IBM Software Group

Rational. software

Cheatsheet

The following is a quick cheatsheet of common issues seen during my visit with RBS during October 18 – November 5, 2010.

Written by: Adam Neal

RSARTE CheatSheet

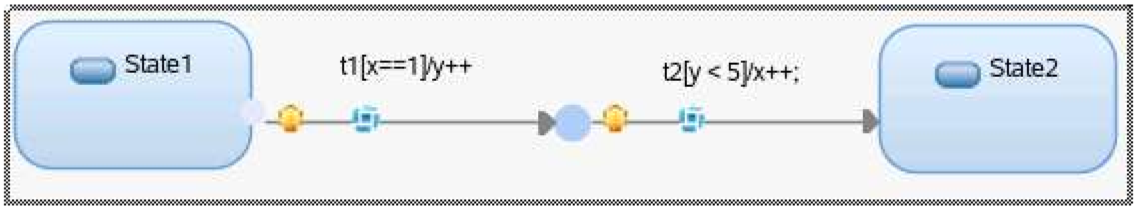
1) Views are detachable – You can make Views free floating by simplying right clicking on the View's tab and select “Detached” (this is great for the code view). Alternatively, make them 'Fast Views' to save space, and just their icons will show around the border of the workbench, waiting for you to click them in order to show the view again.
2) There are two types of refresh to be aware of.
   a) Refresh projects in the PE after making any changes outside the tool (e.g. Undo checkout, or file manipulation via OS terminal). This insures that the model you're looking at is up-to-date.
   b) Use Refresh in the 'Team' submenu to refresh the Clear case Checkout status of elements
3) Want less rigid control over placement of shapes on the diagram? Right click on the diagram (not within the Machine frame, but outside it if there is one) and use the “View” sub menu in order to adjust properties like “Snap to Grid”, and showing the “Grid” or “Ruler”.
4) Use the Link with Editor button in the PE ⇆ to keep the PE synchronized with the editor you`re viewing.
5) Backspace is equivilent to 'left' on windows (navigate to parent in PE)
6) SHIFT+Right/Left is expand/collapse in the PE
7) Guards that use to be associated with a choice point, is now a guard on an outgoing transition. Selecting a choice point will enable the Code View to show its outgoing transitions. So you need to be aware of which outgoing transitions you want to edit the guard of. Alternatively just select the transition to edit its guard. Note: choice points and junction points can have multiple outgoing transitions, just use mutually exclusive guards.
8) Did you know, that internal transitions vs local self transitions have different semantics?
   From the UML Spec:
   • kind=internal implies that the transition, if triggered, occurs without exiting or entering the source state. Thus, it does not cause a state change. This means that the entry or exit condition of the source state will not be invoked.
   • kind=local implies that the transition, if triggered, will not exit the composite (source) state, but it will apply to any state within the composite state, and these will be exited and entered.
   • kind=external implies that the transition, if triggered, will exit the source vertex.

Thus, if you have a self transition, and you know that nested states should not have their entry/exit code invoked then make its 'kind' internal. Some teams have design rules that no entry/exit should ever be used. In this case, using internal transitions over 'local' self transitions will speed up the runtime performance slightly as well, since it removes the runtime checks for entry and exit code.

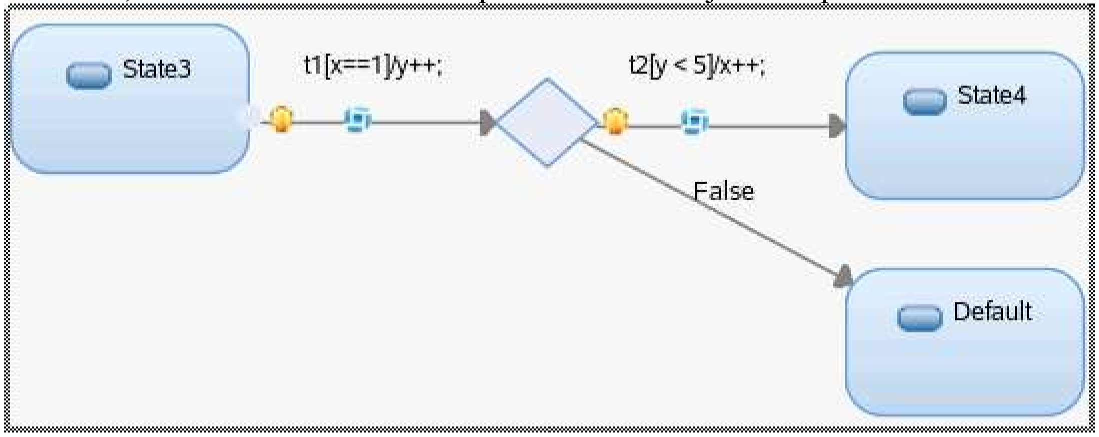
9) When fixing compilation errors, double click the problem associated with the .cpp file rather than the emx/efx file. Then make your changes directly in the Code Editor. When you save, you'll be prompted to verify the changes you made.
Besure not to touch the //{{{USR tags though.

10) Junction Point semantics:
Did you know that UML2 has junction points available for state chart modelling? Junction points are static conditional branches, while choice points are dynamic conditional branches.
In short, static conditional branches mean that all the guards of transitions chained by junction points (terminating at a choicepoint or a state) will be evaluated prior to taking the first transition. Only if an enabled path exists, will the transition be fired, and their effects be executed.
For example, consider the case where we have:

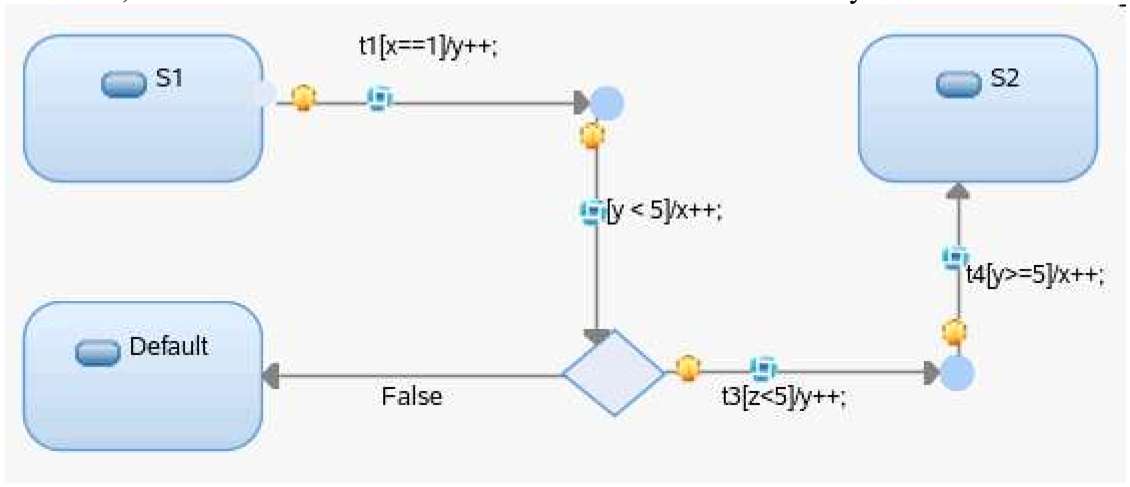
RSARTE CheatSheet



When the event for *t1* arrives, *t1*'s guard && *t2*'s guard must evaluate to true prior to firing *t1* and executing the effect of t1 followed by the effect t2. This means if we start with *x=1 and y=5*, *t1* will never fire. But if we have *x=1 && y=4*, t1 will fire, and when we arrive in *State2*, *y* will equal 5 and *x* will equal 2.
Where as, if the above used a choice point instead of a junction point:

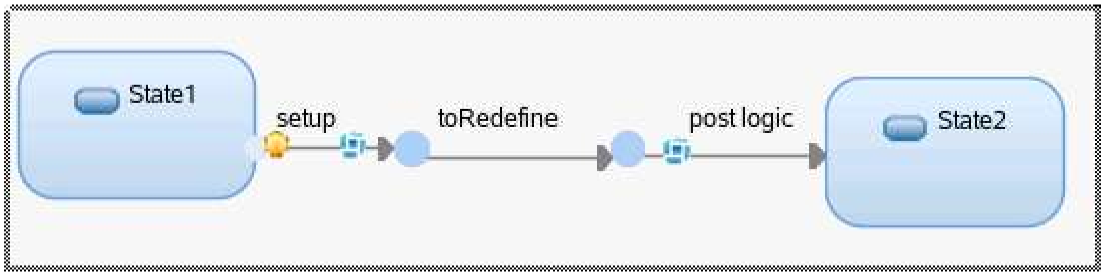


then only *t1*'s guard (*x==1*) would need to evaluate to true before firing *t1* and executing its effect. When the behavior arrives at the choice point, it will then evaluate the outgoing transition guards to determine the path to take, using the unguarded (or '*else*') branch if no guarded path evaluates to *true*. Note that junction points can be used before and/or after a choice point as well. All guards for transitions compounded by junction points must evaluate to *true* in order for that path to be considered enabled. For example in the below case, *x==1 && y < 5* must be true in order for *t1* to fire. Once fired, *y* and then *x* will increment. Then the behavior reaches the choice point and *z<5 && y>=5* must evaluate to *true* in order to transition to *S2* (note: *t3*'s effect will not be executed prior to the evaluation), if both conditions are not *true* then the system will transition to the '*Default*' state, otherwise, *t3*'s effect and then *t4*'s effect will be executed and the system will transition to *S2*.

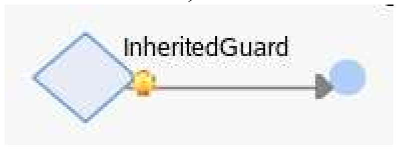


One benifit to using junction points is the facilitation of code reuse in subclasses. For example:

RSARTE CheatSheet



One can setup some pre and post conditions/logic while allowing the main work to be written in specialized capsules. Using this type of pattern, only the 'toRedefine' transition needs to be redefined to have an effect and/or guard, the rest of the logic will be inherited and reused. This is the same pattern that RSARTE uses to facilitate the RoseRT case of a guard condition defined on a choice point in a base class, but the transition should not be defined until the sub capsule.



From the UML spec:

• *junction* vertices are semantic-free vertices that are used to chain together multiple transitions. They are used to construct compound transition paths between states. For example, a junction can be used to converge multiple incoming transitions into a single outgoing transition representing a shared transition path (this is known as a *merge*). Conversely, they can be used to split an incoming transition into multiple outgoing transition segments with different guard conditions. This realizes a *static conditional branch*. (In the latter case, outgoing transitions whose guard conditions evaluate to false are disabled. A predefined guard denoted “else” may be defined for at most one outgoing transition. This transition is enabled if all the guards labeling the other transitions are false.) Static conditional branches are distinct from dynamic conditional branches that are realized by choice vertices (described below).

• *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. If more than one of the guards evaluates to true, an arbitrary one is selected. If none of the guards evaluates to true, then the model is considered ill-formed. (To avoid this, it is recommended to define one outgoing transition with the predefined “else” guard for every choice vertex.) Choice vertices should be distinguished from static branch points that are based on junction points (described above).