Putting Physhun To Work

Introduction

Many IT architectures rely on multiple systems to successfully complete complex transactions. These systems must coordinate and synchronize their activities in order to smoothly deliver the desired collective function. As the activities to be coordinated increase in complexity, the need to monitor each system rises. Inter-system transaction monitoring gives insight into the status of each atomic transaction, the complex process to which the atomic transactions contribute, and overall system performance. The increased contribution of the monitoring and oversight process to the total computational load means that business performance can be increased by increasing the competence of the monitoring solution. When the monitoring solution can quickly take actions when transactions fail, notify support personnel when manual intervention is required, and forecast impending system operational issues, overall system performance is augmented.

Monitoring simple transaction processes is easy, but monitoring complex processes is difficult. When the complex process spans multiple Commercial-Off-The-Shelf (COTS) systems that provide little support for interoperability, the monitoring tasks becomes very, very difficult. This paper describes how the XML Transaction Monitor (XTM) project faced challenges related to inter-system transaction monitoring, and how the Physhun framework proved an integral component in the overall solution.

Our company, Wazee Group, is a proud sponsor of the Physhun project. A few years ago, we noticed a void in the open source world with regard to finite state machine (FSM) technology. Members of our staff began using FSM alternatives over twenty years ago and felt that it was a misunderstood and underutilized technology. Combining our expertise in FSMs with desire to give back to the open source community, our colleague, Justin McCarter, launched the Physhun project. While the XTM was the first Physhun project to be deployed into a 24/7 environment, we believe that the framework could be useful in many other projects and problem domains.

Background

The IT architecture for order entry and fulfillment consists of four major systems. The External Client was a system owned and operated by a business trading partner. The partner wished to submit and manage orders by sending information via XML-based requests and receiving status updates via XML-based notifications.

Internal to the enterprise are two business-oriented systems. The Order Entry (OE) system is a COTS package and allows human users to enter and manage orders. Once the orders are entered and are validated through various business processes, the order is sent to the Order Fulfillment (OF) system for completion. The Order Fulfillment system is another COTS package. As the OF completes its internal processing, notifications are sent back to the Order Entry system. Likewise, once the Order Entry system has completed the order, a notification is sent to the External Client. A high level diagram is shown below.

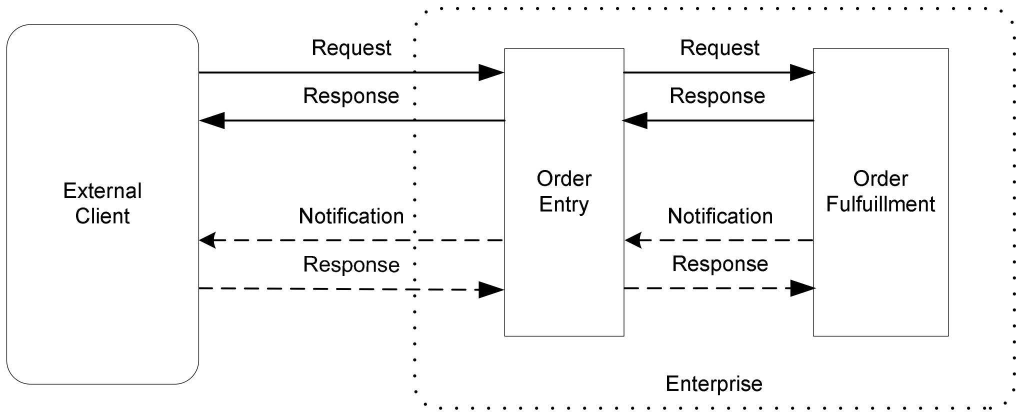


Figure 1- High Level Logical Architecture

The third internal component is the XML Transaction Router. This system routes the XML requests and notifications to the appropriate system. The Router consists of a commercial application with a custom configuration.

The fundamental high level requirements for the monitoring solution consist of:

1. Record the requests and notifications.
2. Send alert when a transaction times out.
3. Send alert when processing of transactions slows down.
4. Allow users to review failed transactions.
5. Allow users to resubmit failed requests and notifications.

Transaction Process Modeling

We began by analyzing the transaction model with respect to the application requirements, in order to determine the complexity of the processes to be modeled. While the high level architecture is straightforward, the transaction process models are non-trivial. An initial analysis of the handling of a new order is given below. This is the “happy path” representation and does not include all of the failure points.

1. A new order request is received by the Router from the External Client
2. A response is generated as part of the request/reply protocol
3. The Router determines the appropriate internal system for the request. In this case it is the Order Management (OM) system.

4. The Router forwards the request to the OM system and receives a reply from the OM system.
5. The OM system creates an internal order which begins processing, including both automated and manual tasks.
6. At the appropriate point in its internal processing, the OM system issues an XML request to the Order Fulfillment system. This request is routed to the OF system via the XML Router.
7. The OF begins its internal processes which also includes both automated and manual steps.
8. As the order is completed, notifications are sent from the OF system to the OM system. The notifications include completion of the line items within the order and the order itself. The notifications are routed through the XML Router system.
9. A response from each notification is return to the OF as part of the notification/response protocol.
10. Once the OM has received all of the notifications for the line items and the final notification of the total order, it sends a notification to the External Client via the XML Router.
11. The Router receives the notification and sends a response back to the OM.

When all of the scenarios are analyzed, the modeling becomes fairly complex. For example, within the cancel order process, when an order is received by the OM system, five events can occur:

1. The request can be sent immediately to the OF system.
2. The request can be queued in OM for manual processing.
3. The request can be completed immediately without interaction with the OF system.
4. The request can timeout while in the OM. This means that an event should have occurred within a time period but it did not.
5. The notification can be refused by the client. (The project sponsor assumed the XML Router was always operational)

The initial and incomplete analysis state transition diagram for the cancel order is shown below.

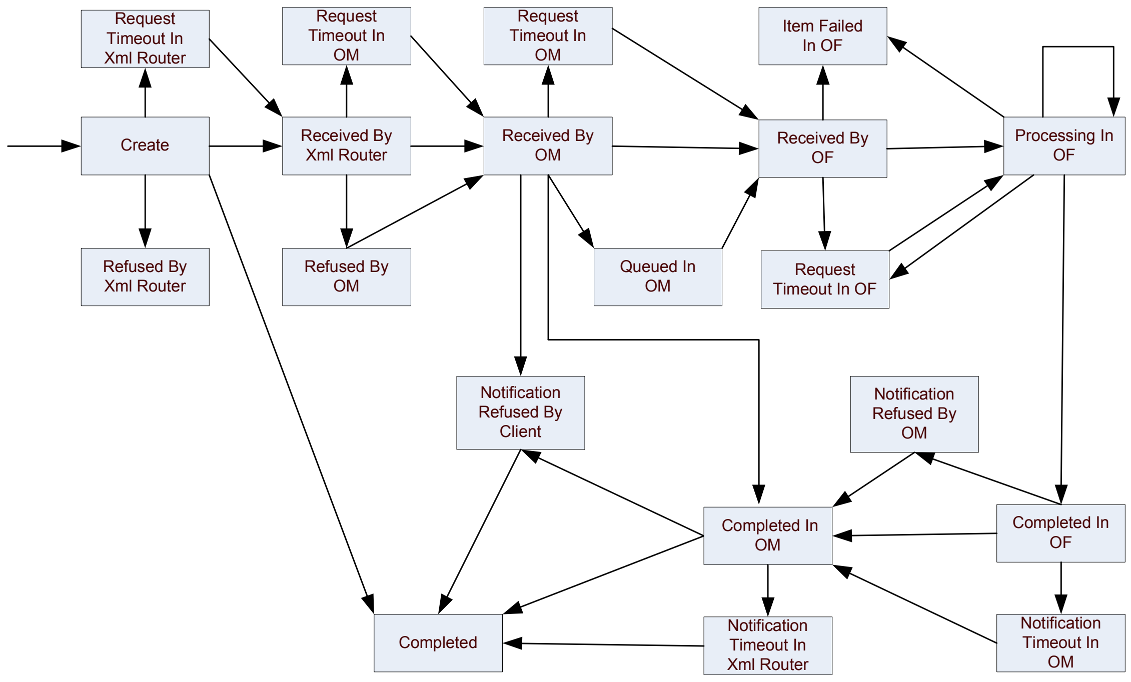


Figure 2 - Cancel Order State Transition Diagram

Obviously, the state transition model of the cancel order is non-trivial. The cancel order process is the *least* complex of the different processes to be monitored. The monitoring consists of detecting, recording, and correlating every XML transaction within the lifecycle of each order processed through the four systems. Due to the significant complexity and the state-oriented nature of the processes, the decision was made to model the lifecycles with finite state model technology.

Finite state models (FSM) have been used for a number of years in a wide spectrum of industries. Example projects using finite state model technology include communication systems, automobiles, avionics systems, and man-machine interfaces. These problem domains share common characteristics: they are usually large in size, high in complexity, and reactive in nature. A primary challenge of these domains is the difficulty of describing reactive behavior in ways that are clear, concise, and complete while at the same time formal and rigorous.

Finite state models provide a way to describe large, complex systems. FSMs view these systems as a set of inbound and outbound events, conditions, and actions. FSM technology also provides a set of rules for evaluating and processing the events, conditions, and actions. The partitioning of the problem into the events, conditions, and actions, the structured processing environment, and the ease of expressing the processing logic are the foremost strengths of FSMs.

The fundamental components of XTM finite state models include:

State represents the “mode of being” for the system at any given time. States contain a set of entry events and exit events. The entry events are published or broadcasted when the state is entered. Exit events are published when the state is exited.

Transition describes a single pathway from a given state to another state. The set of all transitions describe all possible paths among the defined states. A transition contains an event that it is subscribing to, a condition, and an action. A transition also contains the state from which the transition is exiting (i.e. the “from” state) and the state to which the transition is entering (i.e. the “to” state).

Event is the mechanism that the system uses to interact with external systems and with itself.

Condition represents a set of logic that evaluates to a Boolean result. It is used to determine if a transition is to be activated.

Action is the processing to be performed with a transition is activated.

A diagram of the components of a typical FSM is shown below:

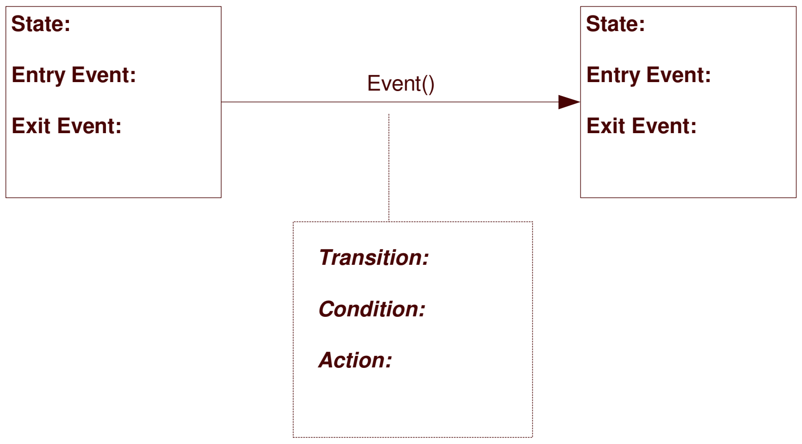


Figure 3 - FSM Components

Since the modeling of the processes leveraged Finite State Machine technologies, it was a logical step to use FSM technology for the implementation of the XTM project. In theory, the transition from analysis models to design artifacts and implementation code should be smooth and require minimal effort. The requirements for the implementation code base were:

1. **Easy to use** – The framework must be straightforward with a rapid learning curve. A large benefit would be a graphical development environment that leverages the visualization of the process models.

2. **Complete** – The framework must completely support the FSM technologies including hierarchical and concurrent models.

3. **Support for persistency and transactions** – Since the process models were very long running, the information must be persisted and done so within a transactional context. The framework should provide built-in transactional capabilities and support for different persistency alternatives.

4. **Java-based** – The code must be Java-based due to the client's development standards.

5. **Inexpensive** – Since the project was an "infrastructure" initiative, the approved budget was very low and costs were to be minimized where possible.

After considering several options, including custom code, the Physhun framework was selected. Physhun is an open source project that provides a robust FSM engine. It allows the modeling, building, and executing of FSM process models in both J2SE and J2EE environments. Physhun supports advanced FSM concepts such as hierarchical (i.e. nested) states, concurrent states, long lived lifecycles, persistency, and transactions. A graphical process modeling tool is also available as a freeware offering. Being open source, the price was right. Physhun met all of the requirements for this project.

Event Generation

For any FSM engine to do meaningful work, it must be presented with events from the outside world. While simple in theory, implementation of this can be very difficult in practice. Event generation consists of three core steps:

1. **Observation** – The FSM system must have access to the information that results in events to be generated. There may be changes in data internal to an application which are important to the process model. If the data cannot be accessed, the appropriate events cannot be generated.

2. **Detection** – Once the information is observed, the FSM system must detect the appropriate change (or lack of change) in the data. This requires filtering the data domain for patterns that result in event creation. Event generation is not dependent on the internal status of the process instances. If conditions are met that causes

an event to be created, the event is always presented to the FSM engine. Since the set of useful events is finite, the filtering may be focused on only that set in order to optimize performance. In some cases, event detection may span multiple sources of observations.

3. **Correlation** – Once an event is created, it is usually associated with a specific process instance. The correlation rules may be simple, such as using a unique business data identifier that maps directly to the process instance, or they may be complex and require multiple data accesses across different data sources.

With the XTM solution, the required observations consisted of a single data source. The XML Transaction Router system is a custom configuration of a commercial off-the-shelf (COTS) package. This system is configured to insert every transaction (i.e. the request/response data and the notify/response data) into an XML-compliant relational database.

The event generation for the XTM project was delegated to a single Java program. The Database Scanner is responsible for examining the XML transactions processed by the XML Transaction Router. When the scanner detects the appropriate conditions, an event is generated and submitted to the Physhun engine for processing. A diagram of the high level components is shown below.

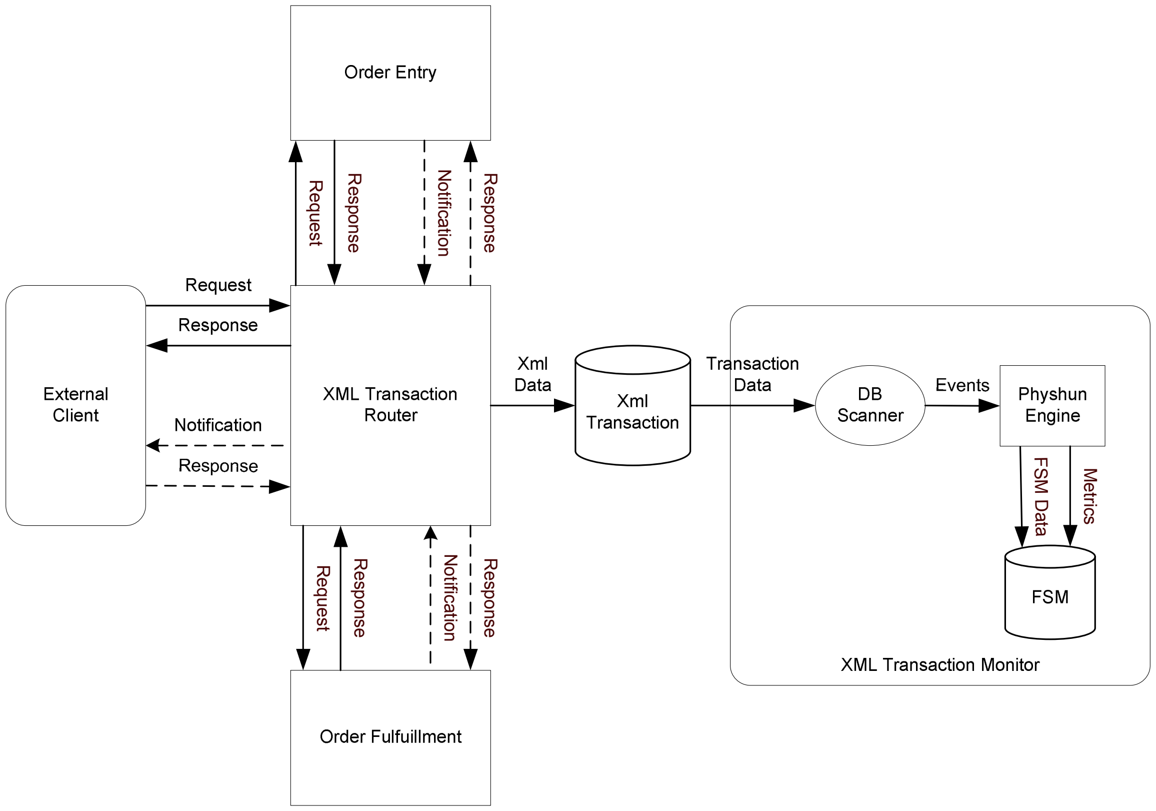


Figure 4 - XTM High Level Components

One of the key requirements with complex event generation is that the chronological order of the events must be maintained. If the events are processed out of sequence, events could be ignored and the process instance becomes invalid in terms of modeling the events in the real world. For example, if an event between the Order Entry and the Order Fulfillment for order #45559 was submitted to the Physhun engine before the create event was received and the process instance for order #45559 was created, the OE-OF event could be ignored temporarily and processed later or dropped forever.

With a solution that uses FSM technology, the event generation strategy must be selected with great consideration. A balance among event volume, processing load, response time, etc must be achieved on a case-by-case basis. For the XTM project, since all of the XML transactions are recorded in the XML Transaction database with a timestamp, we constructed the scanner so that it searches for events in the chronological order and sequence of the XML transactions themselves. If the timestamp information was not available, the scanner would have to be programmed with complex information to allow it to properly sequence events.

Run Time Engine

The core of the XTM solution is the Physhun-based finite state machine engine. This engine is responsible for accepting events from the different scanners, processing the events, and maintaining the state of the process model instances. It created and maintained process instances for eleven different process models. Each model consists of a single configurations file. The XML file contains the Spring configuration information that specifies the states, transitions, and actions of the model. This information is the “execution instructions” for the Physhun engine.

The largest development task with using the Physhun engine was devising a schema to retain the process instance information. Our client’s approved database was Oracle 9 so our persistency mechanism had to be compatible with this database server. The schema that we design is based on a very straightforward relational model as shown below.

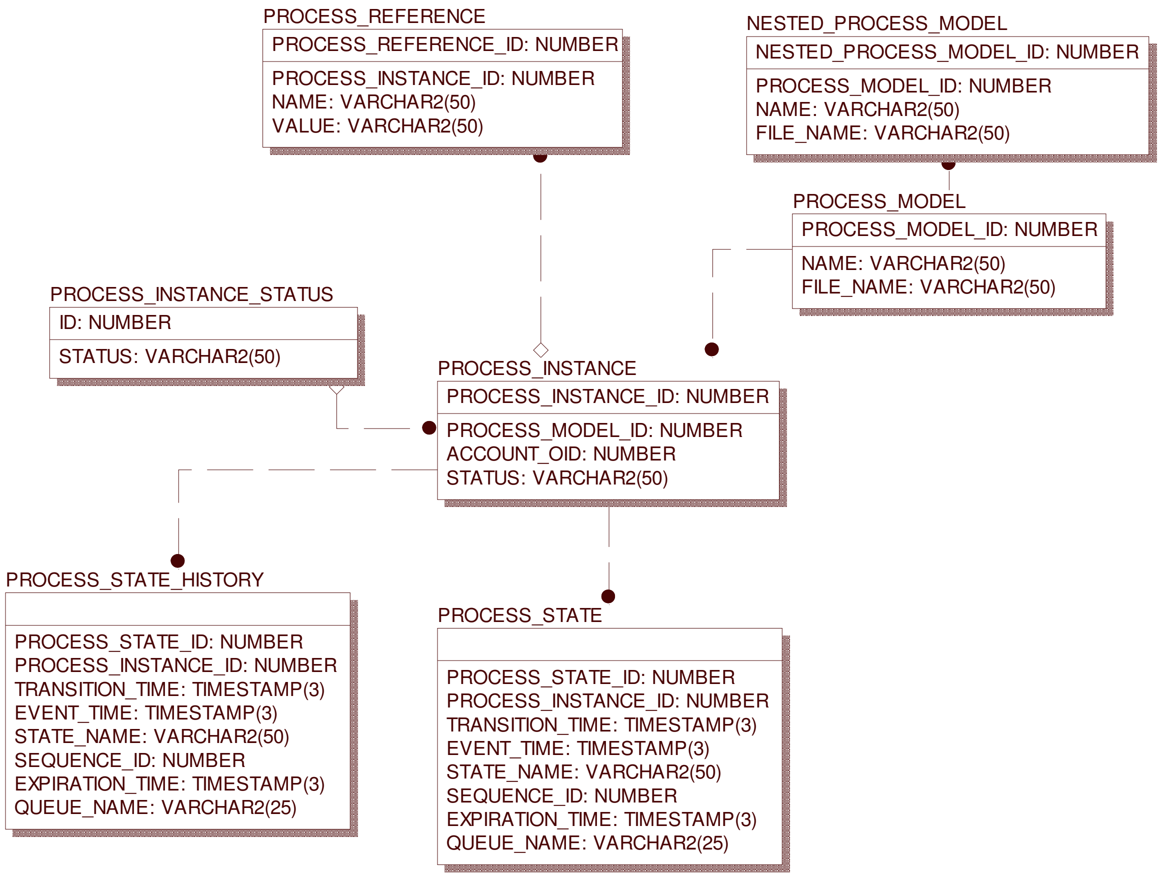


Figure 5 - FSM Database Schema

Physhun Modeler

The Spring-based configuration used by the engine can be generated manually for simple models. However, for non-trivial models, the complexity of the configurations can overwhelm even the most capable designer. Since one of the objectives of using FSM technology is to simplify complex problems, the use of the Physhun Modeler is recommended. The Modeler allows the process model to be visually constructed and maintained, and permits quick and easy modifications as the project evolves. The output of the Modeler is a Spring-based configuration file and an XML Project file that contains the visual layout information used by the Modeler.

The initial process models were entered into to the Modeler tool and evolved throughout the project development lifecycle. The states and transitions are shown in the main drawing area. The Cancel Order process model evolved from the initial analysis model (shown earlier in Figure 2) to the model shown below.

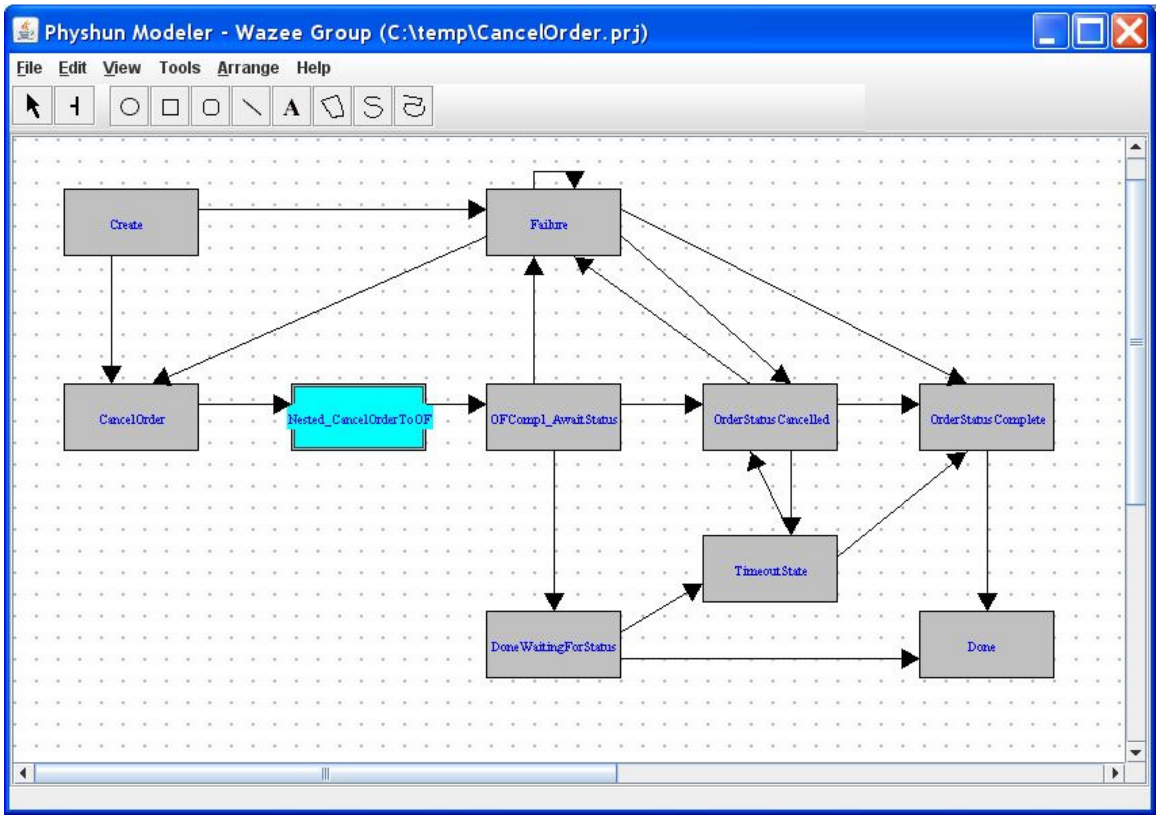
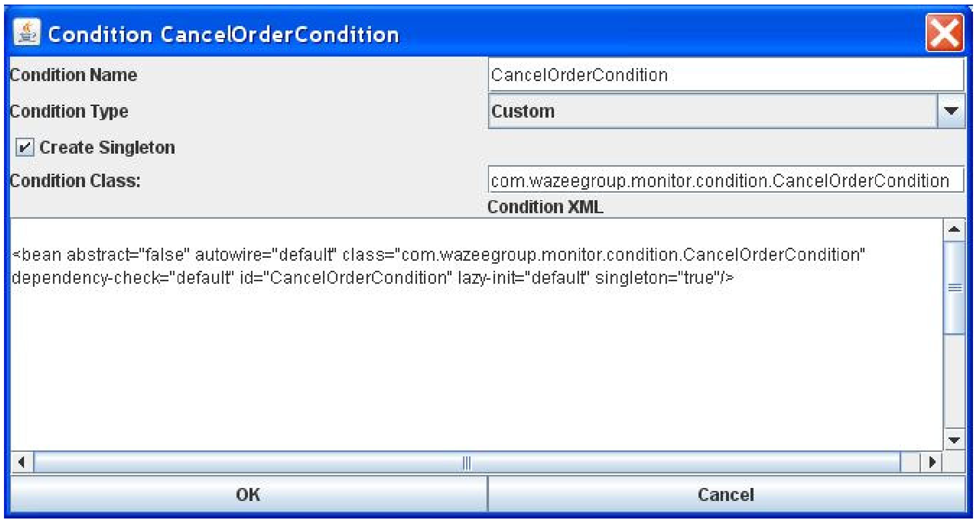


Figure 6- Cancel Order Process

Since actions and conditions are integral components of transitions, they must be specified before transitions can be completed. In the Modeler, there is a Condition Editor where the information regarding specified conditions are maintained. The Spring configuration for the condition is also shown.



Condition Name: CancelOrderCondition

Condition Type: Custom

[✓] Create Singleton

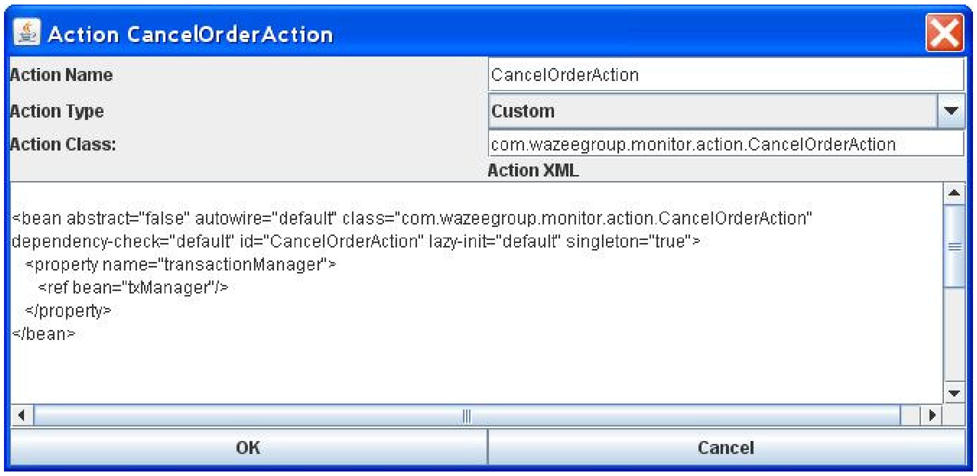
Condition Class: com.wazeegroup.monitor.condition.CancelOrderCondition

Condition XML

```
<bean abstract="false" autowire="default" class="com.wazeegroup.monitor.condition.CancelOrderCondition"
dependency-check="default" id="CancelOrderCondition" lazy-init="default" singleton="true"/>
```

Figure 7 - Cancel Order Condition

The Action Editor is used to create and maintain information of a specific action. The Action Editor is shown below.



Action Name: CancelOrderAction

Action Type: Custom

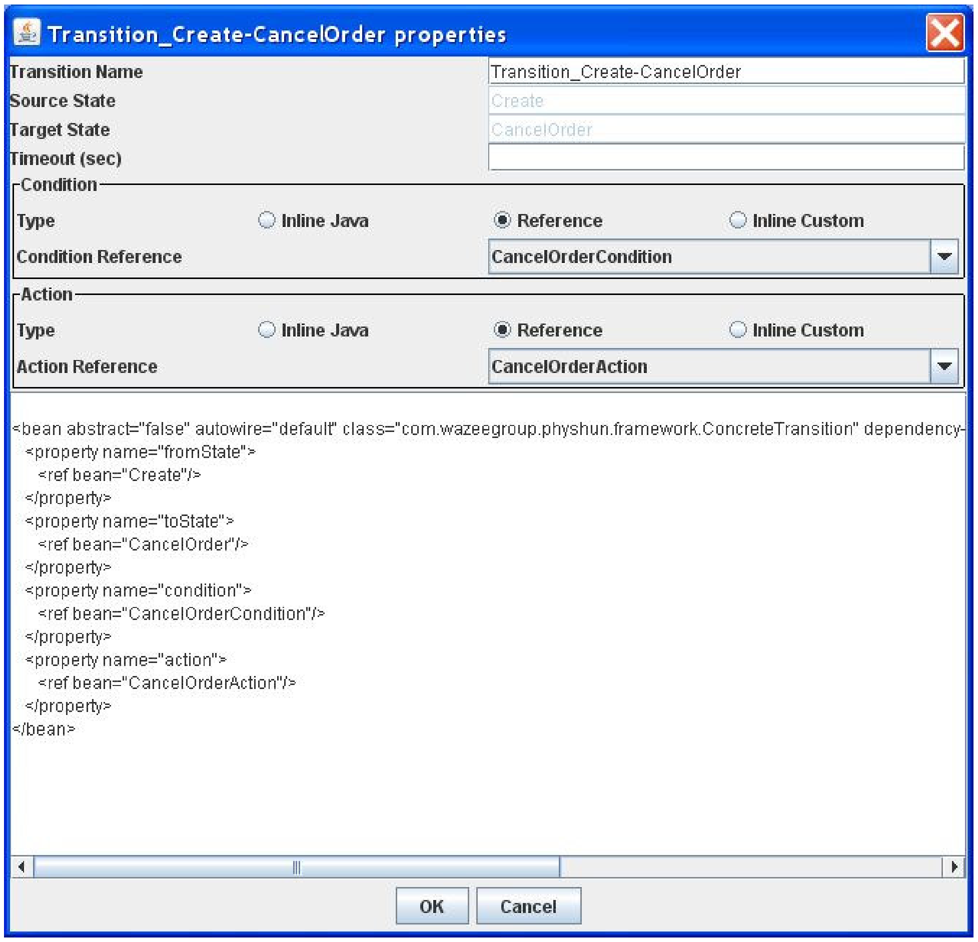
Action Class: com.wazeegroup.monitor.action.CancelOrderAction

Action XML

```
<bean abstract="false" autowire="default" class="com.wazeegroup.monitor.action.CancelOrderAction"
dependency-check="default" id="CancelOrderAction" lazy-init="default" singleton="true">
  <property name="transactionManager">
    <ref bean="txManager"/>
  </property>
</bean>
```

Figure 8 - Cancel Order Action

Once the associated actions and conditions are created, the transitions can be created in the Transition Editor. In this editor, the name, condition, and action information are entered for the specific transition. The Spring configuration is shown in the text area in the lower dialog box.



Transition Name: Transition_Create-CancelOrder
Source State: Create
Target State: CancelOrder
Timeout (sec):

Condition
Type: Inline Java [✓ Reference] Inline Custom
Condition Reference: CancelOrderCondition

Action
Type: Inline Java [✓ Reference] Inline Custom
Action Reference: CancelOrderAction

```
<bean abstract="false" autowire="default" class="com.wazeegroup.physhun.framework.ConcreteTransition" dependency-
  <property name="fromState">
    <ref bean="Create"/>
  </property>
  <property name="toState">
    <ref bean="CancelOrder"/>
  </property>
  <property name="condition">
    <ref bean="CancelOrderCondition"/>
  </property>
  <property name="action">
    <ref bean="CancelOrderAction"/>
  </property>
</bean>
```

Figure 9 - Create Transition

Our process models became more complex as our understanding of the problem became deeper and the requirements of the solution increased. Hierarchical models were used in an attempt to keep the diagrams comprehensible. While mathematically equivalent to a single layer model, nested models can simplify the process model at each level within the hierarchy, making them more human-comprehensible and reducing the cognitive burden of model maintenance. In the Cancel Order process model show previously, the Nested_CancelOrderToOF state is a nested state. It consists of a substate within a lower level process model. The lower level model is the CancelOrderToOF model and is shown below.

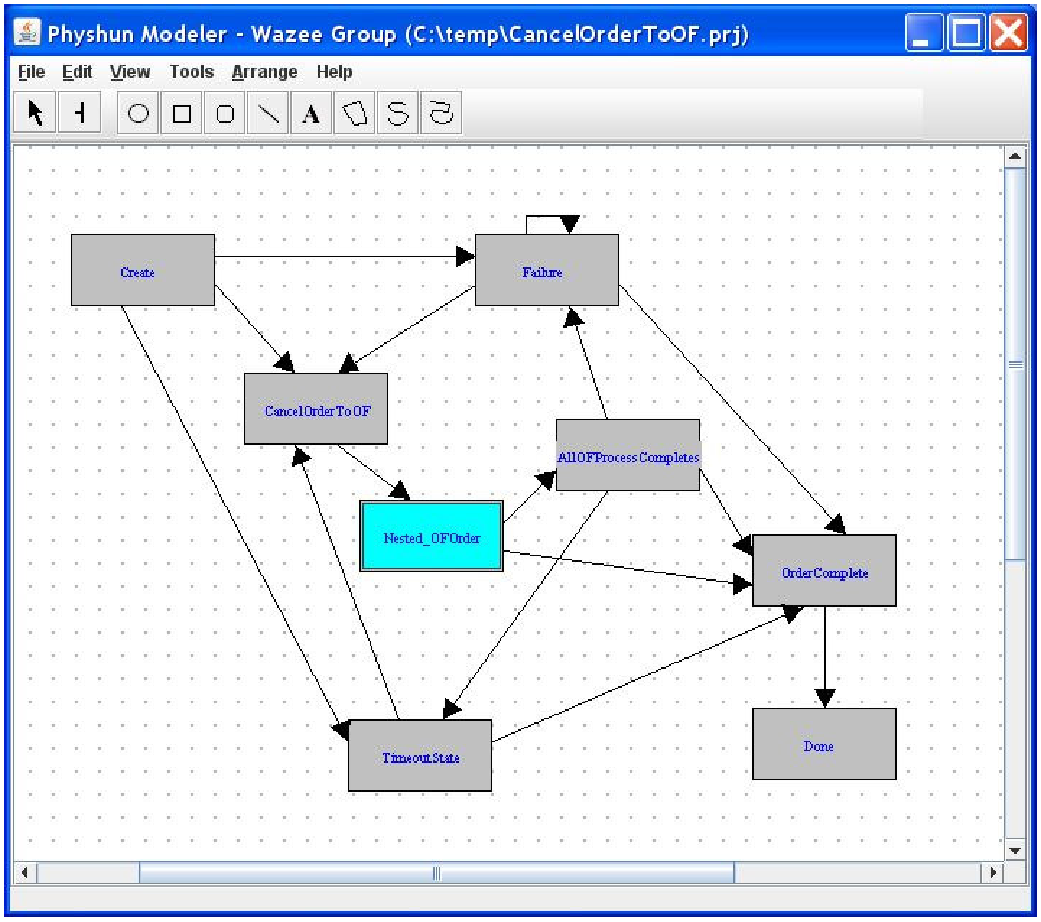


Figure 10 - Cancel Order To OF Model

Simiilarly, the Nested_OFOrder state in the CancelOrderToOF process model maps to the OFOrder process model as shown below.

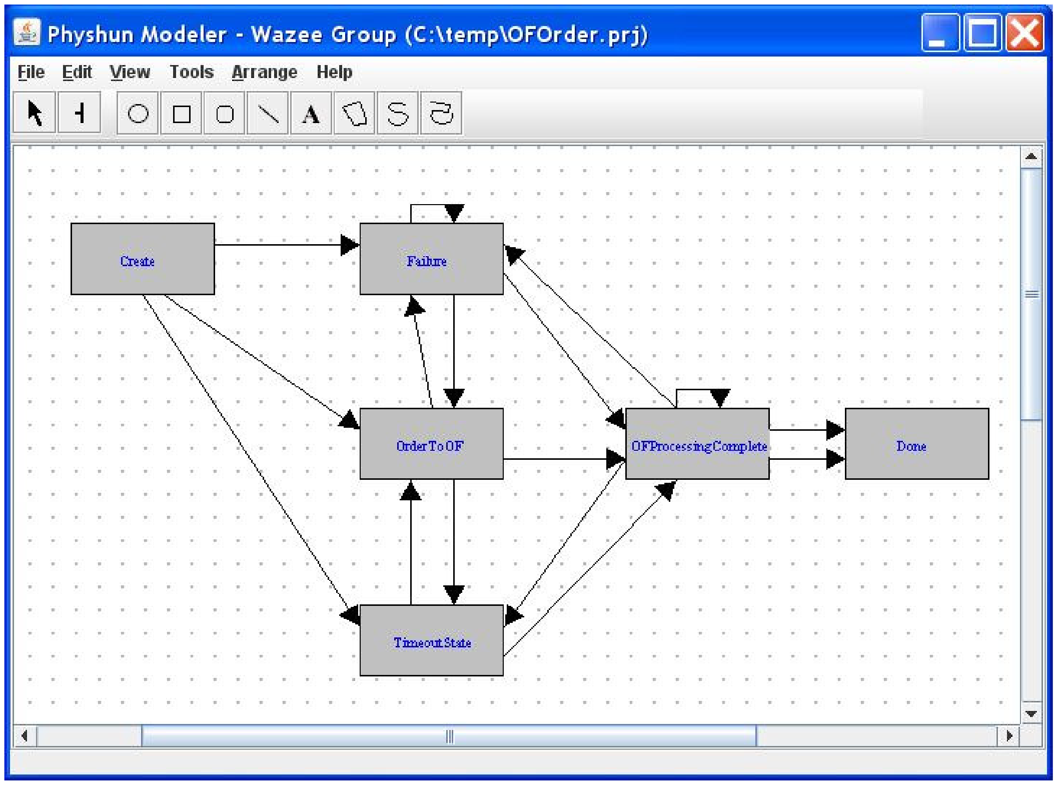

Physhun Modeler - Wazee Group (C:\temp\OFOrder.prj)
File Edit View Tools Arrange Help
Create
Failure
OrderToOF
OFProcessingComplete
Done
TimeoutState

Figure 11 - OF Order Model

Summary

Obviously, there is more to the XTM project than is presented in this paper. The solution as developed also included gathering and presenting transaction-based metrics, various reporting methodologies, notifications of errors and potential performance issues. All of these areas were implemented with well-known technologies and frameworks. The cornerstone of the XTM project is the Physhun framework. The use of this framework allowed the XTM project to be designed, developed, and deployed well within the allotted schedule and budget by a single developer.