JavaFrame modeling with Rational Software Modeler

Introduction

This guide will help you learn how to create and transform JavaFrame models using Rational Software Modeler and the JavaFrameTransformation plug-in. Models can be transformed to a running Java system.

Installation

Install the plug-in by extracting JavaFrameTransformation_x.zip to <RationalSoftwareModelerHome>/rsm/eclipse. The first time you start the program you may have to use the argument –clean in order to make eclipse search for new plugins.

Transforming

Create a Java project which will contain the generated java files.
- Click File > New > Project…
- Click Show All Wizards in the new Project wizard and select Java Project. Don't switch to Java perspective.
- Add javaframe.jar to the build path:
    - Select the java project in Model exporer and go to: Project > Properties > Java Build Path > Libraries > Add external jars…
    - javaframe.jar is located in <RationalSoftwareModelerHome>/rsm/eclipse/plugins/javaframetransformation_x/

Create a transformation configuration.
- Go to: Modeling > Transform > Configure Transformations
- Create a new UML2 to JavaFrame transformation and select the Java project as target.
- Click Apply and Close.

Now you can run the transformation. Right click a model element, select Transform and choose the transformation configuration.

Tip
Open Problems view to see if the generated code has errors and where they are. There sometimes are a lot of unused import warnings, which can be ignored.

Running the system

To run the java project you must create a run configuration.
- Switch to Java perspective (or just open a Java file)
- Go to: Run > Run… > Java Application > New
- Set the name of the configuration and locate the main file.

Trace
If you want to trace the system using JFTrace add the arguments:
 -remote localhost:54321

When using trace make sure to start JFTrace and open the default input socket before you run the JavaFrame system.

Modeling

Getting started

Create an UML project and select the Model element in Model explorer. In the Properties view, select Profiles and Add Profile. Select JavaFrameProfile from the dropdown menu. Ignore the warning about the profile not being released.

In order to be able to use Java types in the model, right click the model element and select Import Model Library > JavaTypes.
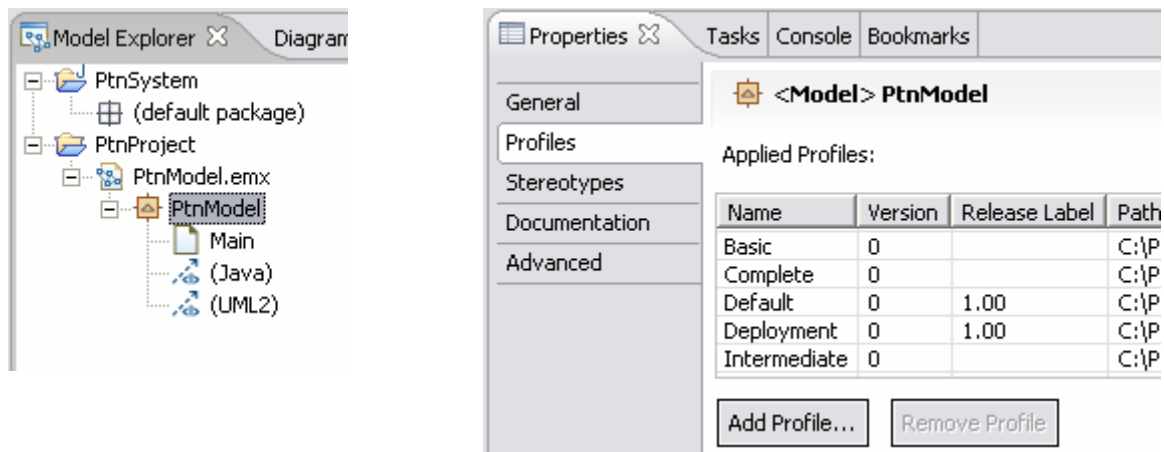
Finally add a package to the model. You always need a top-level package for your system.
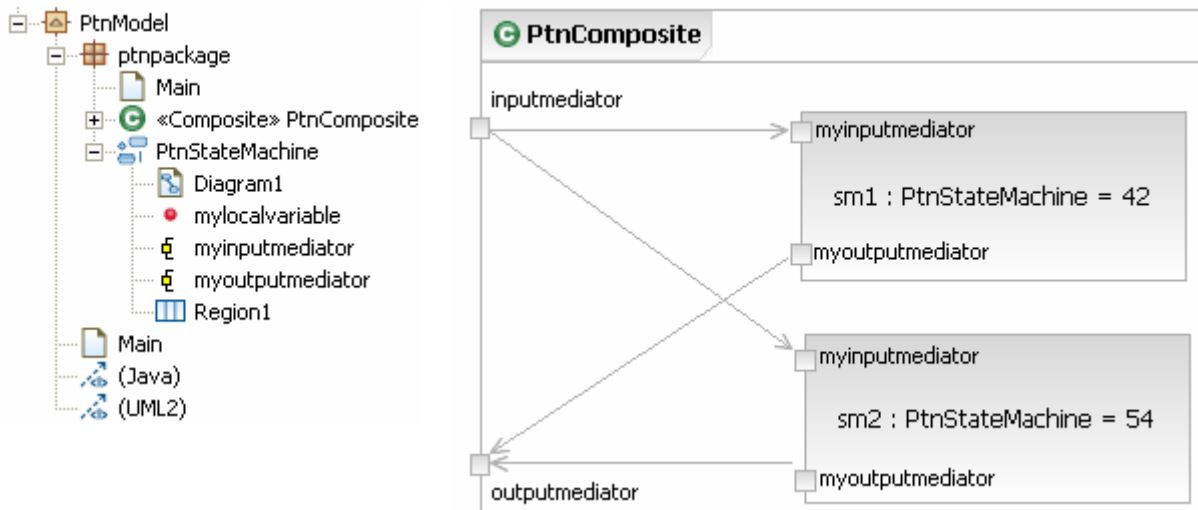
Composites

To create a composite add a Class to any package and apply the Composite stereotype. Then add a Composite Structure Diagram. Composites consist of ports, parts and connectors.

Ports can have a Mediator, defined elsewhere in the model, as type or be of unspecified type. If the type is unspecified there must be one and only one outgoing connector.

A part can be either another composite or a statemachine. If you want to add a statemachine as a part, first create it in Model explorer and drag it to the composite structure diagram. Notice that when you add a port to a statemachine-part in the diagram, the port element will be added to the statemachine in Model explorer and to all other parts of that type.

If a statemachine that is used as a part has parameters. You need to add arguments as a comma separated list in the part's Default Value field.

In the figure above PtnComposite has two parts, both of type PtnStateMachine. PtnStateMachine has a paramter called mylocalvariable of type int. The figure below shows sm2's Default Value field set to 54.
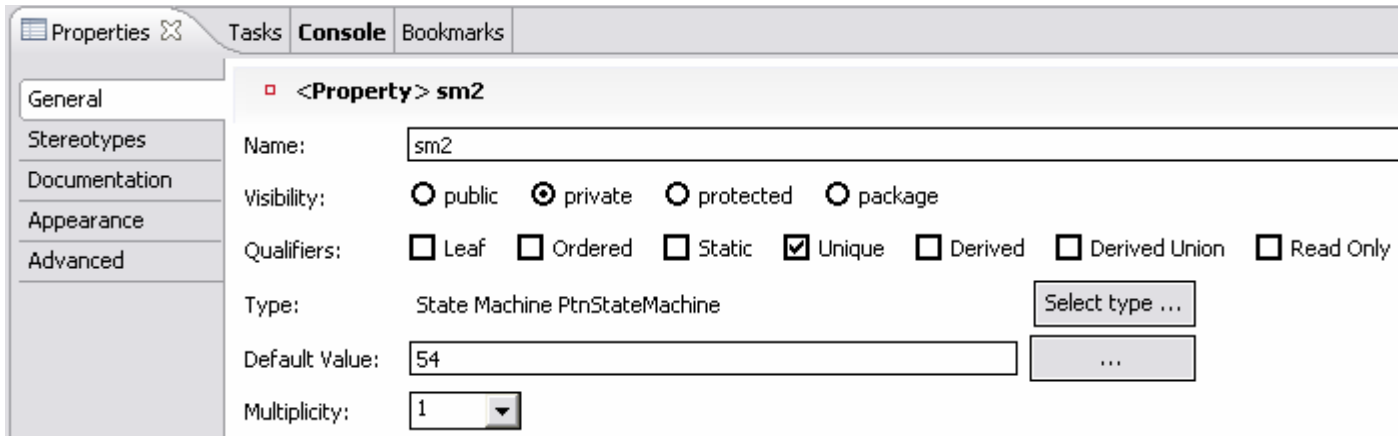
Pitfall
The Default Value field will be pasted into the constructor call of the statemachine and is not checked by the transformation. If you type incorrect code here you will most likely not get an error message but the generated code will be wrong. This is true for all user-code in the model.

Multiplicity
Parts can have multiplicity values 1 (default) and *. Any other multiplicity values will be treated as *. If * is chosen there is initially not added any instances of that part but a StateMachine can create instances with a <<Create>>Activity/Action. Parts with * multiplicity should not have any default value.

Main
The Composite stereotype has a property called main. If this is set to true there will be generated a Main class which creates this composite.

Signals

Signals are sent between ports and cause statemachines to trigger transitions. Signals can have attributes, which will be added as a variable to the generated java class for the signal. Signals can be abstract and they can extend other signals.

A signal element is transformed to a class extending the JavaFrame **Message** class, unless it extends another signal.
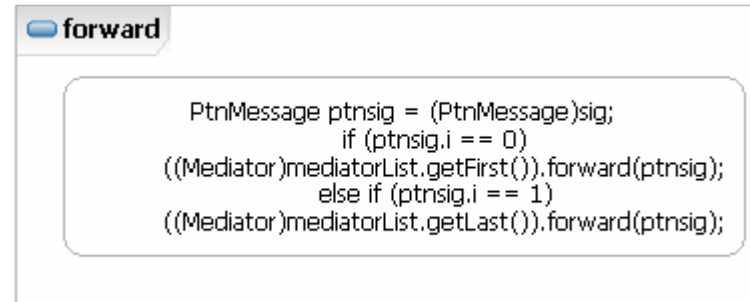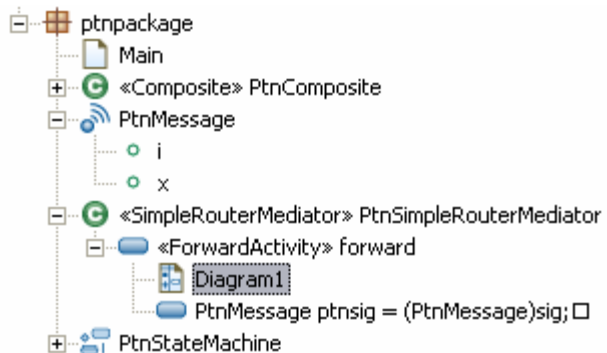
Ports / Mediators

Ports are transformed to javaframe mediators. A standard javaframe mediator can only have one address and will forward all signals it recieves to that address. If you want to define your own mediator behavior, create a class and apply the SimpleRouterMediator, the InputEdgeMediator or the Mediator stereotype. This class can then be used as a type for ports.

ForwardActivity

If you want to redefine what happens when the Mediator receives a Signal you must add an Activity to the class and apply the ForwardActivity stereotype to it. This is mandatory if you use SimpleRouterMediator.

All Activities in the model with be translated into code. If the Activity has an Action the code will be the name of the Action. If there is no Action the code will be the name of the Activity. The ForwardActivity code will be run every time the mediator receives a signal. The code has access to a **sig:Message** pointer which points to the signal received.

If the SimpleRouterMediator stereotype is applied you will have access to a **mediatorList:List** variable. This list contains **Addressable** objects, which can be either Mediators or Statemachines. Otherwise you will have access to an **address:Addressable** variable. The Addressable interface has a one method: **forward(Message sig)**


```
PtnMessage ptnsig = (PtnMessage)sig;
            if (ptnsig.i == 0)
((Mediator)mediatorList.getFirst()).forward(ptnsig);
            else if (ptnsig.i == 1)
((Mediator)mediatorList.getLast()).forward(ptnsig);
```

Tip
User Ctrl-Enter for line break when writing code in Actions. If you find it inconvenient to write code in the model, try using the comment //TODO instead of the real code. After you transform the model, the location of all the //TODO comments will be shown in the Tasks view. Replace the comment with the real code using the regular java editor and finally paste it back into the model.

Attributes
Mediators can have attributes just like Signals. However if a Mediator has attributes, any port that uses it as type must set them with a comma separated list in its Default Value field. There you will have access to all Attributes/Parameters defined in the Composite/Statemachine which owns the port.

StateMachines

Statemachines consist of states, pseudostates and transitions. Supported state types are State, SubmachineState and FinalState. Supported pseudostates are Initial, Entry, Exit and Choice.
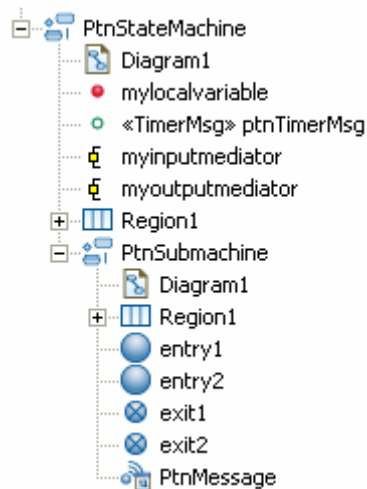
Attributes / Parameters

Statemachines can have both attributes and parameters. Both are added as a variable to the generated java class for the statemachine. Parameters must be set by the Default Value field of a part as explained under Composites. In order to add parameters right click the statemachine, select Properties and go to Parameters.

Attributes owned by statemachines can have the TimerMsg stereotype. The TimerMsg stereotype has a time property with a default value of 1000 milliseconds. This can be changed at Properties View > Stereotypes > Property. The timer is started / stopped by startTimer() and stopTimer() methods. When a TimerMsg reaches its time limit it will be sent to the statemachine, which will handle it like any other message. Both Type and Default Value of a TimerMsg attribute is ignored.

States
All states can have entry/exit activities. In the activity code for entry/exit you have access to a method: output(Message, Mediator, StateMachine) and a csm pointer which points to the enclosing StateMachine.

States can be either regular states or SubmachineStates. A SubmachineState has another statemachine as its submachine. A statemachine used as a Submachine in a SubmachineState, must be owned by the statemachine which owns the SubmachineState.

PtnStateMachine
    Diagram1
    mylocalvariable
    «TimerMsg» ptnTimerMsg
    myinputmediator
    myoutputmediator
    Region1
    PtnSubmachine
        Diagram1
        Region1
        entry1
        entry2
        exit1
        exit2
        PtnMessage

StateMachines used as submachines can have entry/exit points, but make sure to add any entry/exit point to the statemachine and not the region.

Pitfall
If you add entry/exit points to a SubmachineState in a diagram, they will be added as ConnectionPointReference elements. These will not automatically correspond to any entry/exit points defined in the submachine. You can set them manually, but a better way is to first create the Submachine, with all entry/exit points, then drag it to the diagram of the top-level statemachine. All ConnectionPointReference elements will be handled automatically.

Transition-triggers
Transitions from States should have trigger(s). Triggers can be added to a transition in two ways, the first is to add a SignalTrigger and select signal element(s). The second is to use the name of the transition. If a transition doesn't have a SignalTrigger element the name will be interpreted as a comma separated list of signals that trigger the transition. Since there is no TimerMsg signal the only way to add a TimerMsg trigger is to use the name of the transition.

When a statemachine receives a signal it will check all the triggers of the outgoing transitions in its current state. The transitions are checked in the order they are listed in Properties>Triggers of the State, which by default is the order they are added. An exception is a trigger-less transition, which is interpreted as "any signal", and is always checked last. If none of the current states transitions fire, transitions on the enclosing state are checked in the same way. If no transitions fire all the way to the top-level state the signal will be ignored.
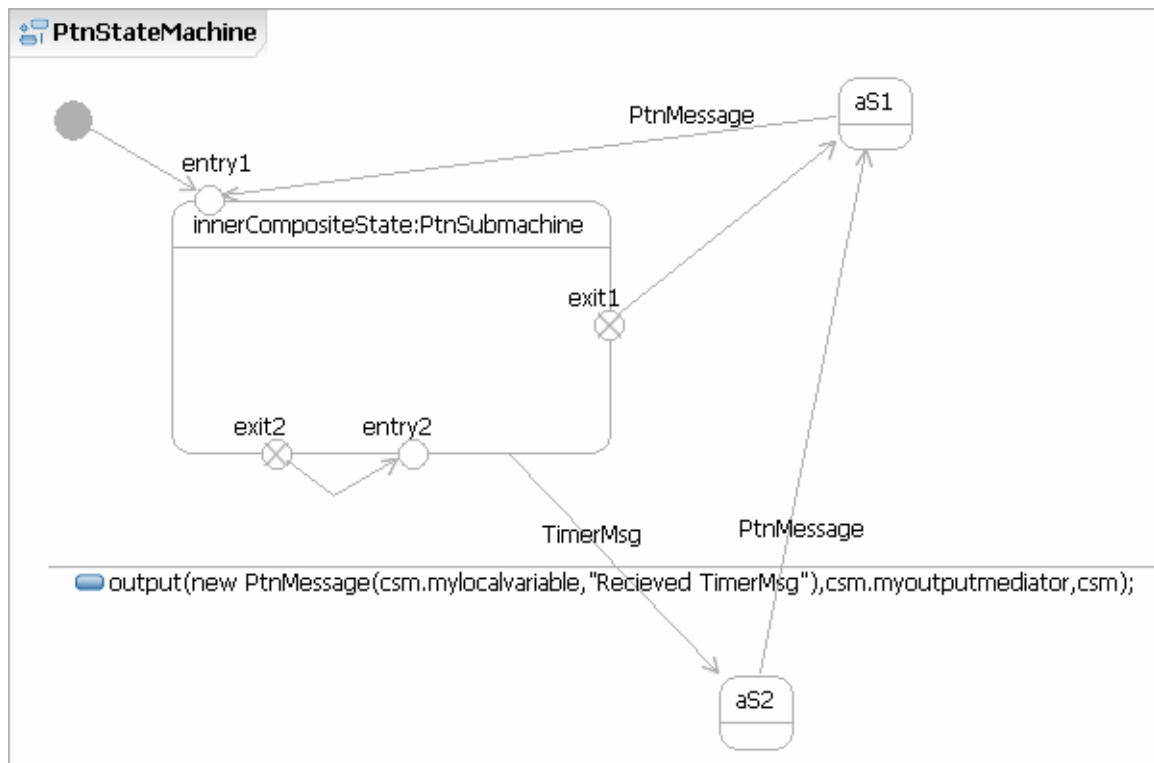
Transition effects
A Transition can have an effect. An effect is an Activity element which is transformed to java code and is run whenever the transition fires. If the activity has an Action element the code will be the name of the Action. If there is no Action the code will be the name of the Activity.

Transition effect code has access to these pointers/methods:
- output(Message,Mediator,StateMachine)
- sig : pointer to the signal that triggered the transition

- csm : the top-level statemachine this state is part of

If the transition can be triggered by more than one signal the sig pointer is of type Message, which is the supertype of all signals. If the transition is triggered by only one signal, the sig pointer has the type of that signal.

Only the top-level statemachines is transformed to a StateMachine class. And the csm pointer points to an object of that type. For instance in the PtnModel example the csm pointer points to a PtnStateMachine object, even in effect code in PtnSubmachine.

Transition create effects
If you want to add an instance to a part with multiplicity * in the composite that owns this statemachine, apply the Create stereotype to the Activity element. The name of the activity should be a java call statement with the name of the method equal to the name of the part. If the part is a statemachine and has parameters you need to add arguments to the call. Set the compositeOwner property of the Create stereotype to the name of the composite that owns this statemachine. This requires that the statemachine is only used as a part in one composite.

For example, you have a composite TestComposite containing a part creator:CreatorStateMachine with multiplicity 1 and another part sm:TestStateMachine with multiplicity *. You want to make a transition inside CreatorStateMachine add a TestStateMachine instance to sm. The name of that transitions <<Create>> activity should be **sm();** The compositeOwner property of the Create stereotype should be TestComposite. Using the Create stereotype requires that the CreatorStateMachine is only used in TestComposite.

Transition guards
Outgoing transitions from Choice Points should have a guard. A guard must either be a boolean expression or [else]. If the Choice Point doesn't have an out-transition with a [else] guard you must make sure that one out-transition always fires when the system reaches the Choice Point. If the system checks all out-transitions from a Choice Point and no transitions fire the model is ill-formed.
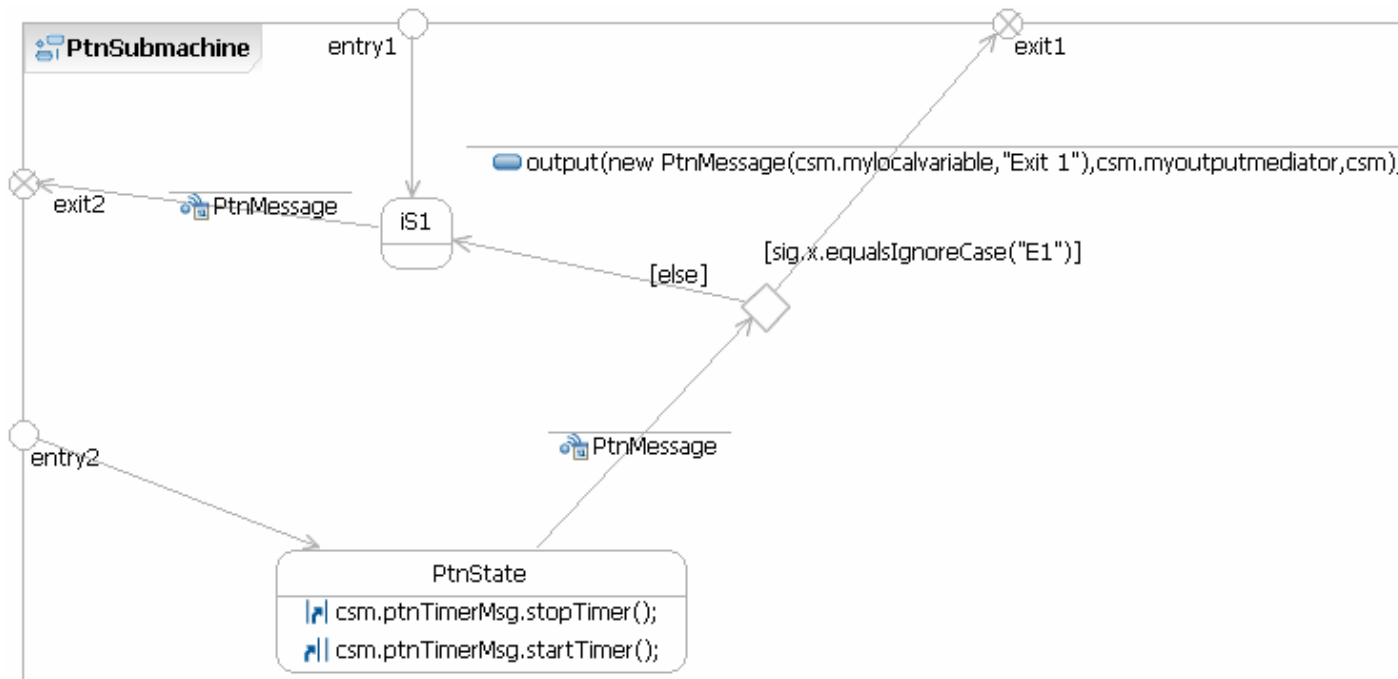
PackageImport

User code is not checked for required imports, so if an activity uses an element which is defined in another package the import must be set manually by adding a PackageImport element to the owner of the activity. Right click, select properties and PackageImport. Add a new PackageImport and select the correct package. PackageImports can be added to Classes, States and StateMachines.

Importing Java libraries

Using types defined in external java libraries is possible by creating and importing a ModelLibrary. A ModelLibrary represents a java library and is a standard UML model with the modelLibrary stereotype. Add the packages and classes needed from the external library to the ModelLibrary and import it to the JavaFrame model by right clicking the model element, selecting Import Model library > File.

Connecting to a test-GUI

Any composite can connect to a test-GUI. In order for the transformation to generate a GUI class you need to define all of the ports of the composite as either output or input.

You define a port as being output by applying the OutputEdgeMediator stereotype to the port. Input ports are defined by using class with the InputEdgeMediator stereotype as the type of the port.

You must define what kind of signals can be sent to the system by adding Reception element(s) to the type of an input port. To add a Reception right click the mediator class, select Properties and go to OwnedReception. Receptions should not have abstract signals.