Using Choice and Junction Points in DevOps Model RealTime vs RoseRT

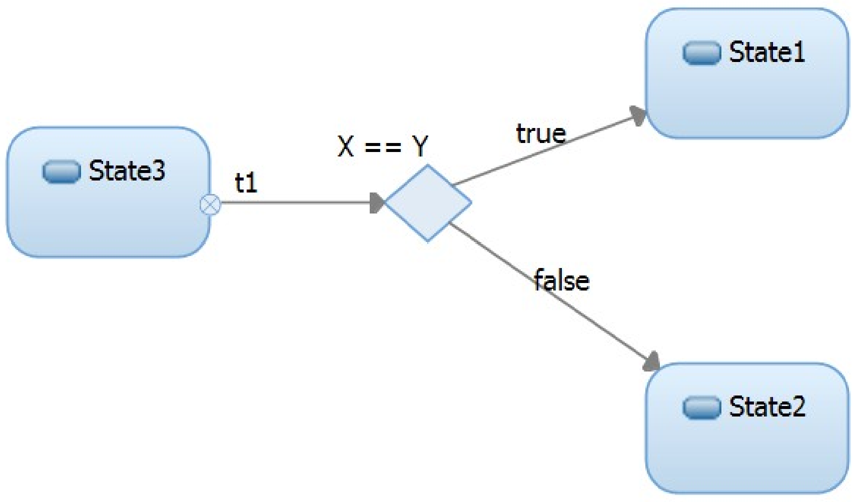
Author: Mattias Mohlin
IBM

This document discusses how choice and junction points can be used in DevOps Model Real-Time, and how this is different from using these concepts in RoseRT. Included are examples and certain pitfalls to avoid.

The document was last updated for Model RealTime 10.2. All screen shots were captured on the Windows platform.

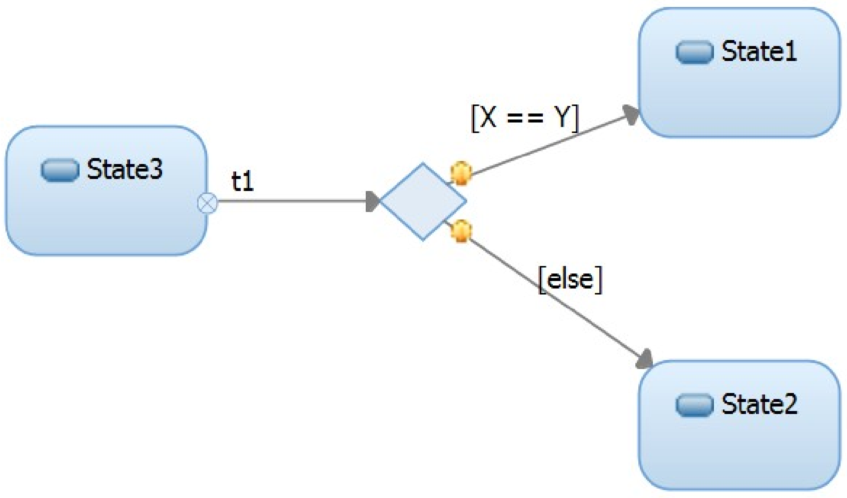
Choice Point Differences from RoseRT

In RoseRT, conditional guards are attached to a choice point with true and false paths as exit transitions. Consider the example below:



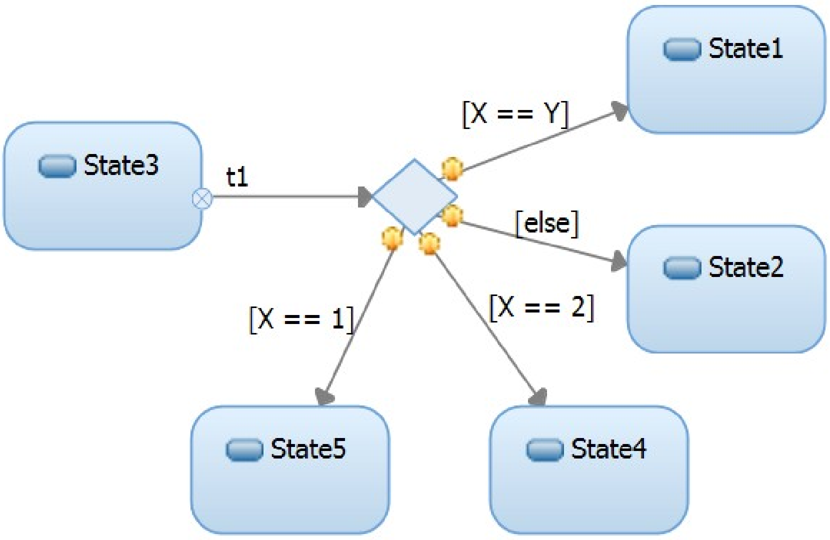
When trigger t1 fires, if X equals Y then the state machine transitions to State1, otherwise to State2. A limitation with this approach is that there can only be two exit paths from each choice point.

In Model RealTime, which is based on UML 2, conditional guards are attached to the transitions rather than to the choice point. The figure below shows the Model RealTime equivalent of the above state chart:



When the trigger t1 has fired and executed its effect, then the guards of the transitions that leave the choice point are evaluated. If X equals Y then the transition is taken to State1, otherwise the transition to State2 is taken. This is hence the same behavior as in the previous RoseRT state chart.

The advantage of choice points in Model RealTime compared to RoseRT is that you can have more than two output transitions. Let's look at an example of this:



You can have as many outgoing transitions as you wish, and adding them may seem straight-forward. But, you have to be careful! What is wrong with the above example? It may look OK at a first glance:

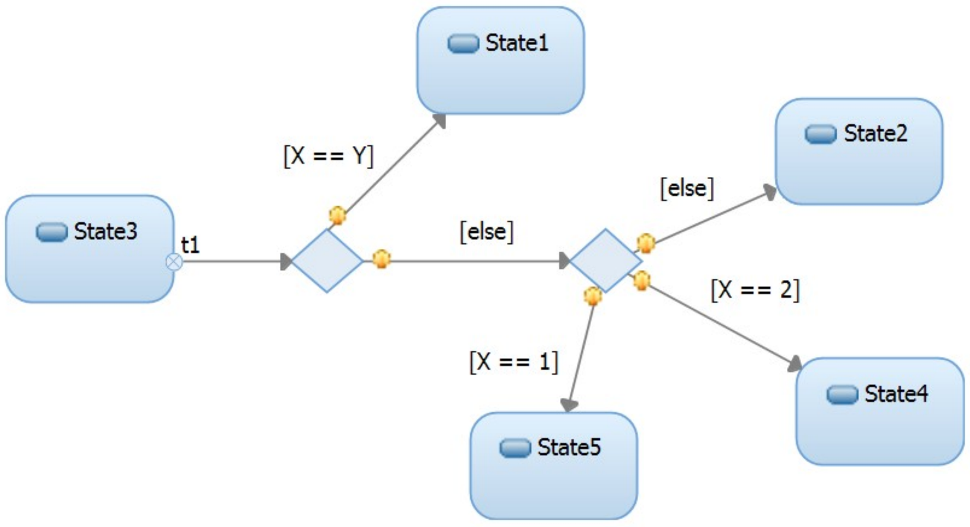
```
if X = 1 then go to State5
if X = 2 then go to State4
if X = Y then go to State1
else go to State2
```

But what happens if X = Y = 2? Will we then go to State1 or to State4?

The answer is…..we don't really know. UML does not define which transition to take if more than one has a fulfilled guard condition. In practice one of the enabled transitions will of course execute (probably the one that was drawn first), but it is not recommended to let your application logic depend on that. So, the rule is to avoid "overlapping guards" such as what we have above, and make sure that all guard conditions on transitions that leave the same choice point are mutually exclusive.

To fix this problem we first need to decide the order that we want the overlapping transition guards to be checked. Should we first check if X == Y or is it better to first check if X == 1 or X == 2? This is a design decision, but let's assume that in this case we want to first check if X == Y.

The improved design is shown below:



Now, we first check if X == Y before we check the other values of X. If X = Y = 2, then we know that we will always transition to State1, because we first check that X = Y regardless of the value of Y.

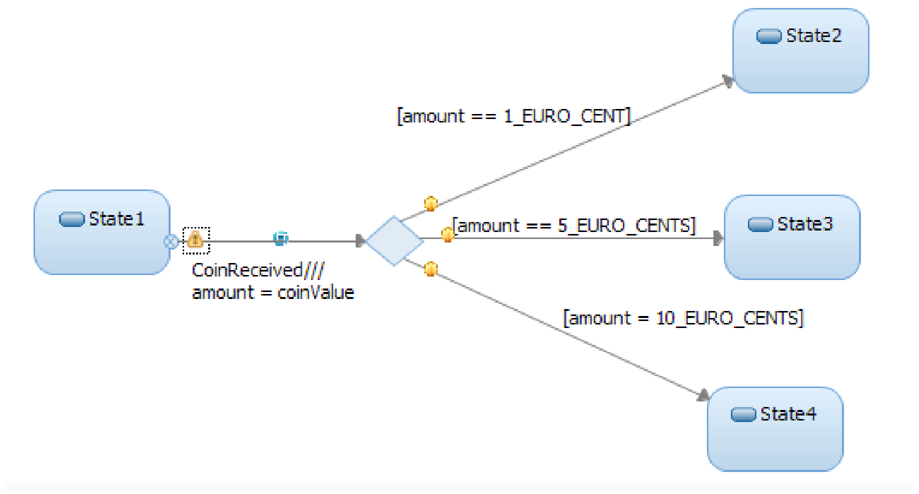
Choice points vs junction points

As we have seen above choice points are used for branching the execution in a state chart based on the dynamic evaluation of guard conditions on transitions. Model RealTime also supports another kind of branching, where guard conditions are instead statically evaluated. This kind of branching uses junction points. The table below describes the differences between junction points and choice points:

Symbol	**Name**	**Branching**	**Description**
●	Junction Point	Static	All outgoing transition guards for all possible paths are evaluated when the first trigger is fired and before executing any action code. This means that no action code in the first transition or any subsequent transitions in the chain of enabled transitions will affect the evaluation of the guards.
◇	Choice Point	Dynamic	The outgoing transition guards for a choice point are evaluated at the time when that choice point is reached during execution. This means that action code of the first transition and previous transitions in the chain of enabled transitions can affect the behavior of subsequent choice points.

Hence, if the branching conditions do not depend on any computation made in the action code that runs when a trigger fires, then you can use junction points. Otherwise, you must use choice points.

Let's look at an example:

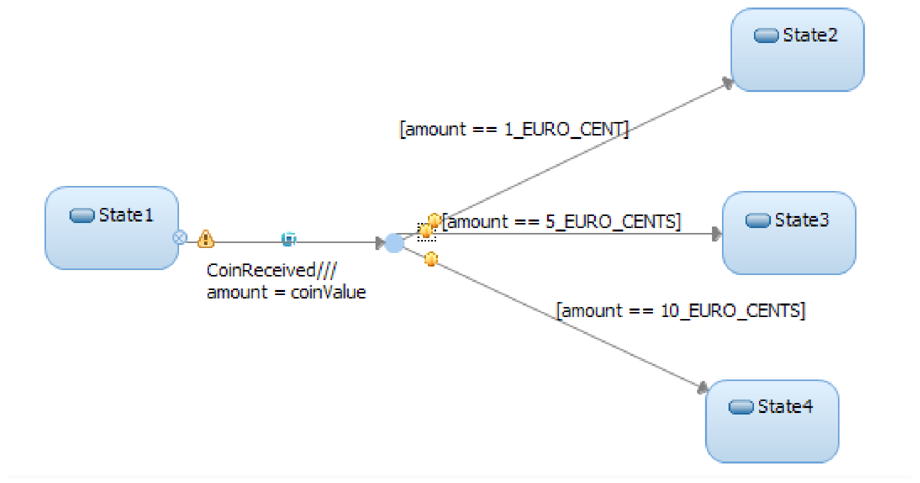


This state chart describes a vending machine. When the CoinReceived trigger is fired we want to go to some state based on the value of the coin that was entered.
The order in which things happen is:

1. Trigger CoinReceived fires
2. The effect action code executes: `amount = coinValue`
3. Guard conditions of the transitions that go out from the choice point are evaluated

So, after the trigger fires, we update the "amount" variable with the value of the coin and then proceed to the choice point. The outgoing choice point transitions are then evaluated and we proceed to the appropriate state based on the value of the coin that was just placed into the machine.

Now, what would happen if we instead would use a junction point in this example?



Assume that the coin placed in the machine is 5 euro cents. What state will we transition to? Is it State3?

The answer is…..we don't really know.

In this case the following happens:

1. Trigger CoinReceived fires
2. The guards on the transitions that go out from the junction point are evaluated, using the **current** (not yet updated) value of "amount"
3. If an enabled transition is found (so that there is a valid path to a state)
   a. Run the action code in the CoinReceived trigger ("amount" will be assigned 5_EURO_CENTS)
   b. Proceed along the transition path to the new state based on what was decided in step 2, executing the effect action code of this transition (not present in this example)
4. If no valid path to a state is found (i.e. all guards evaluate to false)
   a. Do nothing and stay in State1

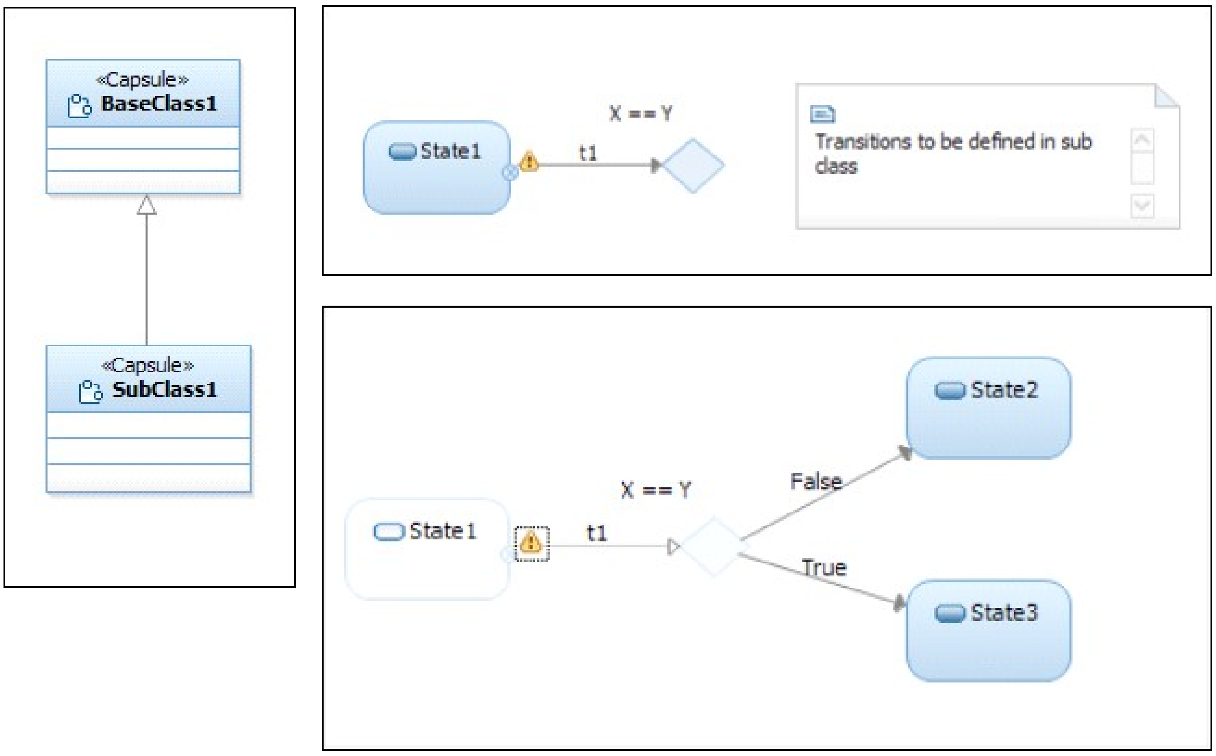
As you see, the transition guards will be evaluated on the **previous** value of "amount" since the guards are evaluated before the action code has a chance to update the "amount" variable with the correct value. We don't know what the new state will be; this depends on the value of "amount" at the time when CoinReceived fires.

To avoid this pitfall, make sure that when you use junction points that you do not try to change, in actions, how outgoing guards will be evaluated. Doing so can result in bugs that are hard to find.

How choice points are imported from RoseRT

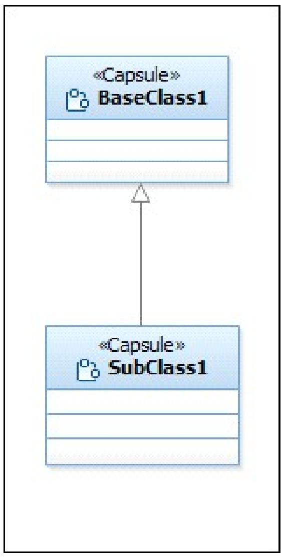
Since the behavior of choice points is different in Model RealTime compare to RoseRT, there can be a difference in how choice points are implemented when a model is imported from RoseRT to Model RealTime. To recap: in RoseRT guards are connected to the choice point, whereas in Model RealTime, guards are connected to the outgoing transitions from the choice points.

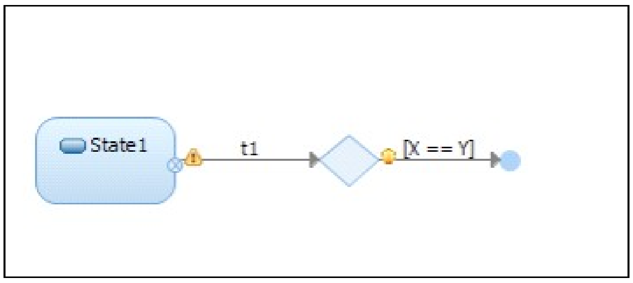
Consider the case where we have a base capsule BaseClass1 and a sub capsule SubClass1 in RoseRT. In RoseRT we are allowed to define the conditional choice point with no outgoing transitions in the base class and then define the transitions down in the sub class. The figure below shows a representation of how this would be in RoseRT. (Note that this example was actually drawn in Model RealTime but it represents how it would work in RoseRT).

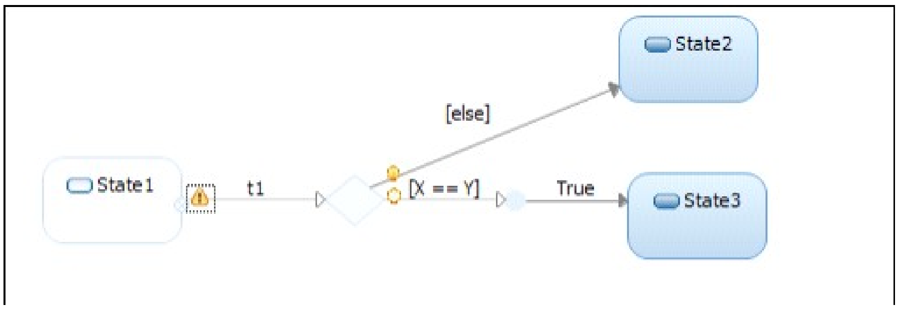


Our branching condition X == Y is defined in the state chart of BaseClass1, and then we add the outgoing transitions in SubClass1.

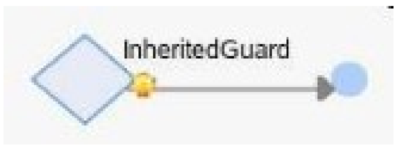
We now want to import this model into Model RealTime. The problem is the X==Y condition in the BaseClass1 that is attached to the choice point. In Model RealTime we cannot associate the condition with the choice point itself so we need to store this in a transition since guard conditions are associated with transitions in Model RealTime.







So, the pattern for importing choice points from RoseRT to Model RealTime is as shown below. A transition to a junction point is added to hold the choice point condition.


InheritedGuard