UML Tool Actions – Activity Elements Actions 1.0 Component Specification

1. Design

The Activity Elements Actions component provides the Actions related to the model elements specific to an activity diagram. The actions are strategy implementations of the action interfaces in the Action Manager component. The provided actions are for adding / removing / copying / cutting / pasting the elements and relationships. The elements are initial node, object flow node, action state, send signal action, accept event action, fork node, join node, decision node, merge node, flow final node and final node. The relationship is transition.

This design is quite straightforward, and it follows the requirement specification exactly to implement all required activity elements actions as mentioned above.

The one thing that worth mention is the cut/copy logic used in this design, as we ought to cut/copy the objects across different running applications/JVMs, the custom Transferable implementation (the ActivityObjectSelection class) and several DataFlavor objects provided by this design will make the cut/copied objects be serialized to the clipboard, and we eventually get the deserialized objects from clipboard and pass them to the paste actions. Both serialization and deserialization mentioned above are done by the JDK automatically; this design only follows the necessary principles in order to make it work.

1.1 Design Patterns

- **Strategy Pattern** – All concrete action classes in this design are strategy implementations of the action interfaces in the Action Manager component.
- **Utility Pattern –** The ActivityObjectClipboardUtility and ActivityObjectCloneUtility classes implement this pattern to provide utility methods.

1.2 Industry Standards

UML 1.5

1.3 Required Algorithms

For those are not quite familiar with the java Clipboard stuff, the following link might be helpful:
http://www.javaworld.com/javaworld/jw-08-1999/jw-08-draganddrop.html

1.3.1 Check the tagged value of specific tag definition

TagDefinition(tagType).value = dataValue appears a lot in the documentation, it means that the node contains a TaggedValue object, whose TaggedDefinition's tagType is as specified, and the TaggedValue's dataValue is as specified.

To do such a check, we first call ModelElement#getTaggedValues() to get all registered TaggedValue objects, and then for each TaggedValue object, get its TagDefinition (TaggedValue#type), if the TagDefinition#tagType is as expected, check the TaggedValue#dataValue is as expected too. The element is valid if one such TaggedValue object is found.

1.4 Component Class Overview

1.4.1 Package com.topcoder.uml.actions.model.activity

- **AbstractActivityUndoableAction**: This abstract class implements the UndoableAction interface and extends the AbstractUndoableEdit abstract class. It is

used as the base class for all the undoable actions in this design, it also provides a logging method that could be used by its subclasses to log the exceptions raised in the redo/undo methods, and its name attribute is used as the action's presentation name.

- **AddStateNodeAbstractAction**: AddStateNodeAbstractAction abstract class extends AbstractActivityUndoableAction abstract class and this action will simply add the StateVertex node into the ActivityGraph. It will also pass the element to the ProjectConfigurationManager, to apply any initial formatting.
- **AddInitialNodeAction**: AddInitialNodeAction class extends AddStateNodeAbstractAction abstract class, and this action is specifically used to add Pseudostate node whose kind attribute equals to PseudostateKind.INITIAL in activity graph.
- **AddObjectFlowStateAction**: AddObjectFlowStateAction class extends AddStateNodeAbstractAction abstract class, and this action is specifically used to add ObjectFlowState node in activity graph.
- **AddActionStateAction**: AddActionStateAction class extends AddStateNodeAbstractAction abstract class, and this action is specifically used to add ActionState node in activity graph.
- **AddSendSignalActionAction**: AddSendSignalActionAction class extends AddStateNodeAbstractAction abstract class, and this action is specifically used to add SimpleState with a tag definition attached (TagDefinition("SendSignalAction").value="True") in activity graph.
- **AddAcceptEventActionAction**: AddAcceptEventActionAction class extends AddStateNodeAbstractAction abstract class, and this action is specifically used to add SimpleState with a tag definition attached (TagDefinition("AcceptEventAction").value="True") in activity graph.
- **AddForkNodeAction**: AddForkNodeAction class extends AddStateNodeAbstractAction abstract class, and this action is specifically used to add Pseudostate node whose kind attribute equals to PseudostateKind.FORK in activity graph.
- **AddJoinNodeAction**: AddJoinNodeAction class extends AddStateNodeAbstractAction abstract class, and this action is specifically used to add Pseudostate node whose kind attribute equals to PseudostateKind.JOIN in activity graph.
- **AddDecisionNodeAction**: AddDecisionNodeAction class extends AddStateNodeAbstractAction abstract class, and this action is specifically used to add Pseudostate node whose kind attribute equals to PseudostateKind.CHOICE in activity graph.
- **AddMergeNodeAction**: AddMergeNodeAction class extends AddStateNodeAbstractAction abstract class, and this action is specifically used to add Pseudostate node whose kind attribute equals to PseudostateKind.JUNCTION in activity graph.
- **AddFlowFinalNodeAction**: AddFlowFinalNodeAction class extends AddStateNodeAbstractAction abstract class, and this action is specifically used to add FinalState node with a tag definition attached (TagDefinition("FinalNodeType").value="FlowFinalNode") in activity graph.
- **AddFinalNodeAction**: AddFinalNodeAction class extends AddStateNodeAbstractAction abstract class, and this action is used specifically to add FinalState node into activity graph.

- **AddTransitionAction**: AddTransitionAction class implements the UndoableAction interface and this action will simply add the Transition object into the ActivityGraph. It will also pass the element to the ProjectConfigurationManager, to apply any initial formatting.

- **CopyStateNodeAbstractAction**: CopyStateNodeAbstractAction abstract class implements the TransientAction interface. It is the base class for all copy actions; it is used to copy the StateVertex node to the clipboard.
- **CopyInitialNodeAction**: CopyInitialNodeAction class extends CopyStateNodeAbstractAction abstract class. This action is used specially to copy Pseudostate node with kind attribute equal to PseudostateKind.INITIAL to the clipboard.
- **CopyObjectFlowStateAction**: CopyObjectFlowStateAction class extends CopyStateNodeAbstractAction abstract class. This action is used specially to copy ObjectFlowState node to the clipboard.
- **CopyActionStateAction**: CopyActionStateAction class extends CopyStateNodeAbstractAction abstract class. This action is used specially to copy ActionState node to the clipboard.
- **CopySendSignalActionAction**: CopySendSignalActionAction class extends CopyStateNodeAbstractAction abstract class. This action is used specially to copy SimpleState node with a tag definition attached (TagDefinition("SendSignalAction").value="True") to the clipboard.
- **CopyAcceptEventActionAction**: CopyAcceptEventActionAction class extends CopyStateNodeAbstractAction abstract class. This action is used specially to copy SimpleState node with a tag definition attached (TagDefinition("AcceptEventAction").value="True") to the clipboard.
- **CopyForkNodeAction**: CopyForkNodeAction class extends CopyStateNodeAbstractAction abstract class. This action is used specially to copy Pseudostate node with kind attribute equal to PseudostateKind.FORK to the clipboard.
- **CopyJoinNodeAction**: CopyJoinNodeAction class extends CopyStateNodeAbstractAction abstract class. This action is used specially to copy Pseudostate node with kind attribute equal to PseudostateKind.JOIN to the clipboard.
- **CopyDecisionNodeAction**: CopyDecisionNodeAction class extends CopyStateNodeAbstractAction abstract class. This action is used specially to copy Pseudostate node with kind attribute equal to PseudostateKind.CHOICE to the clipboard.
- **CopyMergeNodeAction**: CopyMergeNodeAction class extends CopyStateNodeAbstractAction abstract class. This action is used specially to copy Pseudostate node with kind attribute equal to PseudostateKind.JUNCTION to the clipboard.
- **CopyFlowFinalNodeAction**: CopyFlowFinalNodeAction class extends CopyStateNodeAbstractAction abstract class. This action is used specially to copy FinalState node with a tag definition attached (TagDefinition("FinalNodeType").value="FlowFinalNode") to the clipboard.
- **CopyFinalNodeAction**: CopyMergeNodeAction class extends CopyStateNodeAbstractAction abstract class. This action is used specially to copy FinalState node to the clipboard.

- **CopyTransitionAction**: CopyTransitionAction class implements the TransientAction interface. It is used to copy the Transition object to the clipboard.

- **CutStateNodeAbstractAction**: CutStateNodeAbstractAction abstract class extends AbstractActivityUndoableAction abstract class. This action will copy the StateVertex node to the clipboard, and then remove this node from its attached activity graph.
- **CutInitialNodeAction**: CutInitialNodeAction class extends CutStateNodeAbstractAction abstract class. This action is used specially to cut Pseudostate node with kind attribute equal to PseudostateKind.INITIAL to the clipboard, this node will also be removed from its attached activity graph.
- **CutObjectFlowStateAction**: CutObjectFlowStateAction class extends CutStateNodeAbstractAction abstract class. This action is used specially to cut ObjectFlowState node to the clipboard, this node will also be removed from its attached activity graph.
- **CutActionStateAction**: CutActionStateAction class extends CutStateNodeAbstractAction abstract class. This action is used specially to cut ActionState node to the clipboard, this node will also be removed from its attached activity graph.
- **CutSendSignalActionAction**: CutSendSignalActionAction class extends CutStateNodeAbstractAction abstract class. This action is used specially to cut SimpleState node with a tag definition attached (TagDefinition("SendSignalAction").value="True") to the clipboard, this node will also be removed from its attached activity graph.
- **CutAcceptEventActionAction**: CutAcceptEventActionAction class extends CutStateNodeAbstractAction abstract class. This action is used specially to cut SimpleState node with a tag definition attached (TagDefinition("AcceptEventAction").value="True") to the clipboard, this node will also be removed from its attached activity graph.
- **CutForkNodeAction**: CutForkNodeAction class extends CutStateNodeAbstractAction abstract class. This action is used specially to cut Pseudostate node with kind attribute equal to PseudostateKind.FORK to the clipboard, this node will also be removed from its attached activity graph.
- **CutJoinNodeAction**: CutJoinNodeAction class extends CutStateNodeAbstractAction abstract class. This action is used specially to cut Pseudostate node with kind attribute equal to PseudostateKind.JOIN to the clipboard, this node will also be removed from its attached activity graph.
- **CutDecisionNodeAction**: CutDecisionNodeAction class extends CutStateNodeAbstractAction abstract class. This action is used specially to cut Pseudostate node with kind attribute equal to PseudostateKind.CHOICE to the clipboard, this node will also be removed from its attached activity graph.
- **CutMergeNodeAction**: CutMergeNodeAction class extends CutStateNodeAbstractAction abstract class. This action is used specially to cut Pseudostate node with kind attribute equal to PseudostateKind.JUNCTION to the clipboard, this node will also be removed from its attached activity graph.
- **CutFlowFinalNodeAction**: CutFlowFinalNodeAction class extends CutStateNodeAbstractAction abstract class. This action is used specially to cut FinalState node with a tag definition attached (TagDefinition("FinalNodeType").value="FlowFinalNode") to the clipboard, this node will also be removed from its attached activity graph.

- **CutFinalNodeAction**: CutObjectFlowStateAction class extends CutStateNodeAbstractAction abstract class. This action is used specially to cut FinalState node to the clipboard, this node will also be removed from its attached activity graph.
- **CutTransitionAction**: CutTransitionAction class implements the UndoableAction interface. This action will copy the Transition object to the clipboard, and then remove this object from the its attached activity graph.

- **PasteStateNodeAbstractAction**: PasteStateNodeAbstractAction abstract class extends AbstractActivityUndoableAction abstract class. This action adds the StateVertex object retrieved from the clipboard into the specified activity graph.
- **PasteInitialNodeAction**: This class extends PasteStateNodeAbstractAction abstract class. This action adds the Pseudostate node with kind attribute equal to PseudostateKind.INITIAL retrieved from the clipboard into the specified activity graph.
- **PasteObjectFlowStateAction**: This class extends PasteStateNodeAbstractAction abstract class. This action adds the ObjectFlowState node retrieved from the clipboard into the specified activity graph.
- **PasteActionStateAction**: This class extends PasteStateNodeAbstractAction abstract class. This action adds the ActionState node retrieved from the clipboard into the specified activity graph.
- **PasteSendSignalActionAction**: This class extends PasteStateNodeAbstractAction abstract class. This action adds the SimpleState node with a tag definition attached (TagDefinition("SendSignalAction").value="True") retrieved from the clipboard into the specified activity graph.
- **PasteAcceptEventActionAction**: This class extends PasteStateNodeAbstractAction abstract class. This action adds the SimpleState node with a tag definition attached (TagDefinition("AcceptEventAction").value="True") retrieved from the clipboard into the specified activity graph.
- **PasteForkNodeAction**: This class extends PasteStateNodeAbstractAction abstract class. This action adds the Pseudostate node with kind attribute equal to PseudostateKind.FORK retrieved from the clipboard into the specified activity graph.
- **PasteJoinNodeAction**: This class extends PasteStateNodeAbstractAction abstract class. This action adds the Pseudostate node with kind attribute equal to PseudostateKind.JOIN retrieved from the clipboard into the specified activity graph.
- **PasteDecisionNodeAction**: This class extends PasteStateNodeAbstractAction abstract class. This action adds the Pseudostate node with kind attribute equal to PseudostateKind.CHOICE retrieved from the clipboard into the specified activity graph.
- **PasteMergeNodeAction**: This class extends PasteStateNodeAbstractAction abstract class. This action adds the Pseudostate node with kind attribute equal to PseudostateKind.JUNCTION retrieved from the clipboard into the specified activity graph.
- **PasteFlowFinalNodeAction**: This class extends PasteStateNodeAbstractAction abstract class. This action adds the FinalState node with a tag definition attached (TagDefinition("FinalNodeType").value="FlowFinalNode") retrieved from the clipboard into the specified activity graph.

- **PasteFinalNodeAction**: This class extends PasteStateNodeAbstractAction abstract class. This action adds the FinalState node retrieved from the clipboard into the specified activity graph.
- **PasteTransitionAction**: PasteTransitionAction class implements UndoableAction interface. This action adds the Transition object retrieved from the clipboard into the specified activity graph.

- **RemoveStateNodeAbstractAction**: RemoveStateNodeAbstractAction abstract class extends AbstractActivityUndoableAction abstract class and it is the parent class of all actions regarding to remove a StateVertex object from the activity graph it is attached to.
- **RemoveInitialNodeAction**: RemoveInitialNodeAction class extends RemoveStateNodeAbstractAction abstract class, it is used to remove the Pseudostate node with kind attribute equal to PseudostateKind.INITIAL.
- **RemoveObjectFlowStateAction**: RemoveObjectFlowStateAction class extends RemoveStateNodeAbstractAction abstract class, it is used to remove the ObjectFlowState node.
- **RemoveActionStateAction**: RemoveActionStateAction class extends RemoveStateNodeAbstractAction abstract class; it is used to remove the ActionState node.
- **RemoveSendSignalActionAction**: RemoveSendSignalActionAction class extends RemoveStateNodeAbstractAction abstract class, it is used to remove the SimpleState node with a tag definition attached (TagDefinition("SendSignalAction").value="True").
- **RemoveAcceptEventActionAction**: RemoveAcceptEventActionAction class extends RemoveStateNodeAbstractAction abstract class, it is used to remove the SimpleState node with a tag definition attached (TagDefinition("AcceptEventAction").value="True").
- **RemoveForkNodeAction**: RemoveForkNodeAction class extends RemoveStateNodeAbstractAction abstract class; it is used to remove the Pseudostate node with kind attribute equal to PseudostateKind.FORK.
- **RemoveJoinNodeAction**: RemoveJoinNodeAction class extends RemoveStateNodeAbstractAction abstract class; it is used to remove the Pseudostate node with kind attribute equal to PseudostateKind.JOIN.
- **RemoveDecisionNodeAction**: RemoveDecisionNodeAction class extends RemoveStateNodeAbstractAction abstract class; it is used to remove the Pseudostate node with kind attribute equal to PseudostateKind.CHOICE.
- **RemoveMergeNodeAction**: RemoveMergeNodeAction class extends RemoveStateNodeAbstractAction abstract class; it is used to remove the Pseudostate node with kind attribute equal to PseudostateKind.JUNCTION.
- **RemoveFlowFinalNodeAction**: RemoveFlowFinalNodeAction class extends RemoveStateNodeAbstractAction abstract class, it is used to remove the FinalState node with a tag definition attached (TagDefinition("FinalNodeType").value="FlowFinalNode").
- **RemoveFinalNodeAction**: RemoveFlowFinalNodeAction class extends RemoveStateNodeAbstractAction abstract class; it is used to remove the FinalState node.

- **RemoveTransitionAction**: RemoveTransitionAction class implements the Undoable interface and it aims to remove the transition object from the activity graph it attached to.

- **ActivityObjectSelection**: ActivityObjectSelection class implements the Transferable interface, and it is used to transfer the activity objects (which are serializable) to the clipboard. This class is used by all cut/copy actions in this design.
- **ActivityObjectClipboardUtility**: This utility class is used to copy specific activity element to the clipboard. It is used by all the cut/copy actions.
- **ActivityObjectCloneUtility**: ActivityObjectCloneUtility is a utility class; it provides method to clone the activity model object for the cut/copy actions.
- **ActivityDataFlavor**: ActivityDataFlavor class defines all the DataFlavor constants used in this design.

1.5 Component Exception Definitions

1.5.1 Custom Exceptions

- **ActivityObjectCloneException**: ActivityObjectCloneException class extends the ActionExecutionException class, and this exception is used if the clone operation of activity objects fails.
- **ActionExecutionException**: This exception is from the Action Manager component, all actions in this design will throw this exception in the execute methods.

1.5.2 System Exceptions

- **IllegalArgumentException**: Used wherever empty String argument is used while not acceptable. Normally an empty String is checked with trimmed result. It also thrown when null argument is passed in or some other cases when the argument is invalid.

1.6 Thread Safety

This component is not thread-safe, and there is no need to make it thread-safe, as each thread is expected to create its own action object to perform the task rather than share the same action object in different threads. And there is no such requirement too.

Though all actions are immutable, the internal state of their instance variable could be changed, so they are all not thread-safe.

2. Environment Requirements

2.1 Environment

Java 1.5

2.2 TopCoder Software Components

- **Action Manager 1.0**: The UndoableAction and TransientAction interfaces implemented by this design are from this component.
- **UML Model Manager 1.0**: This component is not used directly by this design. But it is referenced indirectly by UML Project Configuration component.
- **UML Project Configuration 1.0**: The ProjectConfigurationManager class used in this design is from this component.
- **UML Model - State Machines 1.0**: Classes from this component are used by this design.

- **UML Model - Activity Graphs 1.0**: Classes from this component are used by this design.
- **UML Model - Core Extension Mechanisms 1.0**: Classes from this component are used by this design.

NOTE: Configuration Manager is not used as all the configurable properties are passed into actions' constructors directly as arguments, which are more convenient as some arguments cannot be easily created from the configured properties (e.g. the ProjectConfigurationManager object, which is supposed to be shared in multiple actions), and it is also more flexible to set these arguments directly to constructor than load them from Configuration Manager.

2.3 Third Party Components

None.

3. Installation and Configuration

3.1 Package Name

com.topcoder.uml.actions.model.activity

3.2 Configuration Parameters

None.

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

None.

4.3 Demo

- `initialNode - Pseudostate object with its kind attribute equal to PseudostateKind.INITIAL`
- `objectFlowState - ObjectFlowState object`
- `actionState - ActionState object`
- `sendSignalActionNode - SimpleState object with a tag definition attached (TagDefinition("SendSignalAction").value="True")`
- `acceptEventActionNode – SimpleState object with a tag definition attached (TagDefinition("AcceptEventAction").value="True")`
- `forkNode - Pseudostate object with its kind attribute equal to PseudostateKind.FORK`
- `joinNode - Pseudostate object with its kind attribute equal to PseudostateKind.JOIN`
- `decisionNode - Pseudostate object with its kind attribute equal to PseudostateKind.CHOICE`
- `mergeNode - Pseudostate object with its kind attribute equal to PseudostateKind.JUNCTION`
- `flowFinalNode – FinalState object with a tag definition attached (TagDefinition("FinalNodeType").value="FlowFinalNode")`
- `finalNode – FinalState object`
- `transition – Transition object`

- clipboard – Clipboard object
- activityGraph - ActivityGraph object owning the elements above
- configManager - ProjectConfigurationManager object

Set up the environment

```
TestHelper.configProjectConfiguration();

CompositeState compositeState = new CompositeStateImpl();

initialNode = new PseudostateImpl();
initialNode.setKind(PseudostateKind.INITIAL);
initialNode.setContainer(compositeState);

objectFlowState = new ObjectFlowStateImpl();
objectFlowState.setContainer(compositeState);

actionState = new ActionStateImpl();
actionState.setContainer(compositeState);

sendSignalActionNode = new SimpleStateImpl();
sendSignalActionNode.setContainer(compositeState);

sendSignalActionNode.addTaggedValue(TestHelper.createTaggedValue("SendSignalActio
n", "True"));

acceptEventActionNode = new SimpleStateImpl();
acceptEventActionNode.setContainer(compositeState);

acceptEventActionNode.addTaggedValue(TestHelper.createTaggedValue("AcceptEventAct
ion", "True"));

forkNode = new PseudostateImpl();
forkNode.setContainer(compositeState);
forkNode.setKind(PseudostateKind.FORK);

joinNode = new PseudostateImpl();
joinNode.setContainer(compositeState);
joinNode.setKind(PseudostateKind.JOIN);

decisionNode = new PseudostateImpl();
decisionNode.setContainer(compositeState);
decisionNode.setKind(PseudostateKind.CHOICE);

mergeNode = new PseudostateImpl();
mergeNode.setContainer(compositeState);
mergeNode.setKind(PseudostateKind.JUNCTION);

flowFinalNode = new FinalStateImpl();
flowFinalNode.setContainer(compositeState);
flowFinalNode.addTaggedValue(TestHelper.createTaggedValue("FinalNodeType",
"FlowFinalNode"));

finalNode = new FinalStateImpl();
finalNode.setContainer(compositeState);

transition = new TransitionImpl();
StateMachine stateMachine = new StateMachineImpl();
transition.setStateMachine(stateMachine);

clipboard = new Clipboard("Test");

activityGraph = new ActivityGraphImpl();
activityGraph.setTop(compositeState);

manager = TestHelper.setUpModelManager();
```

4.3.1 Execute add actions

```
// add initialNode
```

```
AddInitialNodeAction action1 = new AddInitialNodeAction(initialNode,
 activityGraph, manager);
action1.execute();

// undo the action
action1.undo();

// redo the action
action1.redo();

// add objectFlowState
AddObjectFlowStateAction action2 = new AddObjectFlowStateAction(objectFlowState,
 activityGraph, manager);
action2.execute();

// undo the action
action2.undo();

// redo the action
action2.redo();

// add actionState
AddActionStateAction action3 = new AddActionStateAction(actionState,
 activityGraph, manager);
action3.execute();

// undo the action
action3.undo();

// redo the action
action3.redo();

// add sendSignalActionNode
AddSendSignalActionAction action4 = new
 AddSendSignalActionAction(sendSignalActionNode, activityGraph, manager);
action4.execute();

// undo the action
action4.undo();

// redo the action
action4.redo();

// add acceptEventActionNode
AddAcceptEventActionAction action5 = new
 AddAcceptEventActionAction(acceptEventActionNode, activityGraph, manager);
action5.execute();

// undo the action
action5.undo();

// redo the action
action5.redo();

// add forkNode
AddForkNodeAction action6 = new AddForkNodeAction(forkNode, activityGraph,
 manager);
action6.execute();

// undo the action
action6.undo();

// redo the action
action6.redo();

// add joinNode
AddJoinNodeAction action7 = new AddJoinNodeAction(joinNode, activityGraph,
 manager);
action7.execute();

// undo the action
```

```
action7.undo();

// redo the action
action7.redo();

// add decisionNode
AddDecisionNodeAction action8 = new AddDecisionNodeAction(decisionNode,
activityGraph, manager);
action8.execute();

// undo the action
action8.undo();

// redo the action
action8.redo();

// add mergeNode
AddMergeNodeAction action9 = new AddMergeNodeAction(mergeNode, activityGraph,
manager);
action9.execute();

// undo the action
action9.undo();

// redo the action
action9.redo();

// add flowFinalNode
AddFlowFinalNodeAction action10 = new AddFlowFinalNodeAction(flowFinalNode,
activityGraph, manager);
action10.execute();

// undo the action
action10.undo();

// redo the action
action10.redo();

// add finalNode
AddFinalNodeAction action11 = new AddFinalNodeAction(finalNode, activityGraph,
manager);
action11.execute();

// undo the action
action11.undo();

// redo the action
action11.redo();

// add transition
AddTransitionAction action12 = new AddTransitionAction(transition, activityGraph,
manager);
action12.execute();

// undo the action
action12.undo();

// redo the action
       action12.redo();
```

4.3.2 *Execute remove actions*

```
// remove initialNode
RemoveInitialNodeAction action1 = new RemoveInitialNodeAction(initialNode);
action1.execute();

// undo the action
action1.undo();

// redo the action
action1.redo();
```

```
// remove objectFlowState
RemoveObjectFlowStateAction action2 = new
 RemoveObjectFlowStateAction(objectFlowState);
action2.execute();

// undo the action
action2.undo();

// redo the action
action2.redo();

// remove actionState
RemoveActionStateAction action3 = new RemoveActionStateAction(actionState);
action3.execute();

// undo the action
action3.undo();

// redo the action
action3.redo();

// remove sendSignalActionNode
RemoveSendSignalActionAction action4 = new
 RemoveSendSignalActionAction(sendSignalActionNode);
action4.execute();

// undo the action
action4.undo();

// redo the action
action4.redo();

// remove acceptEventActionNode
RemoveAcceptEventActionAction action5 = new
 RemoveAcceptEventActionAction(acceptEventActionNode);
action5.execute();

// undo the action
action5.undo();

// redo the action
action5.redo();

// remove forkNode
RemoveForkNodeAction action6 = new RemoveForkNodeAction(forkNode);
action6.execute();

// undo the action
action6.undo();

// redo the action
action6.redo();

// remove joinNode
RemoveJoinNodeAction action7 = new RemoveJoinNodeAction(joinNode);
action7.execute();

// undo the action
action7.undo();

// redo the action
action7.redo();

// remove decisionNode
RemoveDecisionNodeAction action8 = new RemoveDecisionNodeAction(decisionNode);
action8.execute();

// undo the action
action8.undo();
```

```
// redo the action
action8.redo();

// remove mergeNode
RemoveMergeNodeAction action9 = new RemoveMergeNodeAction(mergeNode);
action9.execute();

// undo the action
action9.undo();

// redo the action
action9.redo();

// remove flowFinalNode
RemoveFlowFinalNodeAction action10 = new RemoveFlowFinalNodeAction(flowFinalNode);
action10.execute();

// undo the action
action10.undo();

// redo the action
action10.redo();

// remove finalNode
RemoveFinalNodeAction action11 = new RemoveFinalNodeAction(finalNode);
action11.execute();

// undo the action
action11.undo();

// redo the action
action11.redo();

// remove transition
RemoveTransitionAction action12 = new RemoveTransitionAction(transition);
action12.execute();

// undo the action
action12.undo();

// redo the action
action12.redo();
```

4.3.3 *Execute cut actions*

```
// cut initialNode
CutInitialNodeAction action1 = new CutInitialNodeAction(initialNode, clipboard);
action1.execute();

// undo the action
action1.undo();

// redo the action
action1.redo();

// cut objectFlowState
CutObjectFlowStateAction action2 = new CutObjectFlowStateAction(objectFlowState,
clipboard);
action2.execute();

// undo the action
action2.undo();

// redo the action
action2.redo();

// cut actionState
```

```
CutActionStateAction action3 = new CutActionStateAction(actionState, clipboard);
action3.execute();

// undo the action
action3.undo();

// redo the action
action3.redo();

// cut sendSignalActionNode
CutSendSignalActionAction action4 = new
 CutSendSignalActionAction(sendSignalActionNode, clipboard);
action4.execute();

// undo the action
action4.undo();

// redo the action
action4.redo();

// cut acceptEventActionNode
CutAcceptEventActionAction action5 = new
 CutAcceptEventActionAction(acceptEventActionNode, clipboard);
action5.execute();

// undo the action
action5.undo();

// redo the action
action5.redo();

// cut forkNode
CutForkNodeAction action6 = new CutForkNodeAction(forkNode, clipboard);
action6.execute();

// undo the action
action6.undo();

// redo the action
action6.redo();

// cut joinNode
CutJoinNodeAction action7 = new CutJoinNodeAction(joinNode, clipboard);
action7.execute();

// undo the action
action7.undo();

// redo the action
action7.redo();

// cut decisionNode
CutDecisionNodeAction action8 = new CutDecisionNodeAction(decisionNode,
 clipboard);
action8.execute();

// undo the action
action8.undo();

// redo the action
action8.redo();

// cut mergeNode
CutMergeNodeAction action9 = new CutMergeNodeAction(mergeNode, clipboard);
action9.execute();

// undo the action
action9.undo();

// redo the action
action9.redo();
```

```
// cut flowFinalNode
CutFlowFinalNodeAction action10 = new CutFlowFinalNodeAction(flowFinalNode,
clipboard);
action10.execute();

// undo the action
action10.undo();

// redo the action
action10.redo();

// cut finalNode
CutFinalNodeAction action11 = new CutFinalNodeAction(finalNode, clipboard);
action11.execute();

// undo the action
action11.undo();

// redo the action
action11.redo();

// cut transition
CutTransitionAction action12 = new CutTransitionAction(transition, clipboard);
action12.execute();

// undo the action
action12.undo();

// redo the action
action12.redo();

// if we want to use system clipboard, just pass in null clipboard
action12 = new CutTransitionAction(transition, null);
action12.execute();
```

4.3.4 *Execute copy actions*

```
// copy initialNode
CopyInitialNodeAction action1 = new CopyInitialNodeAction(initialNode, clipboard);
action1.execute();

// copy objectFlowState
CopyObjectFlowStateAction action2 = new CopyObjectFlowStateAction(objectFlowState,
clipboard);
action2.execute();

// copy actionState
CopyActionStateAction action3 = new CopyActionStateAction(actionState, clipboard);
action3.execute();

// copy sendSignalActionNode
CopySendSignalActionAction action4 = new
CopySendSignalActionAction(sendSignalActionNode, clipboard);
action4.execute();

// copy acceptEventActionNode
CopyAcceptEventActionAction action5 = new
CopyAcceptEventActionAction(acceptEventActionNode, clipboard);
action5.execute();

// copy forkNode
CopyForkNodeAction action6 = new CopyForkNodeAction(forkNode, clipboard);
action6.execute();

// copy joinNode
CopyJoinNodeAction action7 = new CopyJoinNodeAction(joinNode, clipboard);
```

```
action7.execute();

// copy decisionNode
CopyDecisionNodeAction action8 = new CopyDecisionNodeAction(decisionNode,
clipboard);
action8.execute();

// copy mergeNode
CopyMergeNodeAction action9 = new CopyMergeNodeAction(mergeNode, clipboard);
action9.execute();

// copy flowFinalNode
CopyFlowFinalNodeAction action10 = new CopyFlowFinalNodeAction(flowFinalNode,
 clipboard);
action10.execute();

// copy finalNode
CopyFinalNodeAction action11 = new CopyFinalNodeAction(finalNode, clipboard);
action11.execute();

// copy transition
CopyTransitionAction action12 = new CopyTransitionAction(transition, clipboard);
action12.execute();

// if we want to use system clipboard, just pass in null clipboard
action12 = new CopyTransitionAction(transition, null);
action12.execute();
```

4.3.5 *Execute paste actions*

```
// paste initialNode
PasteInitialNodeAction action1 = new PasteInitialNodeAction(initialNode,
 activityGraph);
action1.execute();

// undo the action
action1.undo();

// redo the action
action1.redo();

// paste objectFlowState
PasteObjectFlowStateAction action2 = new
 PasteObjectFlowStateAction(objectFlowState, activityGraph);
action2.execute();

// undo the action
action2.undo();

// redo the action
action2.redo();

// paste actionState
PasteActionStateAction action3 = new PasteActionStateAction(actionState,
 activityGraph);
action3.execute();

// undo the action
action3.undo();

// redo the action
action3.redo();

// paste sendSignalActionNode
PasteSendSignalActionAction action4 = new
 PasteSendSignalActionAction(sendSignalActionNode, activityGraph);
action4.execute();

// undo the action
action4.undo();
```

```
// redo the action
action4.redo();

// paste acceptEventActionNode
PasteAcceptEventActionAction action5 = new
 PasteAcceptEventActionAction(acceptEventActionNode, activityGraph);
action5.execute();

// undo the action
action5.undo();

// redo the action
action5.redo();

// paste forkNode
PasteForkNodeAction action6 = new PasteForkNodeAction(forkNode, activityGraph);
action6.execute();

// undo the action
action6.undo();

// redo the action
action6.redo();

// paste joinNode
PasteJoinNodeAction action7 = new PasteJoinNodeAction(joinNode, activityGraph);
action7.execute();

// undo the action
action7.undo();

// redo the action
action7.redo();

// paste decisionNode
PasteDecisionNodeAction action8 = new PasteDecisionNodeAction(decisionNode,
 activityGraph);
action8.execute();

// undo the action
action8.undo();

// redo the action
action8.redo();

// paste mergeNode
PasteMergeNodeAction action9 = new PasteMergeNodeAction(mergeNode, activityGraph);
action9.execute();

// undo the action
action9.undo();

// redo the action
action9.redo();

// paste flowFinalNode
PasteFlowFinalNodeAction action10 = new PasteFlowFinalNodeAction(flowFinalNode,
 activityGraph);
action10.execute();

// undo the action
action10.undo();

// redo the action
action10.redo();

// paste finalNode
PasteFinalNodeAction action11 = new PasteFinalNodeAction(finalNode,
 activityGraph);
action11.execute();
```

```
// undo the action
action11.undo();

// redo the action
action11.redo();

// paste transition
PasteTransitionAction action12 = new PasteTransitionAction(transition,
 activityGraph);
action12.execute();

// undo the action
action12.undo();

// redo the action
action12.redo();
```

4.3.6 *Select the paste action by the application*

Paste actions are expected to be selected by the application using this component.

```
// copy initialNode
CopyInitialNodeAction action = new CopyInitialNodeAction(initialNode, clipboard);
action.execute();

// an initialNode has been copied to the clipboard,
// the code below shows how to call the correct paste action.
Transferable contents = clipboard.getContents(null);
if (contents.isDataFlavorSupported(ActivityDataFlavor.INITIAL_NODE)) {
    // get the pasted object
    Object data = contents.getTransferData(ActivityDataFlavor.INITIAL_NODE);
    PasteInitialNodeAction action1 = new PasteInitialNodeAction(data,
    activityGraph);

    // execute the action to add data into activityGraph
    action1.execute();
}

// other types of actions are just the same and so it omitted here.
```

5 Future Enhancements

None.