UML Model - State Machines 1.0 Component Specification

The UML Model - State Machines component declares the interfaces from the UML 1.5 framework, from the State Machines package. It provides concrete implementations for each interface and provides powerful API to access the collection attributes.
This is basically an object model for a State Machine, which is used for modeling of state information.

1. Design

This component is really something of a fundamental base for the state diagrams that are used in UML. It defines a set of interfaces (contracts) and basic implementations for the purpose of defining a State Machine. The following have been defined

1. Contract and relations definitions
2. Base implementations

1.1.1 Base implementations decisions

It was decided (actual requirement) that only very basic validation of passed in parameters will be done, and only on the level of input that directly affects lists and collections. Thus we do not allow null elements to be placed into lists or collections. We do allow the setting of null collections/lists though as well as other non-primitive instances. The idea is that validation and the assurance of proper input data should be mostly decided by a higher level which would be put around and on top of this component.

1.1.2 DOM Validation

It is assumed that DOM validation would be done outside of the DOM itself. This means that we are anticipating something of a validation framework, which would take as input a State Machine for example (or a StateVertex) and then validate it based on some external rules.

1.2 Design Patterns

Composite has been used to model the different states since they are made up of other nodes including other states, transitions, and elements.

1.3 Industry Standards

UML 1.5

1.4 Required Algorithms

Please refer to the class docs. There are no real algorithms here.

1.5 Component Class Overview

1.5.1 Interfaces

Guard
A guard is a condition that must be true in order to traverse a transition. It prevents the transition being taken unless the condition evaluates to true. To expand on this concept: A guard is a boolean expression that is attached to a transition as a fine-grained control over its firing. The guard is evaluated when an event instance is dispatched by the state machine. If the guard is true at that time, the transition is enabled, otherwise, it is disabled. Guards should be pure expressions without side effects. Guard expressions with side effects are ill formed.

NOTE: Guards Should Not Overlap. The guards on similar transitions leaving a state must be consistent with one another. For example guards such as [x <0], [x = 0], and [x > 0] are consistent whereas guard such as [x <= 0] and [x >= 0] are not consistent because they overlap.
Implementations are mutable, there is no requirement to make implementations of this interface thread-safe.

Transition
A transition is a progression (or a moving from) from one state to another and is triggered by an event that is either internal or external to the entity being modeled. For a class, transitions are typically the result of the invocation of an operation that causes an important change in state, although it is important to understand that not all method invocations will result in transitions. An action is something; in the case of a class it is an operation that is invoked by/on the entity being modeled.
A transition is a directed relationship between a source vertex and a target vertex. It may be part of a compound transition, which takes the state machine from one state configuration to another, representing the complete response of the state machine to an occurrence of an event of a particular type.(OMG Unified Modeling Language Specification - UML 2.0 Superstructure Specification, p. 624)
Implementations are mutable; there is no requirement to make implementations of this interface thread-safe.

StateMachine
A State Machine models the behavior of a single object, specifying the sequence of events that an object goes through during its lifetime in response to events.
It is a specification that describes all possible behaviors of some dynamic model element (i.e. object). Behavior is modeled as a traversal of a graph of state nodes interconnected by one or more joined transitions (i.e. transition arcs) that are triggered by the dispatching of series of event instances. During this traversal, the state machine executes a series of actions associated with various elements of the state machine. StateMachine contains a top State, which represents the top-level state, and a set of transitions. This means that a state machine owns its transitions and its top state. All remaining states are transitively owned through the state containment hierarchy rooted in the top state (i.e. top state is like a root in a graph) The association to `ModelElement` provides the context of the state machine. A common case of the context relation is where a state machine is used to specify the lifecycle of a classifier.
Implementations are mutable, there is no requirement to make implementations of this interface thread-safe.

StateVertex
A StateVertex is an abstraction of a node in a state chart (state machine) graph. In general, it can be the source or destination of any number of transitions.
All types of states in a state machine are children of StateVertex. Currently we have the following types of vertexes: Pseudostate, State, FinalState, SimpleState, CompositeState (recursive since a StateVertex aggregates a CompositeState as well)
Implementations are mutable, there is no requirement to make implementations of this interface thread-safe.

Pseudostate
Pseudostate are choice vertices which, when reached, result in the dynamic evaluation of the guards of the triggers of its outgoing transitions. This realizes a dynamic conditional branch. It allows splitting of transitions into multiple outgoing paths such that the decision on which path to take may be a function of the results of prior actions performed in the same run-to-completion step.(OMG Unified Modeling Language Specification - UML 2.0 Superstructure Specification, p. 592) Another quick way of defining it would be to state that it is a vertex in a state machine that has the form of a state, but doesn’t behave as a

state.
Implementations are mutable, there is no requirement to make implementations of this interface thread-safe.

State
A state is a stage (or step) in the behavior life cycle of an entity. Another way of stating it is that a state is a condition during the life of an object or an interaction during which it satisfies some condition, performs some action, or waits for some event.
A state is associated with a state-machine to which it belongs.
Implementations are mutable; there is no requirement to make implementations of this interface thread-safe.

CompositeState
A composite state is decomposed into two or more concurrent substates (called regions) or into mutually exclusive
disjoint substates (i.e. sequential). A given state may only be refined in one of these two ways. Naturally, any
substate of a composite state can also be a composite state of either type. (OMG Unified Modeling Language
Specification - UML 1.5 UML Notation, p. 540)
A composite state is a state that contains other state vertices (states, pseudostates, etc.). The association between
the composite and the contained vertices is a composition association. Hence, a state vertex can be a part of at most
one composite state.

SimpleState
A simple state is a state that does not have sub states, i.e. it has no regions and it has no submachine state machine.(OMG Unified Modeling Language Specification - UML 2.0 Superstructure Specification, p. 600) We could state that a simple state is the opposite of a Composite state.
There is no requirement to make implementations of this interface thread-safe.

FinalState
A final state represents the completion of activity in the enclosing state and it triggers a transition on the enclosing state labeled by the implicit activity completion event, if such a transition is defined. If the enclosing region is directly contained in a state machine and all other regions in the state machine also are completed, then it means that the entire state machine is completed. (OMG Unified Modeling Language Specification - UML 2.0 Superstructure Specification, p. 580)
There is no requirement to make implementations of this interface thread-safe

1.5.2 *Implementations including abstract and concrete*

StateMachineImpl
A simple implementation of the StateMachine interface. Not thread-safe.

TransitionImpl
A simple implementation of the Transition interface. Not thread-safe.

GuardImpl
A simple implementation of the Guard interface. Not thread-safe.

StateVertexAbstractImpl
An abstract implementation of the StateVertex interface. Not thread-safe.

PseudoStateImpl
A simple implementation of the Pseudostate interface. Not thread-safe.

AbstractStateImpl
A abstract implementation of the State interface. Not thread-safe.

FinalStateImpl
A simple implementation of the FinalState interface. Not thread-safe.

SimpleStateImpl
A simple implementation of the SimpleState interface. Not thread-safe.

CompositeStateImpl
A simple implementation of the CompositeState interface. Not thread-safe.

1.6 Component Exception Definitions

1.6.1 System exceptions

- **IllegalArgumentException**: Exception thrown in various methods where value is invalid or null. In our case we only use this when manipulating element data for any collection or list,or if the collection is null or contains null element. This is the only portion of the API, which will inherently test input parameters for being null.

1.7 Thread Safety

It is has been stated as a requirement that the Dom should not be inherently thread-safe and as such it is not. No specific precautions have been made to ensure thread-safety.

All implemented classes are mutable which is the main reason why they are not thread-safe. To make them thread-safe the developer would need to synchronize access to each internal variable. This could be achieved by simply declaring all methods synchronized or we could use a synchronized block and lock on specific variables (when dealing with instances) and locking on dummy variables (instances) for primitive variables.

One aspects that has been ensured is that lists and collections are internally encapsulated (shallow) which means that any input in terms of a list/collection or any output in terms of collection/list is always copied (shallow copy)

2. Environment Requirements

2.1 Environment

jdk 1.5+

2.2 TopCoder Software Components

Directly use:

com.topcoder.uml.model.core: many interface extends from ModelElement
com.topcoder.uml.model.datatypes.expressions BooleanExpression will be used.

Indirectly use:

com.topcoder.uml.model.activitygraphs
com.topcoder.uml.model.commonbehavior.instances
com.topcoder.uml.model.commonbehavior.links
com.topcoder.uml.model.core.auxiliaryelements

com.topcoder.uml.model.core.dependencies
com.topcoder.uml.model.core.relationships
com.topcoder.uml.model.extensionmechanisms
com.topcoder.uml.model.modelmanagement

2.3 Third Party Components

None

3. Installation and Configuration

3.1 Package Name

com.topcoder.uml.model.statemachines

3.2 Configuration Parameters

No configuration.

3.3 Dependencies Configuration

None

4. Usage Notes

4.1 Required steps to test the component

- Extract the component distribution.
- Follow Dependencies Configuration.
- Execute 'ant test' within the directory that the distribution was extracted to.

4.2 Required steps to use the component

This is a basic state Machine DOM definition. It can be used to create a OM for a state machine. Please refer to the demo section for examples.

4.3 Demo

In this demo we will demonstrate how to create a DOM representation of a simple state machine:

4.3.1 A simple state machine with a couple states

We will model the following diagram:

abort

Select Amount

other amount [amount<=500]

amount [amount<=500]

Other Amount

abort

Ok

```
       // create an empty state machine
        StateMachine myStateMachine = new StateMachineImpl();
        // Create all the states
        // create the top state
        CompositeState top = new CompositeStateImpl();
        top.setName("Select Amount");
        // create another state
        State otherAmount = new SimpleStateImpl();
        otherAmount.setName("Other Amount");
        // create a final state
        FinalState finalState = new FinalStateImpl();
        finalState.setName("Final State");
        // create a abort state
        Pseudostate abortState = new PseudostateImpl();
        abortState.setName("Abort State");
        abortState.setKind(PseudostateKind.JUNCTION);

        // Create all the transitions (some with guards)

        // abort transition#1
        Transition abortTransition = new TransitionImpl();
        abortTransition.setName("abort");
        abortTransition.setSource(top);
        abortTransition.setTarget(abortState);

        // abort transition#2
        Transition abortTransition2 = new TransitionImpl();
        abortTransition2.setName("abort");
        abortTransition2.setSource(otherAmount);
        abortTransition2.setTarget(abortState);

        // Ok transition
        Transition okTransition = new TransitionImpl();
        okTransition.setName("Ok");
        okTransition.setSource(otherAmount);
        okTransition.setTarget(finalState);

        // amount transition
        Transition amountTransition = new TransitionImpl();
        amountTransition.setName("Amount");
        amountTransition.setSource(top);
        amountTransition.setTarget(finalState);
        // create a new guard and add it to the transition
        Guard amountTransitionGuard = new GuardImpl(new BooleanExpressionImpl(),
amountTransition);
        amountTransition.setGuard(amountTransitionGuard);

        // Other Amount transition
        Transition otherAmountTransition = new TransitionImpl();
        otherAmountTransition.setName("Other Amount");
        otherAmountTransition.setSource(top);
        otherAmountTransition.setTarget(otherAmount);
        // create a new guard and add it to the transition
        Guard otherAmountTransitionGuard = new GuardImpl(new BooleanExpressionImpl(),
  otherAmountTransition);
        otherAmountTransition.setGuard(otherAmountTransitionGuard);

        // Add transitions to states

        // top state
        top.addOutgoingTransition(abortTransition);
        top.addOutgoingTransition(amountTransition);
        top.addOutgoingTransition(otherAmountTransition);

        // Other Amount State
        otherAmount.addOutgoingTransition(abortTransition2);
        otherAmount.addOutgoingTransition(okTransition);
        otherAmount.addIncomingTransition(otherAmountTransition);

        // Final State
        finalState.addIncomingTransition(amountTransition);
        finalState.addIncomingTransition(okTransition);

        // Abort State
        abortState.addIncomingTransition(abortTransition);
        abortState.addIncomingTransition(abortTransition2);

        // Add all transitions to the state machine

        // add top state to the state machine
        myStateMachine.setTop(top);
        // add transitions to the state machine
        myStateMachine.addTransition(abortTransition);
        myStateMachine.addTransition(abortTransition2);
        myStateMachine.addTransition(okTransition);
        myStateMachine.addTransition(amountTransition);
        myStateMachine.addTransition(otherAmountTransition);
```

5. Future Enhancements

- Provide DOM validation