JavaFrame modeling with Rational Software Modeler

1 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.

You will need two files:

- JavaFrameTransformation_x.zip
- javaframe.jar

2 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.

3 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 explorer and go to: Project > Properties > Java Build Path > Libraries > Add external jars…

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 your transformation configuration.

Tip
Open problems view to see if there are any errors in the generated code.

4 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

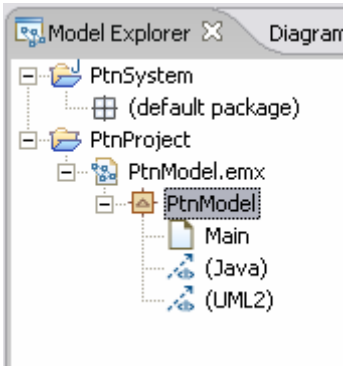
When using trace make sure to start JFTrace and open the default input socket before you run the JavaFrame system.

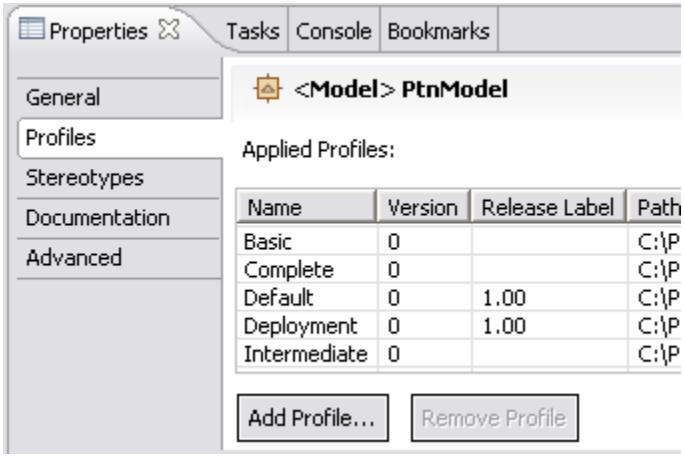
5 Modeling

5.1 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.

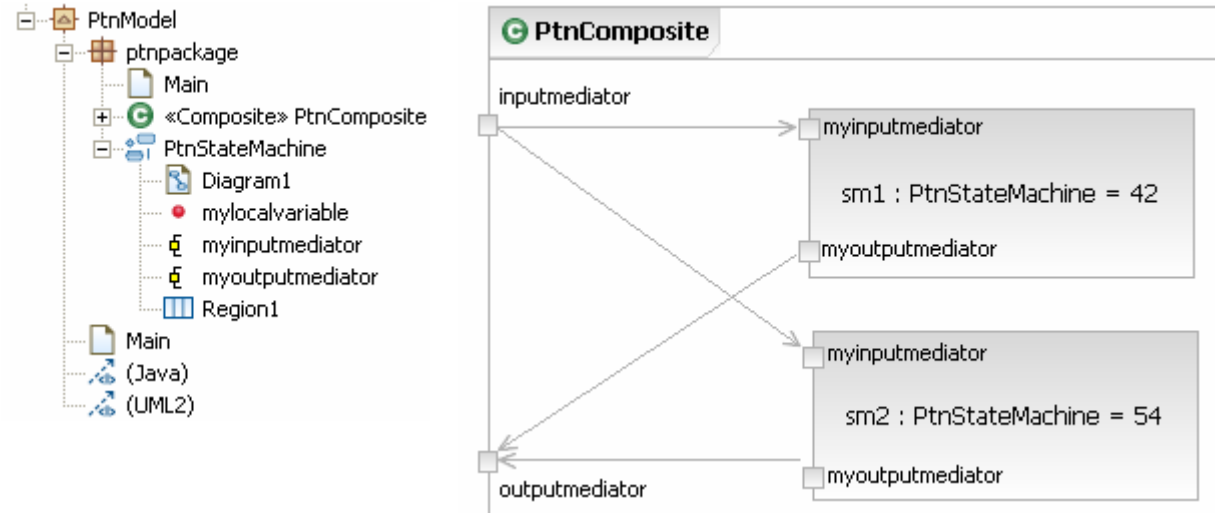
5.2 Composites

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.

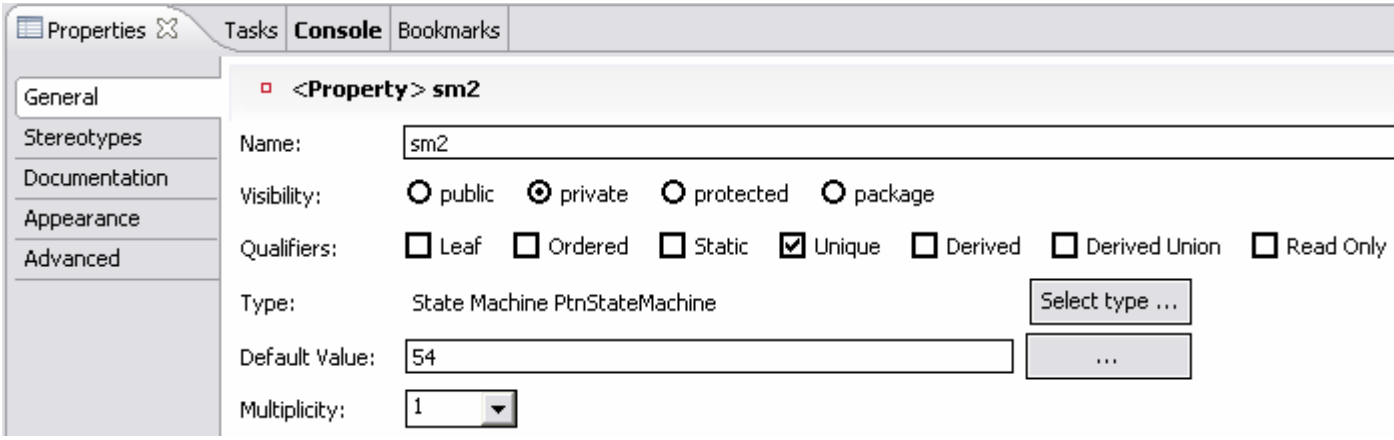
Ports can have a Mediator, defined elsewhere in the model, as type or be of unspecified type. If the type is unspecified there must be one and only one outgoing connector.

A part can be either another composite or a statemachine. If you want to add a statemachine as a part of a composite, 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 parameter 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 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.

Attributes

Attributes on Composites are treated like statemachine parameters in that they will be added to the constructor of the composite as a parameter. Anytime a composite is used as a part the attributes will need to be set in the part's default value field just like statemachine parameters.

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. The generated class will have a static array ARGS which will contain a copy of any arguments sent to the program on startup.

5.3 Signals

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

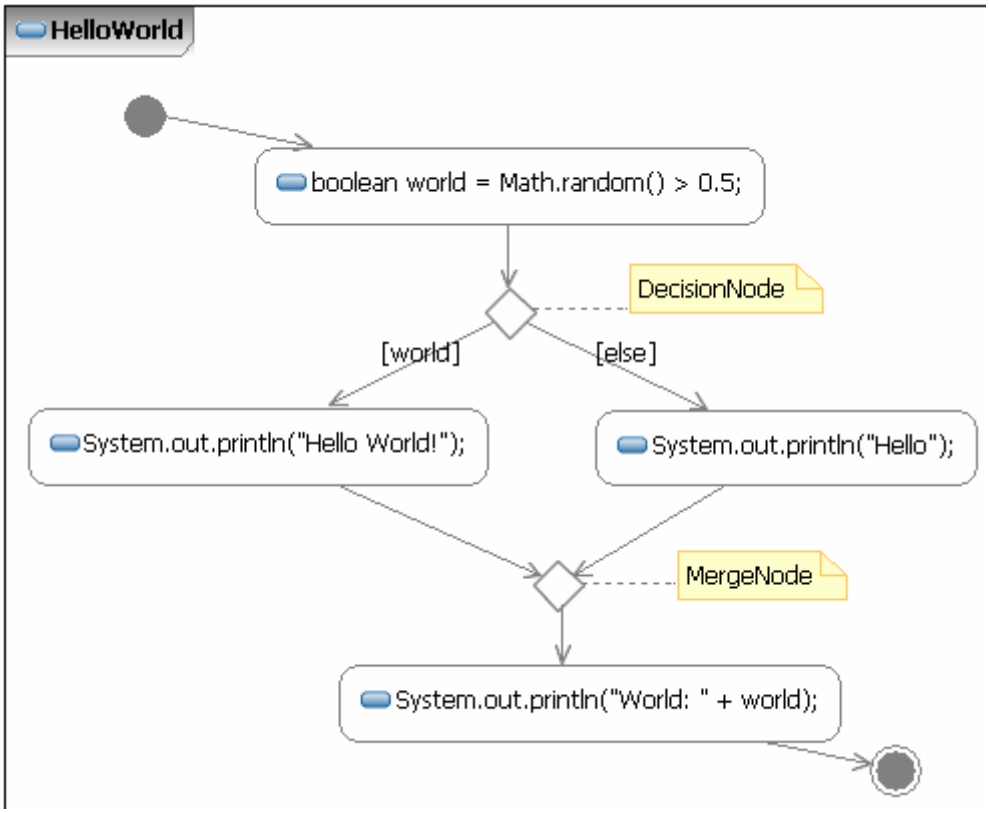
A signal element is transformed to a class extending the JavaFrame **Message** class, unless it extends another signal.

5.4 Activities

A number of places in the model there is a need to represent java code (e.g., to implement an operation or a transition effect), this is always done by an activity element.

If the code to represent is very short it is possible to just use the name of the activity element as the code.

If more code is needed it is better to use actions and control flows. An activity consists of one initial node, a number of control flows, actions, decision nodes and optionally final nodes. The names of actions are interpreted as java code. An example activity diagram is shown below.



Note that loops in the control flows are not supported. Also merge nodes are not really needed as several control flows can target one action or one decision node, effectively using it as a merge node. Only one output control flow from actions is supported.

5.5 Ports / Mediators

Ports are transformed to javaframe mediators. A port can be of different type depending on what you want it to do when it receives a signal. The different types are described below.

Mediator

Mediator is the default type, i.e. if a port has no type it will be interpreted as a standard mediator. A standard javaframe mediator can only have one address and will forward all signals it receives to that address. This means that the UML port can only have one outgoing connector and that this connector must be connected to a port on a part with multiplicity 1.

MultiCastMediator

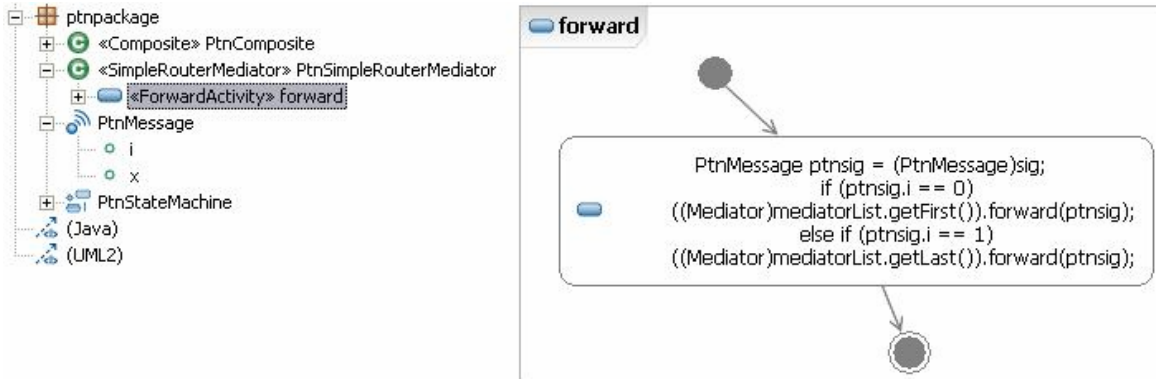
A MultiCastMediator can have several addresses and signals it receives are forwarded to all its addresses. Like a standard mediator a MultiCastMediator does not have a type, but the MultiCastMediator stereotype must be applied to the port element.

SimpleRouterMediator

If you want to define your own behavior when a mediator receives a signal you can use a SimpleRouterMediator. To specify that a port should be a SimpleRouterMediator you

first need to create a class and apply the SimpleRouterMediator stereotype to it, then you set the type of the port to that class. The class also needs an activity element with the ForwardActivity stereotype applied. Like all other activity elements this will be translated into code.

The ForwardActivity specifies the implementation of a method **forward(Message sig)**. In addition to the **sig:Message** parameter the code has access to a **mediatorList:LinkedList** field. That list contains all the mediator objects that this mediator has outgoing connections to. An example of a SimpleRouterMediator with a ForwardActivity is shown below.



SimpleIdRouterMediator

SimpleIdRouterMediator is an extension to SimpleRouterMediator and is usually used to forward signals to a part with multiplicity * based on an id. In addition to the mediatorList field and the forward method SimpleIdRouterMediator has a **addId(String id)** method and a **idList:LinkedList** field. The addId method must be called every time an object is added as an address to this mediator. That is every time an object is added to the part that this mediator is connected to. Objects are added to parts with multiplicity * by using a <<Create>> action as explained under Transition create effects.

Like SimpleRouterMediator a SimpleIdRouterMediator must also have a ForwardActivity. By using addId as described above, the position of an id in idList will correspond to the position of the correct mediator in mediatorList. This gives the following pseudo code for the ForwardActivity (which must be manually implemented):

- Get the id from the signal
- Traverse idList until you find the correct id
- Forward the signal to the mediator object of the same position in mediatorList as the id had in idList.

If the part this mediator forwards signals to is a statemachine and that statemachine enters a FinalState, all mediators of the statemachine will automatically be removed from the mediatorList of this mediator. The corresponding id object in idList will also be removed. If the part is a Composite it can manually be removed by using the **removeActiveObject(ActiveObject oldActiveObject)** method on the owning Composite object.

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.

5.6 StateMachines

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

Attributes / Parameters

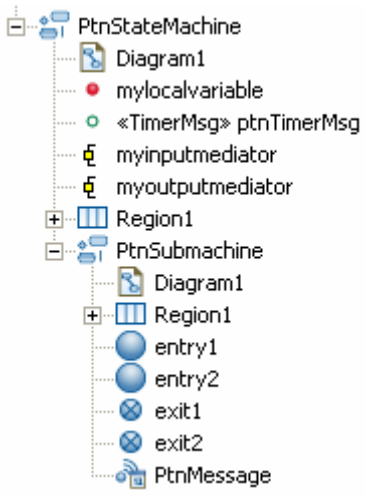
Statemachines can have both attributes and parameters. Both are added as fields to the generated java class for the statemachine. Parameters must be set by the Default Value field of a part as explained under Composites. In order to add parameters right click the statemachine, select Properties and go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



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

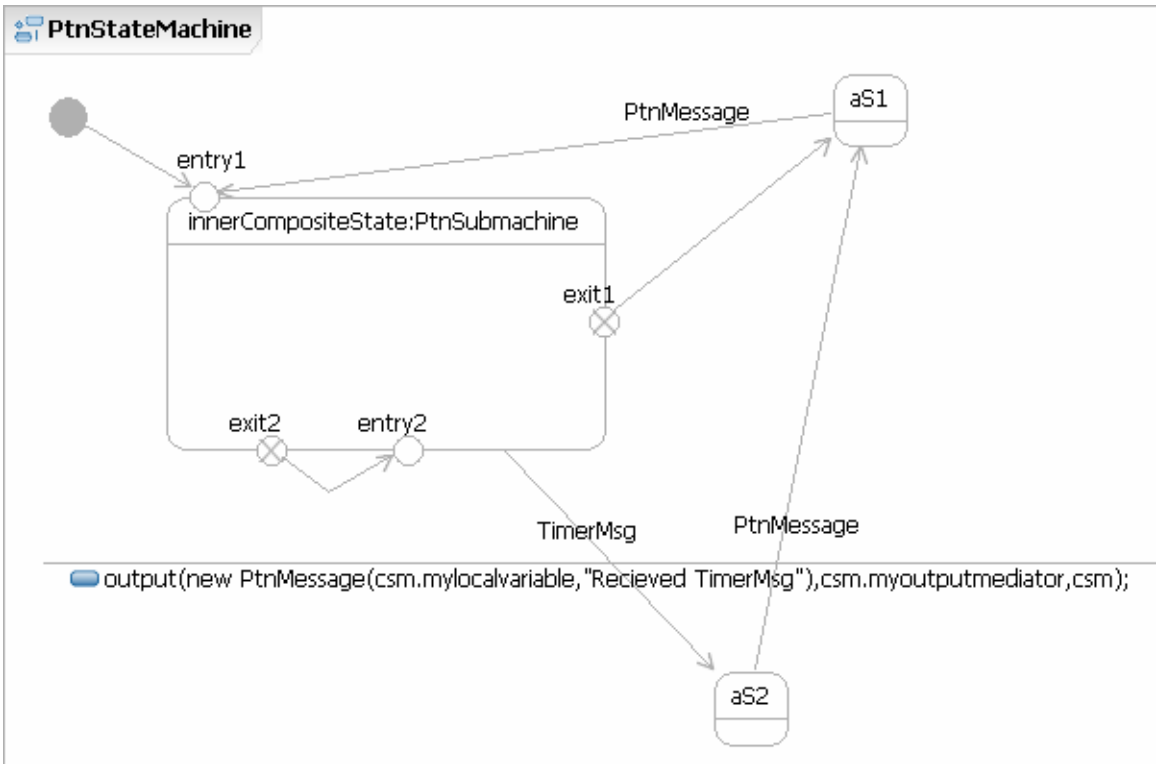
If you add entry/exit points to a SubmachineState in a diagram, they will be added as ConnectionPointReference elements. These will not automatically correspond to any entry/exit points defined in the submachine. You can set them manually, but a better way is to first create the Submachine, with all entry/exit points, then drag it to the diagram of the top-level statemachine. All ConnectionPointReference elements will then 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, no assumptions about the order the transitions are checked should be made. If none of the current state's transitions fire, transitions on the enclosing state are checked. If no transitions fire all the way to the top-level state the signal will be ignored.

A state can have deferrable triggers, to add a deferrable trigger create a signaltrigger element, right click the state > Properties > DeferrableTrigger and add the signaltrigger. Deferred signals do not trigger any transitions they are saved until the statemachine enters another state where they can trigger transitions just like newly arrived signals.



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.

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 region is part of

If the transition is triggered by one signal, the sig pointer has the type of that signal. If the transition can be triggered by more than one signal the sig pointer is of the lowest common type or the Message type if the trigger signals have no common type. Message is the supertype of all signals.

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

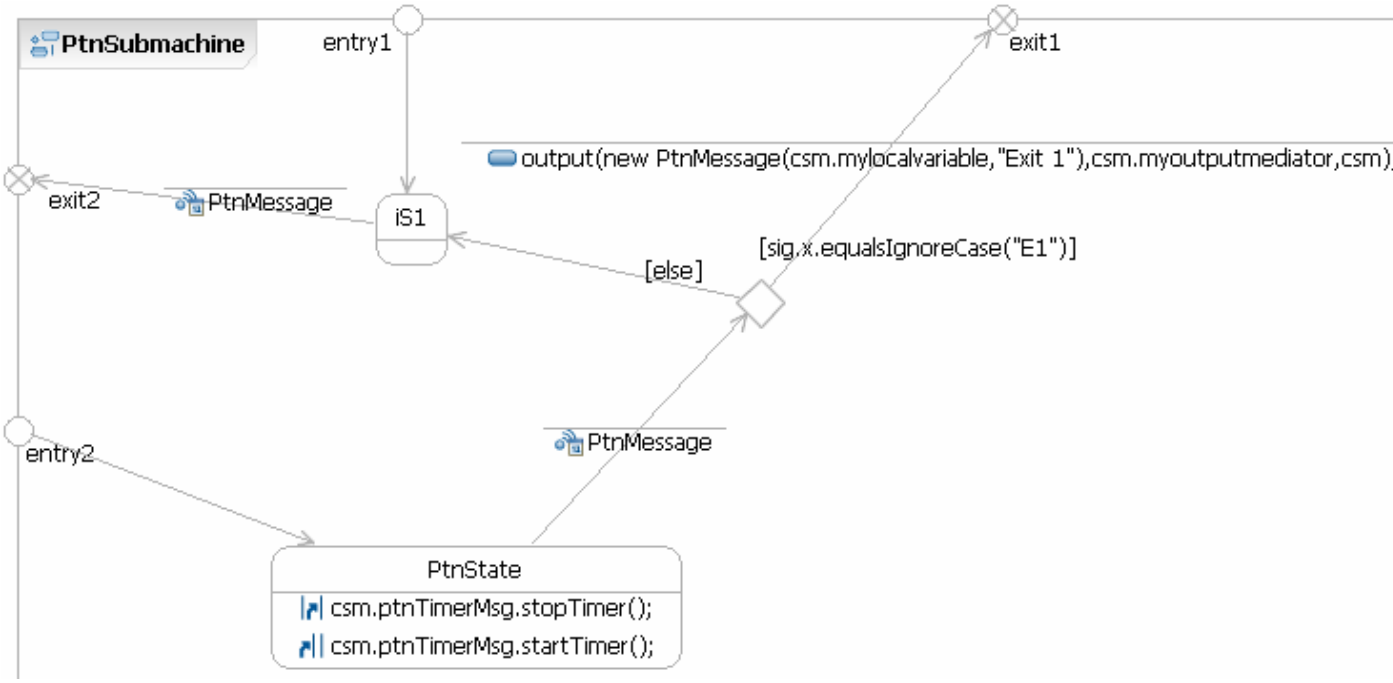
composite that owns this statemachine. This requires that the statemachine is only used as a part in one composite.

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. Remember that when a transition has fired it must always end up in another state, otherwise the model is ill-formed (and the generated program will crash).

Junction and Choice points are treated equally in that they both can have multiple in and out transitions. The difference is that a Choice point should have more than one out transition.



5.7 Common elements

Normal classes

Classes with neither a Mediator nor a Composite stereotype will generate a normal java class. A constructor will be generated with all the non-static attributes as parameters just like for Mediators, Signals.

Operations

StateMachines, composites, mediators, and normal classes can declare operations by adding an operation element. In addition to the operation element you also need to add a Method Activity to the operation in order to specify the implementation (i.e., the Method Activity is the body of the corresponding java method and the Operation element is the signature).

Arrays

Any attributes / parameters which have upper multiplicity more than 1 will be transformed into an array.

Static attributes

Any static attributes are not added to the constructor as arguments/parameters.

5.8 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.

5.9 Transformation configuration properties

The transformation configuration window has a properties tab with these properties:

Overwrite output files - whether any existing files should be overwritten

Generate default constructors – if set to true empty constructors are generated in addition to a constructor with all attributes of a class as parameters.

Organize imports – if true will schedule the eclipse organize imports command on all generated files as a background job after the transformation completes. The organizer searches for types in the generated files, in any imported models, in the JavaFrame API and finally in packages defined in the organize imports preferences for eclipse (can be found at Window > Preferences > Java > Code Style > Organize imports).

Note that in case of a type conflict the organizer will choose an arbitrary type.

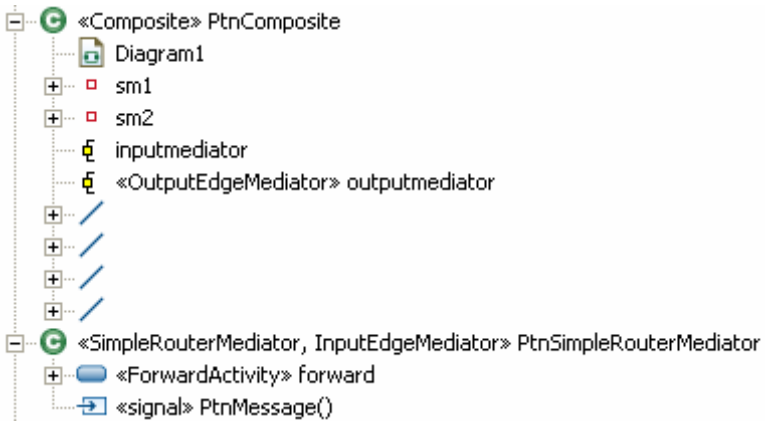
Import All Packages – will generate on-demand imports for all packages in the model and any imported models.

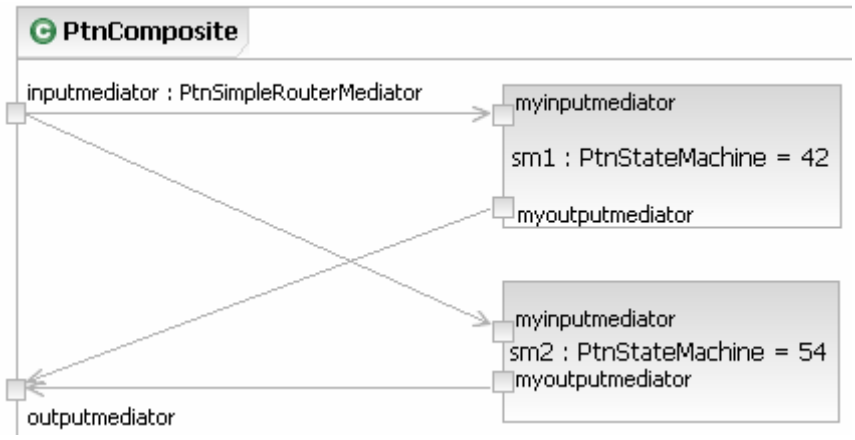
5.10 Connecting to a test-GUI

Any composite or statemachine can connect to a test-GUI. In order for the transformation to generate a GUI class you need to define one or more of the ports 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.





6 Getting the source code

The plugins include the source code. To import it into the workspace go to File > Import > External Plug-ins and Fragments, select import as projects with source folders and click next. Now select the plugins you want to import. The transformation consists of two plugins: no.uio.ifi.javaframeprofile and no.uio.ifi.javaframetransformation. Each plugin is imported as a project. Finally you should switch to the plug-in development perspective.