

## Speed Machine Tutorial in Rational Rose Real Time

The objective of this tutorial is to highlight basic aspects of software modeling. Throughout the Tutorial, the student will realize that although there are many ways to make things work, some of them are preferred to the others. The strategy builds on good modeling techniques that will not only enable us to create robust software, but also prepare it for later maintenance.

Within the context of Model Driven Development, we intend to emphasize the benefit of good modeling techniques using a tool (IBM Rational Rose RealTime) that will automatically generate executable code upon the models we write.

This tutorial requires a basic knowledge of RoseRT (i.e. the student must have completed successfully the LightSystem Tutorial) before proceeding.

To run IBM Rational Rose RealTime (RoseRT) and create a new model, download and execute the file **rosert.bat** from the course web page. Select **RTC++ framework**.

### Speed Machine behavior

In this tutorial, the interaction between a user and a machine is to be modeled. The user can turn a machine **on** and **off**. When the machine is first turned **on**, the initial speed of the machine is set to 1. The user then can **increase** the speed sequentially (1 to 2, 2 to 3 and 3 to 4), or **decrease** the speed (4 to 3, 3 to 2 and 2 to 1) as well. Finally the user can turn the machine **off** but only if the machine is set to speed 1.

To model this behavior we will have four approaches.


#### I. FIRST APPROACH. Single state, single transition:

Consider that we have two interacting capsules: a user and a machine. For the machine capsule, a unique state will be created, with a unique transition. Variables will be needed to store the **on** and **off** state and the **speed**.

##### 1. Save your model.

Save your model as **SpeedMachine1.rtmidl**, remember to do this **very often**.

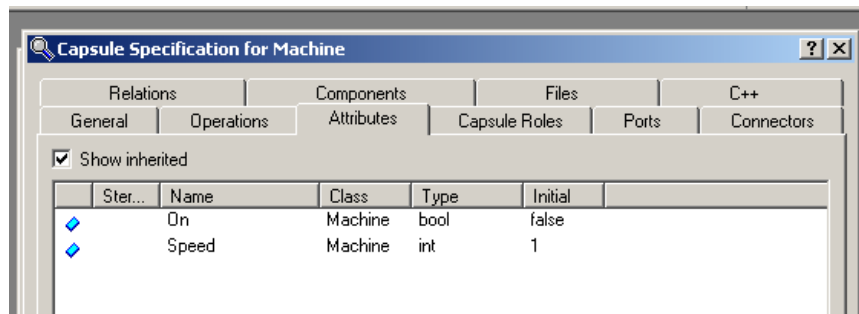
##### 2. Creating a capsule

- Open and expand the Logical View in the Model Browser.
- Open Main Class diagram (double click on it).
- Select the capsule tool and left-click on the window. 
- Rename the capsule name to **User**.
- Create two more capsules with the name **Machine** and **Top**.
- The **Machine** capsule will have two attributes: **On** and **Speed**.
- To **add attributes** in a capsule, double click on the capsule to open the **Capsule Specification**, in the Attributes section right-click and select Insert. Rename NewAttribute1 to **On**. Double-click on it and select Visibility Public in

General Section. In the detail section select **bool** type with initial value **false**.  
(**On** attribute will simulate on and off).


- Create a Speed attribute, type **int** with initial value **1**.

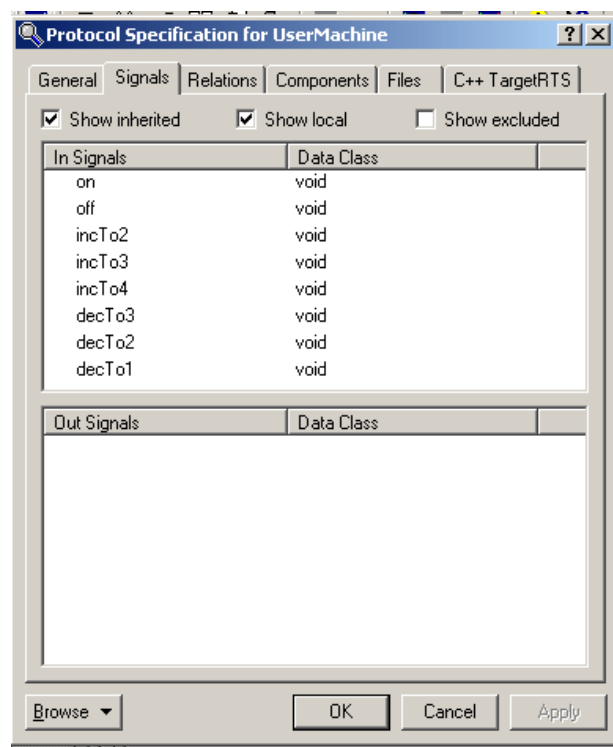
Your capsule specification should look like this:



Once capsules are created, we need to define a protocols to communicate the **User** and **Machine** capsules.

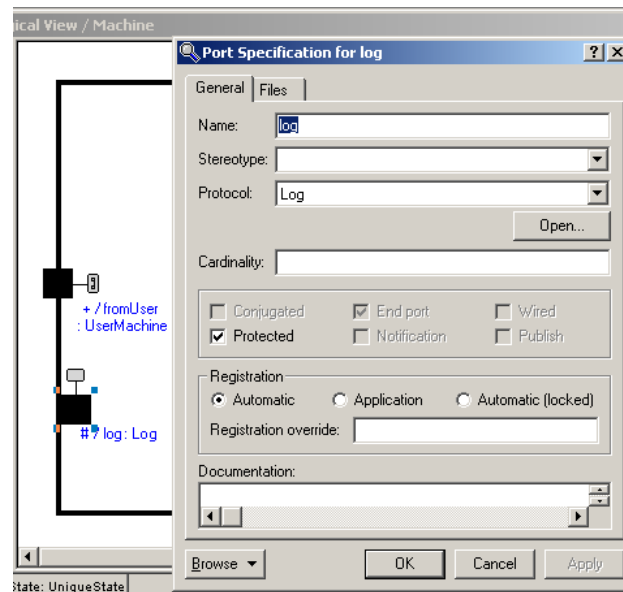
### 3. Creating a protocol

- Select the protocol tool and drag it into the main class diagram. 
- Rename protocol to **UserMachine**.
- To **create the signals of a protocol**, double click on it, open the signal section, right-click on it and select Insert. insert the following **In signals**:



#### 4. Add ports to a capsule.

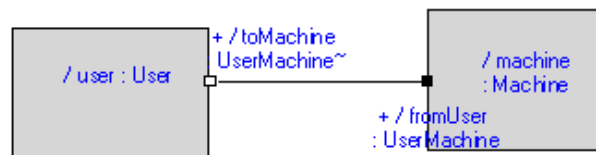
- Expand the **Machine** capsule menu and open the structure diagram.
- Drag a UserMachine protocol into the boundary of the **Machine** capsule. Rename the port name to **fromUser**. Right-click on it and select the property **EndPort**.
- Select the port tool and drag it into the boundary again. Select the Log protocol. Rename it to log. (Log protocol is used to display data in screen). Right-click on the log port and select open specification. Check the protected attribute.



In the User capsule, create a port named **toMachine** with the **UserMachine** protocol, select property **conjugate** (to reverse the direction of the signals).

#### 5. Top Capsule

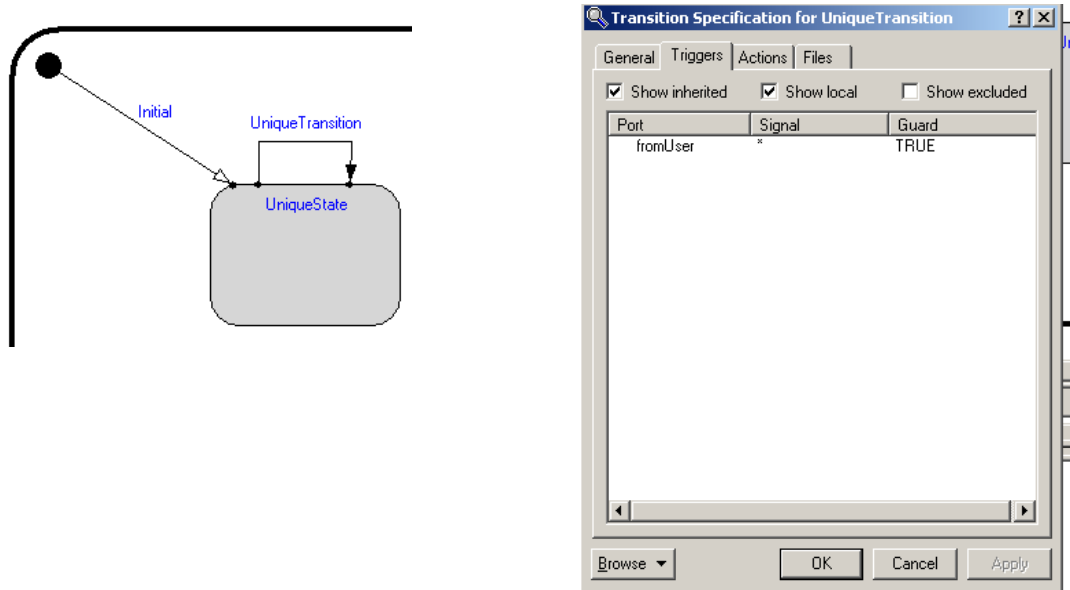
- Open the Structure Diagram of the Top capsule.
- Drag the User and Machine capsules.
- Connect the port **toMachine** of the **User** to the port **fromUser** of the **Machine**. Your diagram should look like this:



#### 6. Create states of a capsule

To add behavior into the capsules, we need to create states.

- Open the **Machine** State diagram.
- With the state tool, create a new state, name it **UniqueState**.
- Create an initial transition to it.
- Create a transition to itself and name it **UniqueTransition**.
- Double click on the **UniqueTransition**, double-click on the triggers section, insert all the signals from the port **fromUser** selecting **\***.



- In the action section, copy and paste the following code (to simulate the machine):

```

if (On==false) {
if ((getMsg()->getSignalName()=="on"){
    On=true;
    log.log("Machine turned on");
} //end if (on received)
else{
    log.log("Signal cannot be processed, machine is off ");
} //end else if (on received)
} //end if (on==false)
else{
    if ((getMsg()->getSignalName()=="off"){
        On=false;
        Speed=1;
        log.log("Machine turned off ");
    } //end if (off received)
    else{
        switch(Speed)
        {case 1:

            if ((getMsg()->getSignalName()=="incTo2"){
                Speed++;
                log.log("Speed incremented to 2 ");
            } //end if (incTo2 received)
            else
                log.log("Signal cannot be processed, speed is 1 ");
                break;

            case 2:if ((getMsg()->getSignalName()=="incTo3"){

```

```
        Speed++;
        log.log("Speed incremented to 3 ");
    }//end if (incTo3 received)
    else{
        if ((getMsg()->getSignalName()=="decTo1"){
            Speed--;
            log.log("Speed decremented to 1 ");
        }//end if (decTo1 received)
        else
            log.log("Signal cannot be processed, speed is 2 ");
    }//else (incTo3 received)
    break;
case 3:if ((getMsg()->getSignalName()=="incTo4"){
        Speed++;
        log.log("Speed incremented to 4 ");
    }//end if (incTo4 received)
    else{
        if ((getMsg()->getSignalName()=="decTo2"){
            Speed--;
            log.log("Speed decremented to 2 ");
        }//end if (decTo2 received)
        else
            log.log("Signal cannot be processed, speed is 3 ");
    }//else (incTo3 received)
    break;
case 4:if ((getMsg()->getSignalName()=="decTo3"){
        Speed--;
        log.log("Speed decremented to 3 ");
    }//end if (decTo3 received)
    else
        log.log("Signal cannot be processed, speed is 4 ");
    break;
    default: log.log("Signal cannot be processed");
} // end switch
} //end else (off received)
} //end else if (on==false)
```

Once the behavior is defined, we need to create a component and its instance to simulate this behavior.

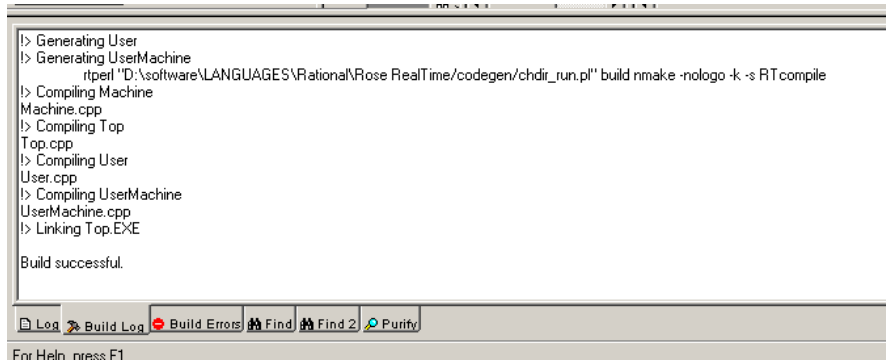
### **7. Creating a Component.**

- Right click on the Component View folder and Select New > Component.
- Rename the new component to **UserMachine1**.
- Double click on the component to open the Specification Dialog.
- Set the environment to C++TargetRTS and type C++Executable.
- In the References section drag and drop the Top capsule.
- Go to the C++Executable section and select the **Top** capsule into the Top Capsule option.
- In the C++Compilation select as **TargetConfiguration NT40T.x86-VisualC++-6.0**.
- Press OK to save changes.

### **8. Building a Component.**

- Right click on the Component **UserMachine1** and select **Set as Active**.
- Right click again and select Build > Build and press OK in the dialog with Generate and Compile.

- The dialog Add Missing Class References will appear, click OK.
- The output window will display the results:



```
> Generating User
> Generating UserMachine
rtperl "D:\software\LANGUAGES\Rational\Rose RealTime\codegen\chdir_run.pl" build nmake -nologo -k -s RTcompile
> Compiling Machine
Machine.cpp
> Compiling Top
Top.cpp
> Compiling User
User.cpp
> Compiling UserMachine
UserMachine.cpp
> Linking Top.EXE

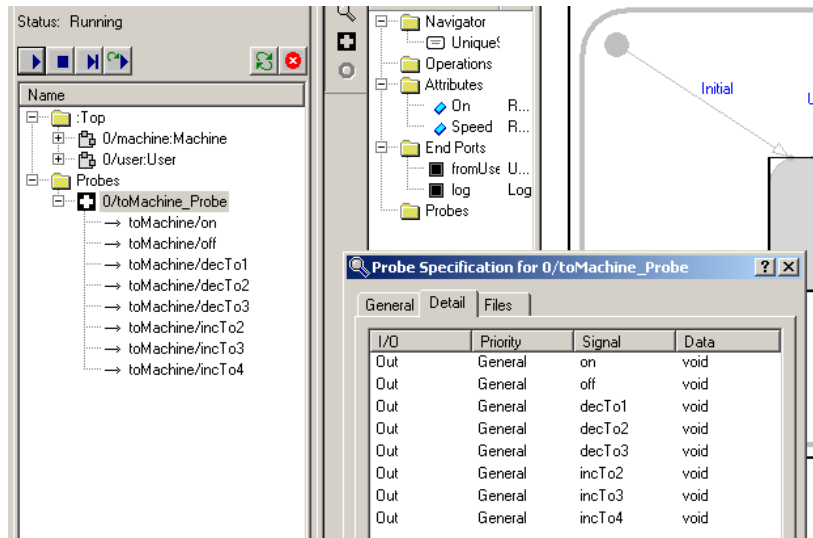
Build successful.
```

### **9. Running a Component Instance**

- Right click on the Deployment View folder and select New > Processor.
- Rename it to **LocalHost**.
- Drag and drop the **UserMachine1** component into the LocalHost. A **UserMachine1Instance** is created.
- Right click on the **UserMachine1Instance** and select Run.
- Select NO in the message Build the component (as we already did it in step 8),
- The execution control panel and a console window will appear.
- Click on the play button and Instances of the **Machine** and **User** capsules will appear.

### **10. Trace a Component Instance**

- Setting Probes.
  - Expand the Top capsule to see **Machine** and **User** capsules.
  - Right-click on the **User** Instance and select **Structure Monitor**.
  - Select the tool Probe and drag it into the **toMachine** port.
  - Right-click on the on the **toMachine** Probe and select Open Inject.
  - Right-click in the browser expand the probe and select Insert.
  - Insert all the signals defined in the protocol, with direction out.
  - Your window should look like:



- State monitors.
  - Close the structure monitor window.
  - Right-click on the Machine capsule and Open State Monitor.
  - Right-click on the Signals in the Probe window and
  - Inject signals decTo2, on, incTo2, IncTo3, off.
  - You must have the following results in the console window:

```

c:\Z:\RoseRT\Machine1\UserMachine1\build\Top.EXE
Rational Rose RealTime C++ Target Run Time System
Release 6.30.C.00 (<+c)
Copyright (c) 1993-2000 Rational Software
rosert: observability listening at tcp port 30719

*****
*           Please note: $TDIN is turned off.           *
*   To use the command line, telnet to the above mentioned port.   *
*   The _output_ of any command will be displayed in _this_ window. *
*****

Signal cannot be processed, machine is off
Machine turned on
Speed incremented to 2
Speed incremented to 3
Machine turned off
    
```

This is it with the 1<sup>st</sup> approach.

**Notes to the Single-state, single-transition model:**

Although we have accomplished the specified behavior, we did not model it (AT ALL!!!). Now consider that the behavior specification changes (as it always do) and you are required to allow changes of speed from 1 to 3 and from 1 to 4. What would be required to change in your RoseRT model to make it work? What about introducing a 5<sup>th</sup> speed or adding a reverse speed? If the actual code does not look like a mess already... Think about how it will look after the changes above?

Finally, consider that you are not the author of the original code, and you are required to give maintenance (make the changes) to the model above. Does it look like an easy task to do?

## II. SECOND APPROACH: Single state, multiple transitions.

Notice that in the first approach, we are not taking advantage of any modeling techniques: a unique transition manages all the signals that are received and the behavior is simulated in the code of this unique transition. In the second approach, the model will have again two interacting capsules, **User** and **Machine** within the **Top** capsule. For the machine capsule, a unique state will be created, but now **a transition for each signal** will be defined. Variables will store the on, off status and the speed.

- Repeat steps 1-6 of the first approach, or just rename your model as: **SpeedMachine2.rtmidl** to reuse your previous work.
- For step 7, open the Machine State diagram.
- Create a new state, name it **UniqueState**.
- Create an initial transition to it.

In the following steps a transition for each one of the In signals will be created, validated and actions will be executed.

**Use the help to learn the difference between log.log and log.show (it's the CR)**

- Create a transition to itself, rename it as **turnOn**.
- Double click on the on, double-click on the triggers section, insert the signal on from the port fromUser.
- Insert the following code on the actions section:

```
if (!On) {
    On=true;
    log.log("Machine turned on ");
}
else
    log.log("Machine already on ");
```

- Repeat the last 3 steps for signal **turnOff**, insert the following code in the actions section:

```
if (On){
    On=false;
    log.log("Machine turned off ");
}
else
    log.log("Machine is already off ");
```

- Create a transition for signal **decTo1**, with the following action:

```
if (On)
    if (Speed==2){
        Speed--;
        log.log("Speed decremented to 1");
    }
    else{
        log.show("Signal cannot be processed, speed is: ");
        log.log(Speed);
    }
else
    log.log("Signal cannot be processed, machine is turned off ");
```

- Create a transition for signal **decTo2**, with the following action:

```
if (On)
  if (Speed==3){
    Speed--;
    log.log("Speed decremented to 2 ");
  }
  else{
    log.show("Signal cannot be processed, speed is: ");
    log.log(Speed);
  }
else
  log.log("Signal cannot be processed, machine is turned off ");
```

- Create a transition for signal **decTo3**, with the following action:

```
if (On)
  if (Speed==4){
    Speed--;
    log.log("Speed decremented to 3 ");
  }
  else{
    log.show("Signal cannot be processed, speed is: ");
    log.log(Speed);
  }
else
  log.log("Signal cannot be processed, machine is turned off ");
```

- Create a transition for signal **incTo2**, with the following action:

```
if (On)
  if (Speed==1){
    Speed++;
    log.log("Speed incremented to 2 ");
  }
  else{
    log.show("Signal cannot be processed, speed is: ");
    log.log(Speed);
  }
else
  log.log("Signal cannot be processed, machine is turned off ");
```

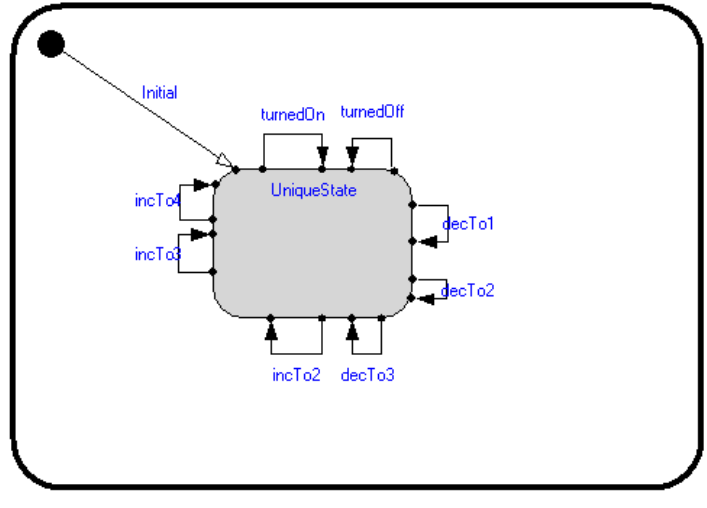
- Create a transition for signal **incTo3**, with the following action:

```
if (On)
  if (Speed==2){
    Speed++;
    log.log("Speed incremented to 3 ");
  }
  else{
    log.show("Signal cannot be processed, speed is: ");
    log.log(Speed);
  }
else
  log.log("Signal cannot be processed, machine is turned off ");
```

- And finally, create a transition for signal **incTo4**, with the following action:

```
if (On)
  if (Speed==3){
    Speed++;
    log.log("Speed incremented to 4 ");
  }
  else{
    log.show("Signal cannot be processed, speed is: ");
    log.log(Speed);
  }
else
  log.log("Signal cannot be processed, machine is turned off ");
```

- The state machine of your Machine capsule should look like:



- Repeat steps 8 to 11 to create a component UserMachine2, run and trace its instance.
- Inject signals **on**, **incTo2**, **decTo3**, **off**. Observe how the transitions are executed with each one of the signals injected.
- Your results should look like:

```
z:\RoseRT\Machine2\UserMachine2\build\Top.EXE
Rational Rose RealTime C++ Target Run Time System
Release 6.30.C.00 (<+c>)
Copyright (c) 1993-2000 Rational Software
rosert: observability listening at tcp port 30214

*****
* Please note: STDIN is turned off. *
* To use the command line, telnet to the above mentioned port. *
* The _output_ of any command will be displayed in _this_ window. *
*****

Machine turned on
Speed incremented to 2
Signal cannot be processed, speed is: int 2
Machine turned off
```

That's it for the second approach!

## Notes to the Single-state, multiple-transitions model:

Do you see any benefit of having multiple transitions as opposed of a single one? Considering the changes described at the end of the first approach, do you think that making them in this new model will be easier?

Here are some tips: to allow changes of increase of speed, you need to modify the code in two transitions (and you don't have to mess with the deeply nested if structure). To add a fifth speed, we need a new transition (again, no need to deal with the original nested if structure). Do you see any benefits now? Finally, play again the maintenance role, and ask yourself if you rather deal with the first model, or with the second?

Still, we are not done yet...

### III. THIRD APPROACH: Multiple-states, multiple transitions.

Notice that in the two previous models, we needed variables to store the status and the speed of the machine. If we take advantage of **hierarchical** state machines, our model will be more structured and clear, not having the behavior simulated by code.

In this approach, the machine capsule will have two states: on and off. The state on will contain a sub-state for each one of the speed states. A transition for each signal will be created. Variables are not needed any more.

- Repeat steps 1-2 of the first approach, name your model as **SpeedMachine3.rtmidl**.
- In the third step, skip the section add attributes to the Machine capsule.
- Repeat steps 4-6.
- For step 7, creating the state machine, open the Machine State diagram.
- Create two new states, **On** and **Off**.
- Create a transition from **Off** to **On** state. Rename it as **turnOn**. In the triggers section, add the signal **on** from the port **fromUser** and the action:

```
log.log("Machine turned on");
```

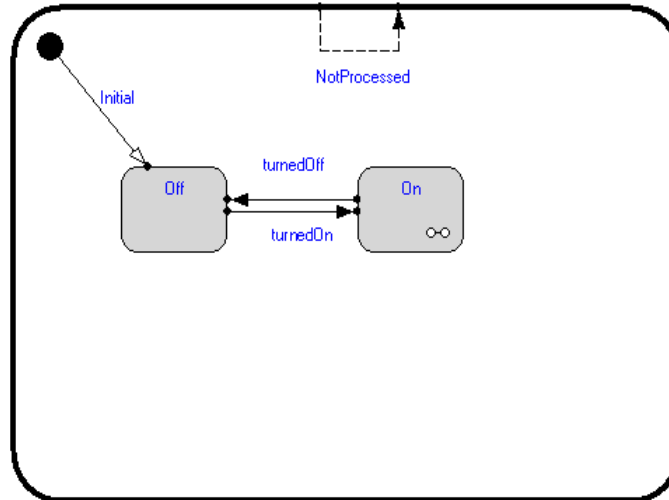
- Create a transition **turnOff** from **On** to **Off** state, triggered when the signal **off** is received:

```
log.log("Machine turned off");
```

- Create a self transition in the **top** state (see figure below) to “catch” all the signals that will not be processed by the off capsule, on capsule or all its sub-capsules.
- Double-click on this transition. Rename it as **NotProcessed**. Check the box to set it as internal. In the trigger section add all the signals (\*) from the fromUser port. Add in the action section:

```
log.log("Signal cannot be processed");
```

- Your capsule should look like this:



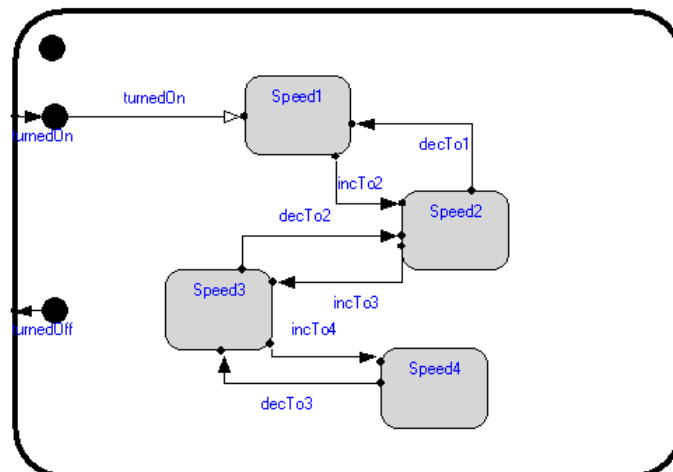
- Double click in the **On** state.
- Create 4 sub-states: **Speed1**, **Speed2**, **Speed3** and **Speed4**.
- There is no **initial** transition to **Speed1** (think about it: no need)
- Create a transition from turnedOn to the **Speed1** (the initial speed)
- Create a transition from **Speed1** to **Speed2**, name it **incTo2**, triggered by the **incTo2** signal, and displaying the message:

```
log.log("Speed incremented to 2 ");
```

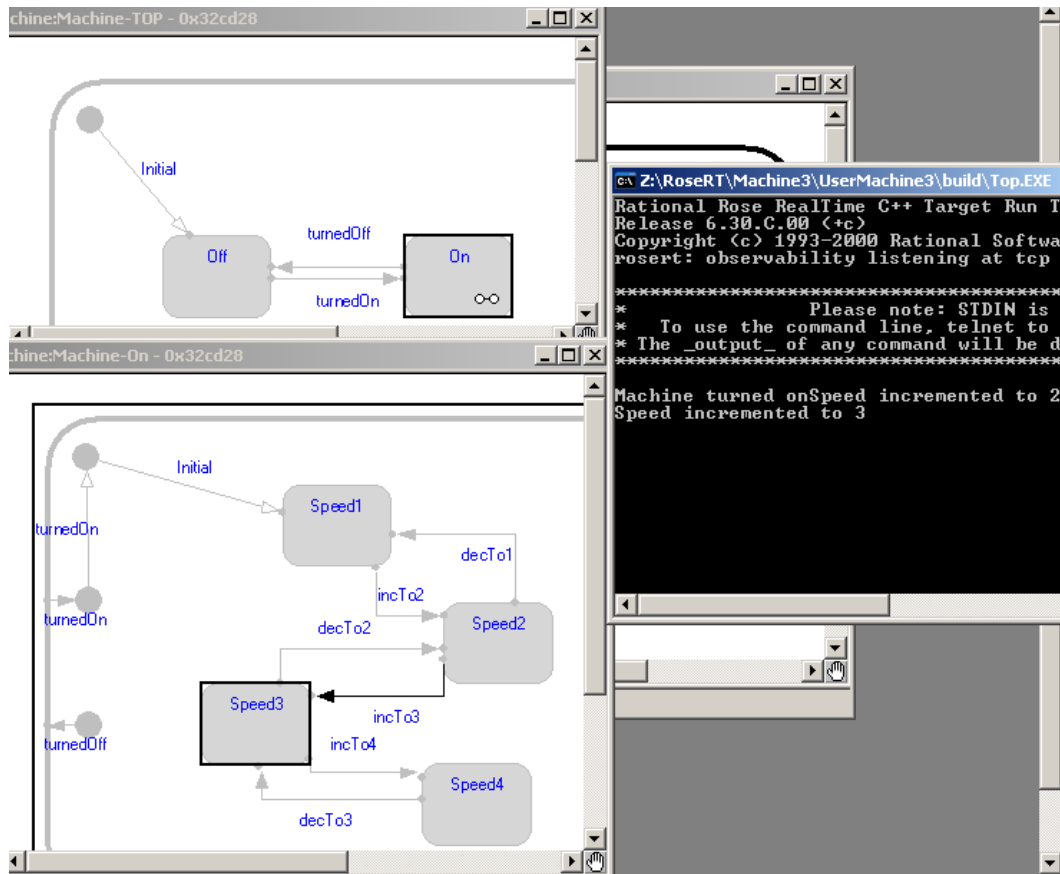
- Conversely, create a transition from **Speed2** to **Speed1**, name it **decTo1**, triggered by the **decTo1** signal, displaying the message:

```
log.log("Speed decremented to 1 ");
```

- Repeat the last two steps for the sub-states Speed3 and Speed4, the state On should look like:



- Repeat steps 8 to 11 to create a component **UserMachine3**, run and trace its instance.
- Open the state monitor for the machine capsule and the state on. Rearrange your window to see the behavior.
- Inject signals **on**, **incTo2**.
- Your results should look like:



### Notes to the Multiple-states, multiple-transitions model:

The main difference between this and the previous two approaches: variables **On** and **Speed** are replaced by states, the behavior of the model is more clear and structured. Notice that at any moment the machine can be turned off, we are taking advantage of modeling techniques.

Using the **NotProcessed** internal transition, the top capsule can handle all the incoming signals and display an error if a signal can not be processed.

This model is also more flexible for maintenance: suppose again that a new speed is to be added, speed5. In the first two approaches, the transitions and actions are to be changed, the if-nested code in the unique trigger of the first approach will be not easy to change. Conversely, in the third approach, you just need to add a new state and the appropriate transitions.

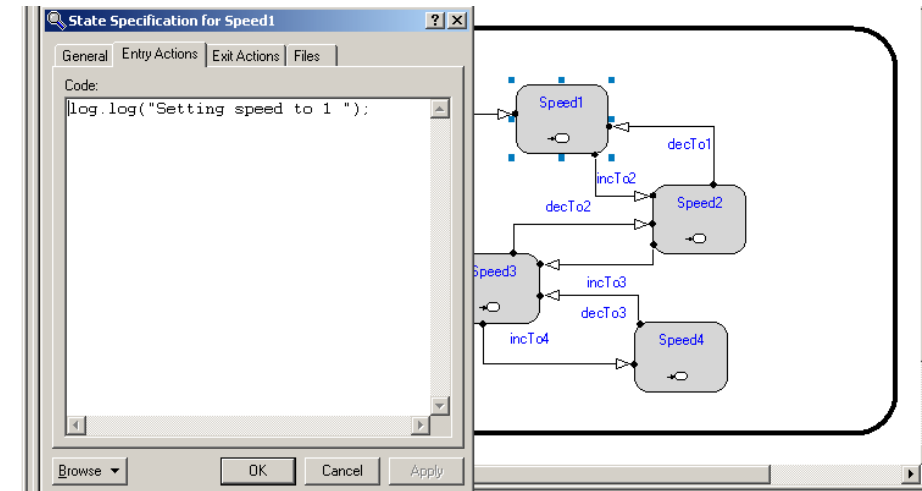
Still, we can improve.

#### IV. FOURTH APPROACH: Multiple-states, multiple-transitions, entry code.

This approach will take advantage of additional features of RoseRT. We will use the entry code of a state and eliminate the code of the transitions. The behavior of the machine behaves completely on the transitions and states, and not on the code.

- Open the model used in the third approach and rename it as **SpeedMachine4.rtmidl**.
- In the **On** state of the Machine capsule:
  - Delete all the actions of the transitions (**incTo2**, **incTo3**, **decTo2**, etc.).
  - Right-click on the **Speed1** state and open its specification.
  - In the section **Entry Actions** add the following code to display the **Speed1** state is the actual state of the simulation.

```
log.log("Setting speed to 1 ");
```



- Repeat the last step for the states **Speed2**, **Speed3** and **Speed4**.
- Repeat steps 8 to 11 to create a component **UserMachine4**, run and trace its instance with the examples of the third approach.
- Results must be the same as in the previous model, although we cannot identify whether the speed is reached incrementing or decrementing the speed, but this was not part of the requirement and makes our life easier 😊.

#### Notes to the Multiple-states, multiple-transitions , entry code model:

Although behavior is almost the same than in the previous approach, the model is less repetitive in terms of action code of transitions.

As in third approach, we are taking advantage of the property of states and transitions and not relying on variables.

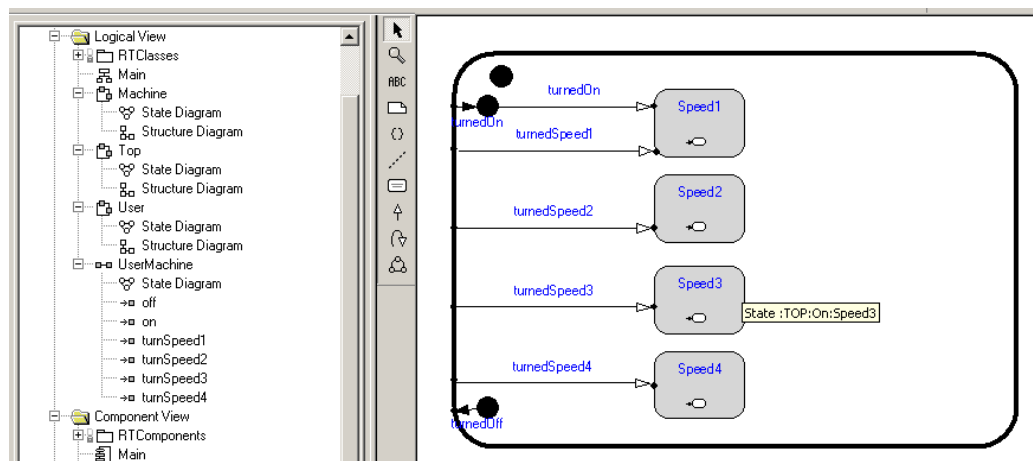
In addition, actions are executed when reaching each state (display the current speed) instead of the transitions. We take advantage of this feature when we want something to be done each and every time we arrive to the state.

To add a new speed is now easier and the only code we need to add is the specification for the new speed state. Analyze the advantages of using sub-states, internal transitions, entry code, etc.

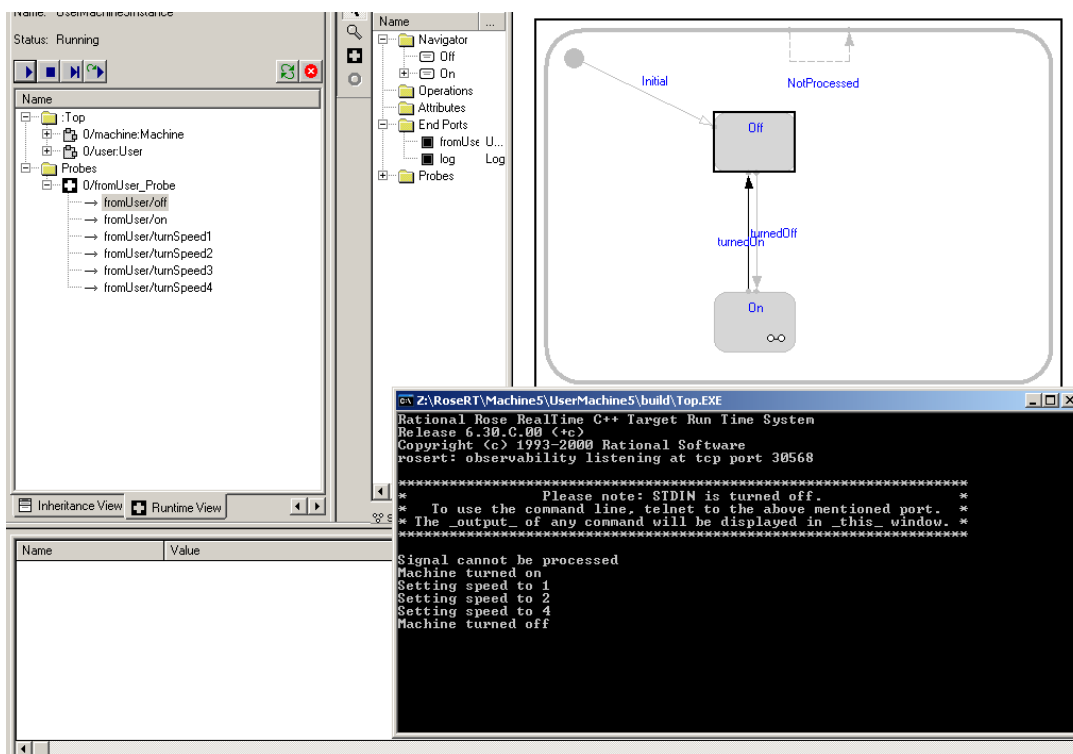
#### IV. FIFTH APPROACH: Specifications change.

Imagine that once you finished the model, specifications have changed (Does it sound familiar?). Initial speed is 1, but you can change to any other speed in any order. If you already have the fourth model, it is straight forward to do the changes.

- Open the model used in the fourth approach and rename it as **SpeedMachine5.rtmidl**.
- The UserMachine protocol will be changed:
  - Change the signals of the UserMachine protocol, keep **on** and **off** signals.
  - Delete all others (**incTo2**, **incTo3**, etc). (in fact you could keep this old functionality if you wanted)
  - Add InSignals: **turnSpeed1**, **turnSpeed2**, **turnSpeed3**, **turnSpeed4**.
- In the On state of the Machine capsule:
  - Delete all the sequential transitions from one speed to the other.
  - Add **group** transitions from the **on** state to **speed1**, **speed2**, **speed3** and **speed4**. Your model should look like this:



- Repeat steps 8 to 11 to create a component UserMachine5, run and trace its instance inject the following signals: **turnSpeed3**, **On**, **turnSpeed2**, **turnSpeed3**, **Off**.
- Your results should look like this:



### Notes to the last model:

Using modeling techniques, will allow us to have a more clear, understandable and flexible model.

Flexibility can be notice in this last model, when specifications were changed.

It was very easy to change transitions, and it would be very easy to add or delete speeds. (Just imagine going through the code of approach number one, and changing the behavior at this point!).

Similar to these basic modeling techniques, we will learn a set of tips and predefined structures that will make your life easy if you know how to use them.

Good luck!

Comments about this tutorial:  
Last updated August 23, 2005

Natalia Villanueva Rosales  
[nvillanu@scs.carleton.ca](mailto:nvillanu@scs.carleton.ca)

Juan Pablo Zamora Zapata  
[jpzzapat@scs.carleton.ca](mailto:jpzzapat@scs.carleton.ca)