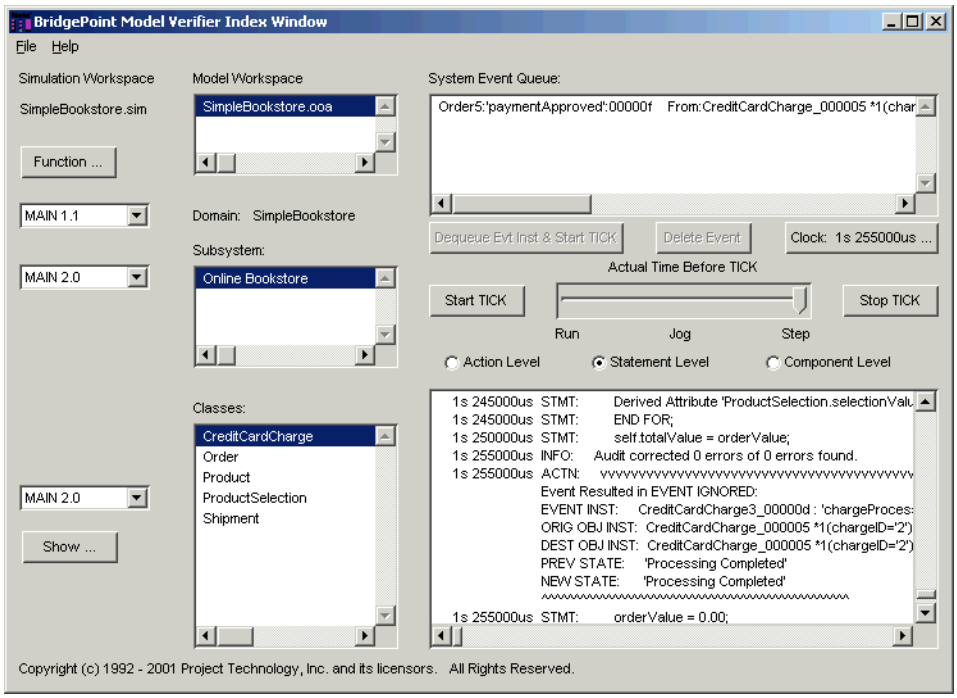
3. For the charge to be approved, the resultCode parameter value must be set to the enumeration value BankCardApprovalCode::approved. Set that value (the type name is important).



4. Return to the main simulator window. Since (in our simulation) the credit card company responded in less than 60 seconds—before the timeout—we want to process the chargeProcessed signal instead of the chargeProcessingNotCompleted signal. Select the chargeProcessed signal and press “Dequeue event and start tick.”

5. The credit card charge transitions to state “Processing Completed” and signals “paymentApproved” to the Order.

6. Dequeue the chargeProcessingNotCompleted signal (remember, it’s still there—it doesn’t disappear by magic!). Since the state transition table entry for (state: Processing Completed, event chargeProcessingNotCompleted) is “Event Ignored,” the event is dequeued without causing any transition or executing any state procedure. (The Event Ignored is, fortunately, recorded on the simulation log.)



Finish the Order by shipping it

1. Dequeue the paymentApproved event. The Order transitions to “Being Packed and Shipped.” Its state procedure signals requestShipment to create the Shipment.

2. Dequeue requestShipment. Open the Shipment instance window and verify that the Shipment was indeed created and is now in state “Preparing Shipment.”

3. Simulate the shipping clerk and shipping company packing, picking up, and confirming delivery of the shipment. Create the signals to the Shipment and then process (dequeue) them.

4. The final signal, deliveryConfirmed, transitions the Shipment into the Delivered state. The state procedure signals orderDelivered to the Order. Process that event, and the Order has been delivered!

Additional things you might want to try

There are many variations on this scenario that you can try with the Simple Bookstore models:

1. Add a second item to the order. This is the multiple-item ordering scenario of FIgure 15.10.

2. Add additional items of the same product to the order. Note that additional instances of ProductSelection are not created, but rather the existing instance’s quantity attribute is incremented.

3. Have the credit card company decline the charge.

4. Dequeue the chargeProcessingNotCompleted delayed signal to simulate the credit card company failing to respond in turn.

5. Have multiple orders being processed simultaneously.

6. Try adding an item to an already-checked-out Order (or other combinations that result in Can’t Happen transitions).

Additional things that will identify errors in the models

The Simple Bookstore isn’t perfect, though. As with any early increment, it is subject to many simplifying assumptions. Break any of these, and the models are likely to fail.

1. Add a selection with a product ID for which no Product instance exists.

2. Add a selection with a negative quantity.

3. Try to cancel an Order that has been checked out but not yet shipped.

4. Have the Shipment be picked up before it is packed. (Or another way, what happens if a shipment is picked up but the shipping clerk forgot to record it as “packed?”)

5. Have the delivery company return a shipment as undeliverable.

6. Have the customer return an Order

We leave correcting these problems (or adding these additional features) as exercises to the reader. From time to time we will post solutions on the website, www.executableumlbook.com.