



Screen Orchestrator Guide

Version 2004.5

September 2004

Siebel Systems, Inc., 2207 Bridgepointe Parkway, San Mateo, CA 94404

Copyright © 2004 Siebel Systems, Inc.

All rights reserved.

Printed in the United States of America

No part of this publication may be stored in a retrieval system, transmitted, or reproduced in any way, including but not limited to photocopy, photographic, magnetic, or other record, without the prior agreement and written permission of Siebel Systems, Inc.

Siebel, the Siebel logo, TrickleSync, Universal Agent, and other Siebel names referenced herein are trademarks of Siebel Systems, Inc., and may be registered in certain jurisdictions.

Other product names, designations, logos, and symbols may be trademarks or registered trademarks of their respective owners.

PRODUCT MODULES AND OPTIONS. This guide contains descriptions of modules that are optional and for which you may not have purchased a license. Siebel's Sample Database also includes data related to these optional modules. As a result, your software implementation may differ from descriptions in this guide. To find out more about the modules your organization has purchased, see your corporate purchasing agent or your Siebel sales representative.

U.S. GOVERNMENT RESTRICTED RIGHTS. Programs, Ancillary Programs and Documentation, delivered subject to the Department of Defense Federal Acquisition Regulation Supplement, are "commercial computer software" as set forth in DFARS 227.7202, Commercial Computer Software and Commercial Computer Software Documentation, and as such, any use, duplication and disclosure of the Programs, Ancillary Programs and Documentation shall be subject to the restrictions contained in the applicable Siebel license agreement. All other use, duplication and disclosure of the Programs, Ancillary Programs and Documentation by the U.S. Government shall be subject to the applicable Siebel license agreement and the restrictions contained in subsection (c) of FAR 52.227-19, Commercial Computer Software - Restricted Rights (June 1987), or FAR 52.227-14, Rights in Data—General, including Alternate III (June 1987), as applicable. Contractor/licensor is Siebel Systems, Inc., 2207 Bridgepointe Parkway, San Mateo, CA 94404.

Proprietary Information

Siebel Systems, Inc. considers information included in this documentation and in Siebel eBusiness Applications Online Help to be Confidential Information. Your access to and use of this Confidential Information are subject to the terms and conditions of: (1) the applicable Siebel Systems software license agreement, which has been executed and with which you agree to comply; and (2) the proprietary and restricted rights notices included in this documentation.

Contents

1	Overview.....	8
2	Statechart and State Machine Concepts.....	9
2.1	WHAT ARE STATECHARTS?	9
2.2	WHAT IS THE STATE MACHINE?	10
2.3	WHY USE STATECHARTS AND THE STATE MACHINE?	10
2.4	STATECHART NOTATION EXPLAINED.....	11
2.4.1	States	11
2.4.2	Events	13
2.4.3	State Transitions.....	13
2.4.4	Pseudo-States	15
2.4.5	Chart Notes	17
2.5	SIMPLE STATECHART EXAMPLE.....	18
3	Basic Orchestrator Drawing.....	20
3.1	THE MAIN ORCHESTRATOR WINDOW.....	20
3.2	THE STATECHART DRAWING COMPONENTS	23
3.2.1	Drawing States or Pseudo-states.....	23
3.2.2	Drawing State Transitions	25
3.2.3	Drawing Chart Notes.....	30
3.3	MORE DRAWING STATE DETAILS.....	31
3.3.1	Adding Child States	31
3.3.2	Moving States	32
3.3.3	Editing State Details.....	32
3.3.4	Resizing States.....	34
3.3.5	Deleting states	35
3.4	MORE DRAWING TRANSITION DETAILS.....	35
3.4.1	Note On Transition Arrows.....	35
3.4.2	Drawing Transitions To The Master State	36
3.4.3	Drawing Transitions To And From Parent And Child States	38
3.4.4	Drawing Transitions To And From Non-Related Child States	38
3.4.5	Editing Transition Details	38
3.4.6	Deleting Transitions	39
3.5	MORE DRAWING STATECHART DETAILS	39
3.5.1	The Statechart Name.....	40
3.5.2	Renaming The Statechart	40
3.5.3	Saving A Statechart	41
3.5.4	Renaming a Saved Statechart	42
3.5.5	Opening A Statechart.....	42
3.6	MISCELLANEOUS DRAWING FEATURES	42

3.6.1	Using The Grid And Snap To Features	42
3.6.2	Using The Navigation Panel.....	43
3.6.3	Printing Statecharts.....	44
3.6.4	Export StateChart as a GIF	44
4	Preview And Web Deployment Capabilities	45
4.1	INTRODUCTION	45
4.2	PREVIEW CAPABILITY.....	45
4.3	WEB DEPLOYMENT CAPABILITY	48
4.3.1	Install an appropriate web server	48
4.3.2	WAR properties saved per statechart.....	48
4.3.3	Configuring the WAR file within the tool	49
4.3.4	Deploying the war file.....	49
5	Designing Events With Processes And Guard Conditions.....	51
5.1	HANDLING AN EVENT	51
5.2	ASSOCIATING PROCESSES WITH EVENTS AND TRANSITIONS	51
5.3	SETTING THE INPUT REQUIREMENTS.....	53
5.4	DELETING INPUT REQUIREMENTS	54
5.5	HOW THE REQUEST DATAPACKETS ARE BUILT.....	54
5.6	DEFINING GUARD CONDITIONS.....	55
5.6.1	NullGuardCondition.....	55
5.6.2	FixedValueGuardCondition	55
5.6.3	InputBasedGuardCondition.....	56
5.6.4	ResultBasedGuardCondition.....	57
5.6.5	TimeoutGuardCondition	58
5.6.6	EmptyResponseGuardCondition	59
5.7	OTHER CONTROLLER CLASSES.....	60
5.7.1	The SimpleController	60
5.7.2	The AutoViewController	60
5.7.3	Additional Controllers.....	61
5.7.4	Custom Controllers	61
5.8	ADD COMMON FIELDS TO EVERY REQUEST	61
5.9	WORKED EXAMPLE	62
5.10	BLOCKING EVENTS FROM STATES	66
6	Writing Controller Classes	68
6.1	THE RESPONSIBILITIES OF A CONTROLLER	68
6.2	THE ICONTROLLER INTERFACE.....	69
6.3	THE SIMPLECONTROLLER CLASS	71
6.4	THE MAIN CONTROLLER CLASS.....	71
6.5	THE MODIFIED CONTROLLER CONTRACT	72

6.5.1	The Inputs Object	73
6.6	EXTENDING THE CONTROLLER CLASS	73
6.7	ADDING A NEW CONTROLLER TO THE ORCHESTRATOR	76
6.7.1	Do Nothing.....	76
6.7.2	Add the controller to the statechart.properties file	76
6.7.3	Create a Customizer for the Controller.....	77
7	Writing Guard Condition Classes	79
7.1	THE RESPONSIBILITY OF A GUARD CONDITION	79
7.2	THE IGUARDCONDITION INTERFACE	79
7.3	ADDING A NEW GUARD CONDITION TO THE ORCHESTRATOR.....	79
7.3.1	Do Nothing.....	79
7.3.2	Add the guard condition to the statechart.properties file.....	79
7.3.3	Create a customizer for the guard condition.....	80
8	Writing JSPs	81
8.1	RESPONSIBILITIES OF A JSP	81
8.2	GETTING DATA INTO THE JSP	81
8.2.1	Inputs bean.....	81
8.2.2	ProcessExecutionRecords bean	82
8.2.3	State bean	82
8.2.4	View bean.....	82
8.2.5	RequestContext bean	82
8.3	FIRING AN EVENT FROM A JSP	83
8.3.1	Using the .jsm URL extension.....	83
8.3.2	Using the StateMachine URL.....	83
9	Orchestrator Process Integration	84
9.1	INTRODUCTION	84
9.2	IMPORTING IN PROCESSES FROM AN AUTOMATED METHODOLOGY MODEL	84
9.3	MANUALLY INPUTTING PROCESS INFORMATION.....	86
9.4	REMOVING ALL SESSIONS FROM THE SIEBEL PROCESS LIST	88
9.5	EDITING/DELETING PROCESSES	88
9.6	ASSIGNING PROCESSES TO THE STATE CHART.....	88
9.6.1	Assigning processes to a state.....	88
9.6.2	Adding Processes to a State Transition	89
10	Advanced Drawing	94
10.1	UNDO / REDO FEATURES	94
10.1.1	Undo Example.....	94
10.1.2	Redo Example.....	97
10.2	COPY, CUT & PASTE FEATURES	99

10.2.1	Copy Example	100
10.2.2	Cut & Paste Example	102
10.3	PARENT STATES AS SUB-CHARTS	104
10.3.1	Transitions Leaving And Entering Parent States	107
10.3.2	Bringing Sub-charts To The Front.....	108
10.3.3	Why Use The Sub-chart Feature?	109
10.4	MULTIPLE USER SUPPORT	109
10.4.1	Note on users working on the same files	111
11	Introduction To Writing A Swing Application.....	112
11.1	OVERVIEW	112
11.1.1	StateMachineEvent class	112
11.1.2	StateMachineEventSource interface.....	113
11.1.3	StateMachineEventDispatcher class.....	113
11.1.4	ViewContainer interface.....	113
11.2	WRITING THE APPLICATION MAIN CLASS	113
11.2.1	Create and display a ViewContainer.....	113
11.2.2	Create a StateMachineEventDispatcher	113
11.2.3	Fire the first Event	113
11.3	WRITING THE VIEWCONTAINER CLASS	113
11.4	WRITING THE VIEW CLASSES.....	114
11.4.1	IView interface.....	114
11.4.2	StateMachineEventSource interface.....	114
11.5	PUTTING THE VIEW CLASSES IN THE CHART.....	115
11.6	MANAGING VIEWPROPERTIES	115
11.7	THE TOOL VIEW REQUIREMENTS	115
11.7.1	The JSPView class.....	117
11.7.2	The JSPViewBeanInfo Class	121
11.8	THE SWING APPLICATION REQUIREMENTS.....	123
11.8.1	The ViewController interface.....	124
11.8.2	Setting the application properties.....	124
11.8.3	The state machine events.....	124
11.9	A SWING APPLICATION EXAMPLE	125
12	Validating Input Requirements.....	133
12.1	DEFINE THE VALIDATION RULES	133
12.2	HOW THE STATE MACHINE HANDLES THE VALIDATION CHECK.....	134
13	Generating JSP and Swing Panels	137
13.1	RUNNING THE GENERATOR	137
14	MCA Services Timing Points.....	139

1 Overview

The Screen Orchestrator is a tool that allows a user to design and implement an application using statechart principles. It allows the user to visually draw a statechart representation of their proposed application and interactively allows the user to specify the actual processes and state types that will be used by the application when run. Finally it allows the actual visually drawn statechart to be deployed on a live HTTP or application server. There the application can be run and controlled by the state machine. The state machine reads the deployed statechart and uses it to control the actual application.

Understanding statechart principles and notation is a prerequisite to using the orchestrator tool correctly. It is extremely important that the following subsections are read in order to get a basic understanding of statecharts, their notation and the state machine, as these are the basic premises on which the tool operates.

2 Statechart and State Machine Concepts

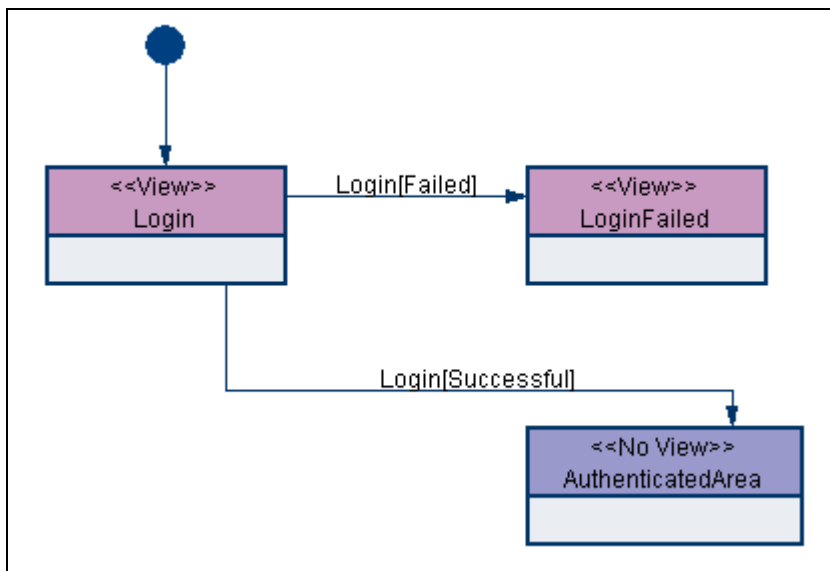
2.1 What Are Statecharts?

The UML definition of a statechart diagram is as follows:

Statechart diagrams represent the behaviour of entities capable of dynamic behaviour by specifying its response to the receipt of event instances. Typically, it is used for describing the behaviour of class instances, but statecharts may also describe the behaviour of other entities such as use-cases, actors, subsystems, operations, or methods. (from www.omg.org)

For us, the key concept in understanding why we use statecharts to represent user interfaces is the fact that statechart diagrams are capable of handling or modelling dynamic behaviour through events. Users interact with a user interface dynamically through events. Statecharts are therefore ideally suited to describing how a user interacts with a user interface.

For example, consider the following user interaction with a login screen in a user interface. To login the user must first enter their username and password. The user must click the login button to activate the login request. If the login is successful then the user is allowed into the rest of the system. If the login fails then the user is taken to the login-failed screen. The clicking of the login button is an event that must occur for the request to be processed. The event has two possible outcomes in this instance, it is either successful or it fails. The statechart representation of this user interaction is shown in the following figure (the statechart notation will be explained in a later section).



Login statechart diagram.

The statechart in the figure shows how a login user interface can be modelled in statechart notation. When the user interacts with a screen an event is created. An event can be created when a user presses a button,

or checks a radio button, or submits a form, or for whatever action the designer may wish the user to take. States in terms of the tool are often what the user sees on the computer screen (the view). In the login example, we have three states or screens, the login screen itself, the loginFailed screen and the AuthenticatedArea, where one or more screens and hence states, may exist.

Statecharts, in terms of the orchestrator tool, are used therefore to capture the user's interaction with the user interface through the modelling of the events that describe that system. Statecharts can also for other purposes. For example, it can be used to capture the flow of a process on the server-side of the application. It can be used in long-lived multiple transitions to route a process from one state of the transition to the next. This user guide will concentrate primarily with using statecharts for the design and implementation of user interfaces.

2.2 What Is The State Machine?

While a statechart is the representation of the modelled user interaction of a user interface, the state machine is a framework that allows that statechart to be actually used to control the real user interface. With the state machine framework, a statechart drawn by the orchestrator tool can be used to directly control a real user interface. The state machine framework is based upon an open source project, the jstatemachine (www.jstatemachine.org). This framework has been extended to be aware of Siebel Retail Finance processes and is part of MCA Services. The state machine framework is loaded and runs on any HTTP server that supports Java servlets and Java Server Pages (JSPs). The state machine reads the statecharts produced by the orchestrator tool and uses that chart to control the real user interaction coming from the user interface. The key thing to remember here is that the tool constructs the statecharts, while the state machine loads that statechart and uses it to control the actual user interface.

2.3 Why Use Statecharts And The State Machine?

Any large system's user interface today is usually designed using a modern integrated development environment (or IDE). While these tools are extremely powerful in building complex user interfaces, ¹user interface software often has the following characteristics:

- The code can be difficult to understand and review thoroughly
- The code can be difficult to test in a systemic and thorough way
- The code can contain bugs even after extensive testing and bug fixing
- The code can be difficult to enhance without introducing unwanted side effects
- The quality of the code tends to deteriorate as enhancements are made to it

Despite the obvious problems associated with user interface development, little effort has been made to improve the situation. The use of statecharts to specify the flow and control of the user interface is a major step in improving this situation. The user interface design can now be captured and easily understood, and can be interpreted by existing and new users developing the system. The user now has the visual record of the flow of control of the system. They can also see the side effects any change on the user interface will have. Statecharts and their notation are ideally suited to this.

¹ Constructing the User Interface with Statecharts by Ian Horrocks

The design of the user interface is too often left almost entirely to the developer and their understanding of the use cases. Greater design work needs to be done on the user interface in order that the user interface can be more easily understood, developed and maintained. Statecharts can play a key part in achieving this.

An extension of using statecharts is the state machine. If the user interface can be described using a statechart, then why can't the actual statechart be used within the user interface application to maintain control of the actual system? Any changes in the statechart can then be automatically reflected back in the actual application. This is precisely what the state machine does. It takes the actual specified statechart and controls the application directly with it. The user interface developer is then free to concentrate on building and creating the views for the system.

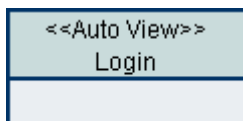
Statecharts and the state machine enforce the model-view-controller (MVC) programming model. The controller of the user interface is the statechart that was drawn, while the state machine is the runtime environment for that chart. The developer can now create views for the application that will contain view code only. The model is defined by the input parameters to the states, events and transitions (later sections will describe what is meant by input parameters and the maintaining of the model details used in the statecharts). Additionally the state machine and the statechart drawing tool are aware of Siebel Retail Finance processes. They can invoke them and more importantly interpret their responses so that zero or minimum control code is required to be written by the user. This leads to a user interface system that is highly controlled and whose side effects can be easily understood and changed as the system grows and enhancements are added, the direct opposite to the normal problems associated user interface designs.

2.4 Statechart Notation Explained

Statechart notation is essentially broken down into two representations: states and state transitions. States are represented in the Orchestrator by a rectangular box, while an arrow represents state transitions. Events, another important statechart element, are not pictorially represented. Events can however be identified in a statechart by examining the labels attached to state transitions.

2.4.1 States

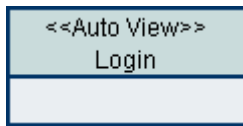
The statechart that the user defines for their user interface is merely the representation of the possible states of the interface. A state, in the case of the state machine and the orchestrator, represents what the user sees on the computer screen. In terms of the orchestrator a screen is termed a view. A standard state drawn on a statechart in the orchestrator is shown in the following figure.



A standard state.

Every state has a title and subtitle. In the case shown in the above figure the text "<<Auto View>>" is a title, and the text "Login" is a subtitle. The subtitle on each state indicates the name of the state or view. The title

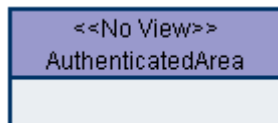
always indicates the type of state that the state represents. The orchestrator provides three basic state types, an “Auto View”, a “View” and a “No View”. Each state type available in the orchestrator tool is shown in the figures below.



An ‘Auto View’ state.



A ‘View’ state.

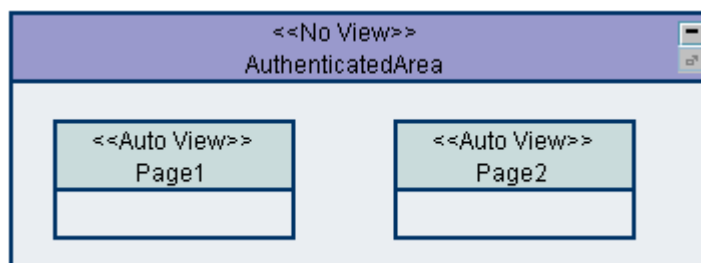


A ‘No View’ state.

The states as shown in the previous figures are colour coded for easy identification. An “Auto View” state indicates that the user is representing a view by the state but that no current view is available to be attached to the state. The state machine can generate an automatic view for this state dependent on the state input parameters and the events leaving the state. The “View” state indicates that the user is representing a view by the state and that the user has an actual view that can be attached to the state. When the state machine runs the application, the attached view will be displayed to the user. The “No View” state is often used to indicate that the state will be a parent state (although an AutoView and View state can also be parent states). Parent states can be used to split the user interface into subsystems. The “No View” state can also be used to represent server-side states as these states do not represent any particular view of the system.

2.4.1.1 Parent and child states

States can be added to other parent states. Once a state is added to another state, that state becomes a child of the enclosing state. The following figure shows the AuthenticatedArea state as a parent state of the child states Page1 and Page2.



Parent and child states

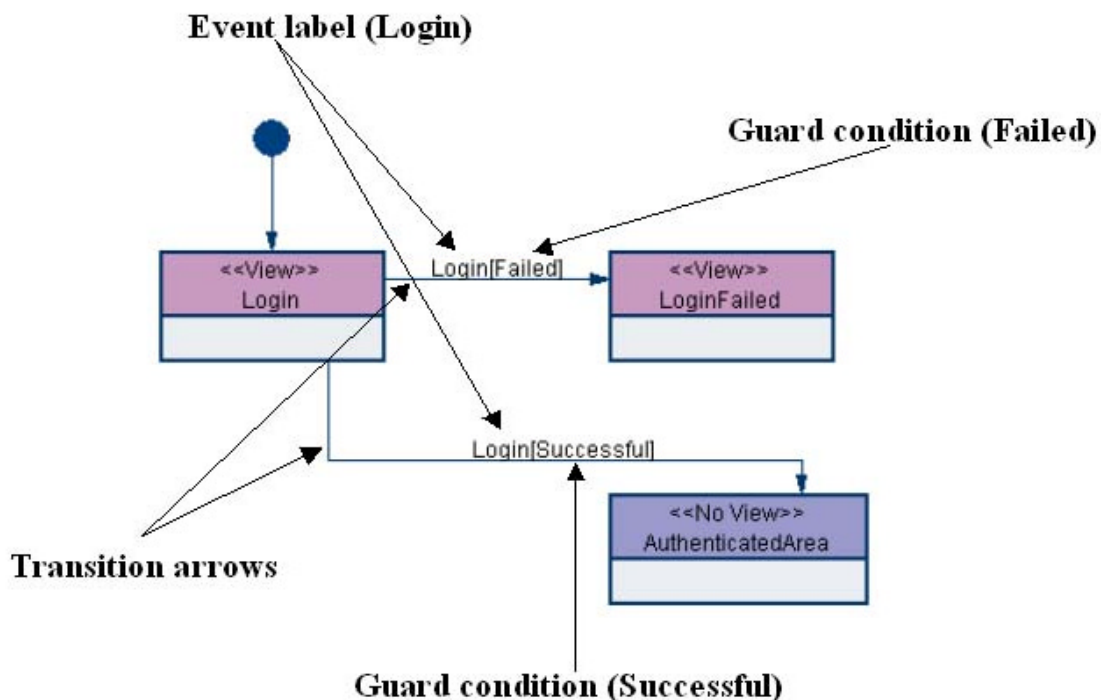
States can inherit events and transitions from their parent states. That is an event or transition available from the parent state is also available from its child states, with the exception of the case where a child state has an event of the same name.

2.4.2 Events

When the user interacts with the screen an event is initiated. They may be pressing a button, checking off a radio button, submitting a form, or any action the designer may wish the user to take. An event is identified in the state machine by its source and name. In the earlier example, clicking the login button on the login screen is an event being initiated.

2.4.3 State Transitions

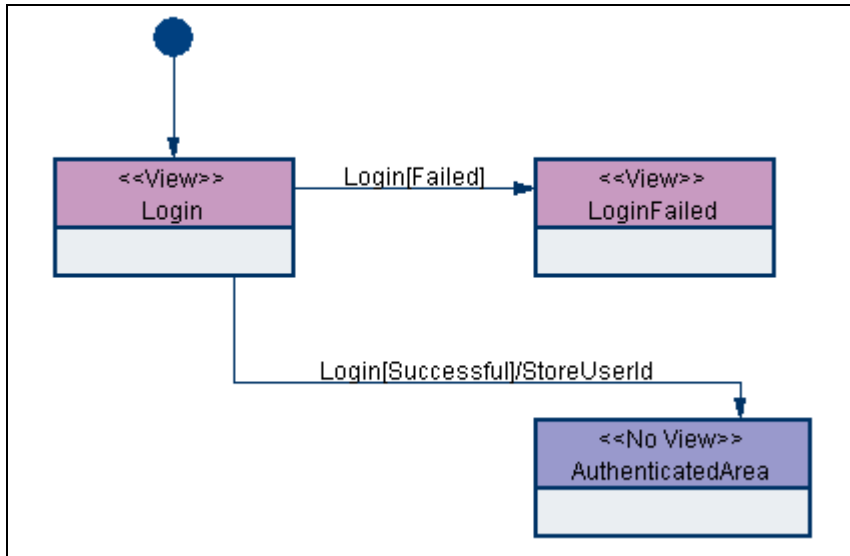
Arrows represent state transitions on a statechart. Using the name of the event the transition arrow on the statechart can be followed to one and only one logical endpoint, i.e. another state. Not all states are reachable at once. Each transition is guarded by a condition or set of conditions, mutually exclusive that must return true for a particular transition to be followed. After the event occurs the guard condition of each transition possible for that event will be tested. One of the conditions will return true and the state machine will follow that transition to the resultant state. Having attained that state the state machine will then inform the user interface and the display will be updated to show what is proper for that state. The next figure indicates the login event and its transitions.



Identifying events and transitions on a statechart.

The syntax for a transition arrow's label has three parts, all of which are optional: *Event [Guard] / Action*.

Actions are associated with transitions and are considered to be processes that occur as a result of a transition. Actions are also termed as “side effects”. When a transition occurs an action or side effect may result. In the login example we could extend the *Login[Successful]* transition to include an action to store the user’s *userId* in the user’s session.

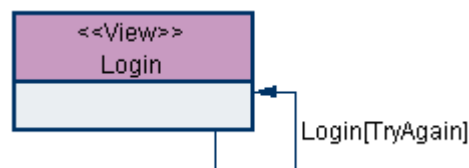


Extending the login example to include a transition action.

When the login button is then pressed the login event is fired. The login results will be tested to see if the login was successful. If the login was successful then the *Login[Successful]/StoreUserId* transition will be followed. The action associated with this transition is to take the user’s *userId* and store it in the user’s http session.

2.4.3.1 Self-Transition

A self-transition behaves exactly as a normal transition would except that its start state and end state are the same. The figure that follows indicates how a self-transition is drawn on the orchestrator tool.



Self Transition

A self-transition is used when you wish to send the user back to the currently displayed screen if a certain guard condition is met. In the example in the figure, when the *Login* event is fired, if the guard condition is “*TryAgain*” then the *Login* screen is re-displayed by the state machine.

2.4.4 Pseudo-States

A number of pseudo-states are also available within statechart notation. Pseudo-states represent special types of states that indicate very specific types of behavior when included on the statechart.

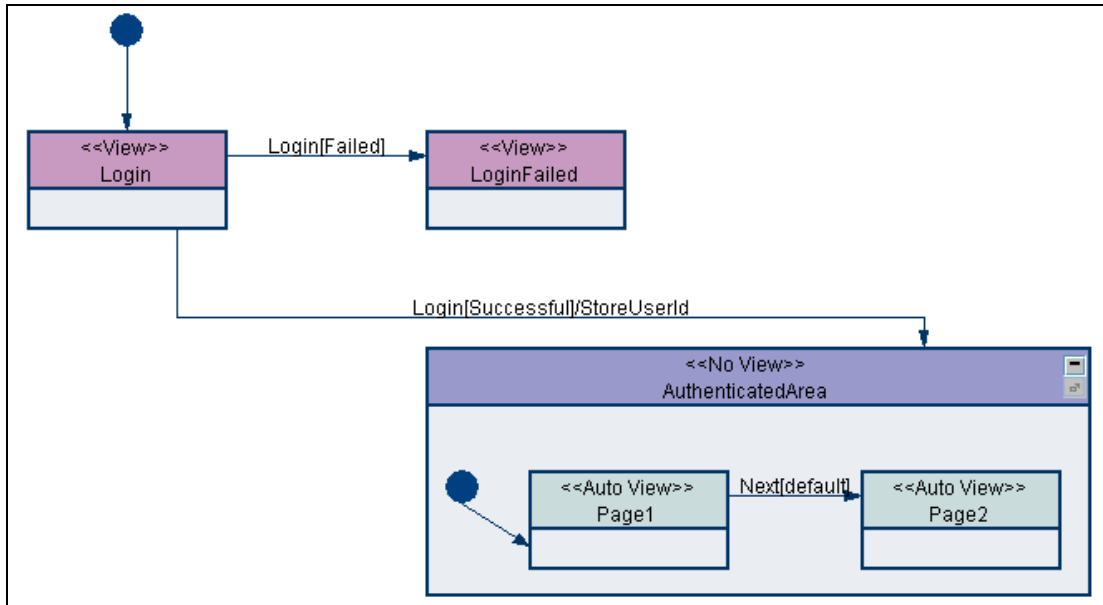
2.4.4.1 Initial State

A solid circle as shown in the next figure represents an initial pseudo-state.



An initial pseudo-state.

An initial pseudo-state indicates the starting point or state for a statechart. Initial pseudo-states can also be used in parent states to indicate when a user enters a parent state where the starting point is within that parent state. The next figure shows the login example extended to include child states in the AuthenticatedArea parent state.



Using initial pseudostates.

The statechart has two initial pseudo-states. One to indicate where the application starts and the second to indicate which state is displayed first when the AuthenticatedArea state is entered (e.g. Page1).

2.4.4.2 History

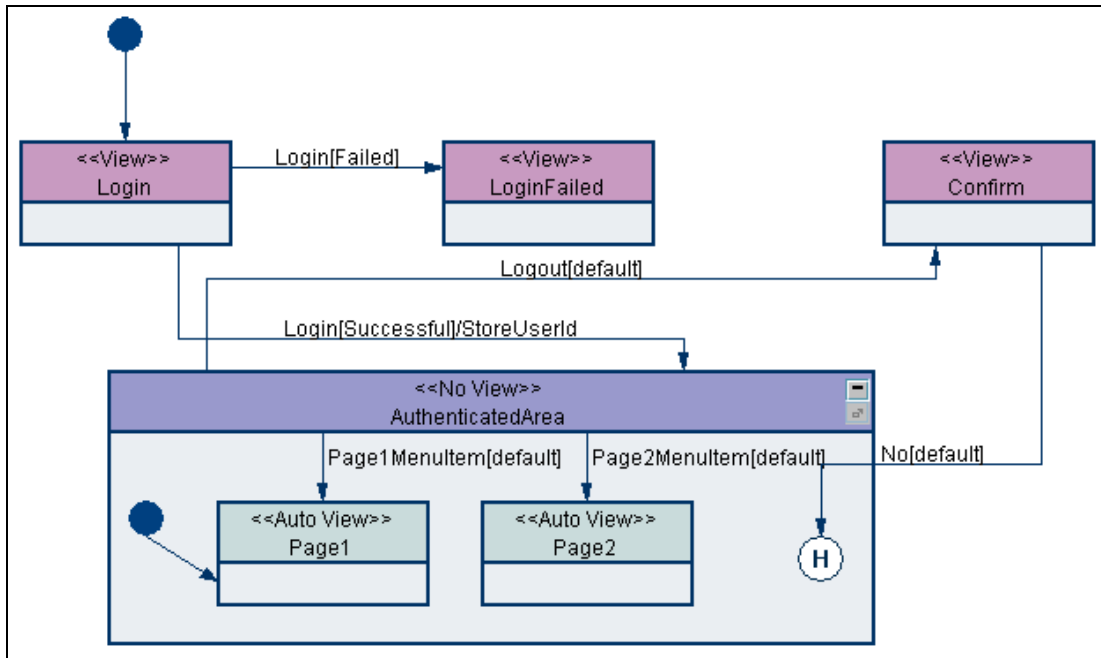
A circle enclosing a "H" as shown in the next figure represents a history pseudo-state.



A history pseudo-state.

A history pseudo-state refers to children of a state that have recently been visited by the user. The history

pseudo-state allows the user to return to the state that was the most recently visited immediate child of the history pseudo-state. The following figure shows a history pseudo-state in the AuthenticatedArea.



The login example extended to use a history pseudo-state.

In this example, if the user cancels their logout event while on the confirm state, the user is returned to the last previously opened state within the Authenticated area (i.e. either Page1 or Page2, depending on which of the two screens have been displayed before the Confirm state was displayed).

2.4.4.3 History-Star

A circle enclosing a “H*” as shown in the next figure represents a history-star pseudo-state.



A history-star pseudo-state.

A history-star pseudo-state is very similar to the history pseudo-state. However, the history-star pseudo-state will recursively return the user to the immediate child of the history-star pseudo-state ending with the deepest visited state.

For example, if the statechart in the previous figure was modified to use history-star rather than history, the state machine would allow the user to return to the last displayed state and its deepest visited state. So if the user visited Page1 followed by a child state of Page1 the user will be returned to the child state of Page1.

2.4.4.4 Final State

A solid circle enclosed by an outer circle as shown in the next figure represents a final pseudo-state.



A final pseudo-state.

The final pseudo-state is used to indicate the final activity allowed on a statechart. It triggers a transition for leaving the application fired from the connected state to the final pseudo-state.

2.4.4.5 Exception State

A circle with an “X” across it as shown in the next figure represents an exception pseudo-state.



An exception pseudo-state.

An exception pseudo-state is not part of the standard UML statechart notation. It was added to the state machine for exception handling. The state machine recognises that exceptions can occur during event processing, and allows you to specify states within your user interface as exception states. When an exception is thrown the user interface will be placed properly in the appropriate exception state, with the user session still intact.

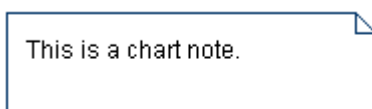
2.4.4.6 Effects of Pseudo-States On State Transitions

Pseudo-States indicate very specific behaviour when used on a statechart. Similarly, pseudo-states have effects on state transitions that you should be aware of. Some pseudo-states can start a state transition but can't be used to end a state transition. While other pseudo-states can't be used to start a state transition but can be used to end a state transition. The table that follows indicates the type of behaviors that are allowed by pseudo-states when drawing a state transition with them on a statechart.

Pseudo-state	State Transition Behavior	
	Start state in a transition	End state in a transition
Initial	Yes	No
History	No	Yes
History-star	No	Yes
Final	No	Yes
Exception	No	Yes

2.4.5 Chart Notes

The orchestrator allows various notes to be added to the statechart. One visual note type is the chart note. The following figure shows a chart note.

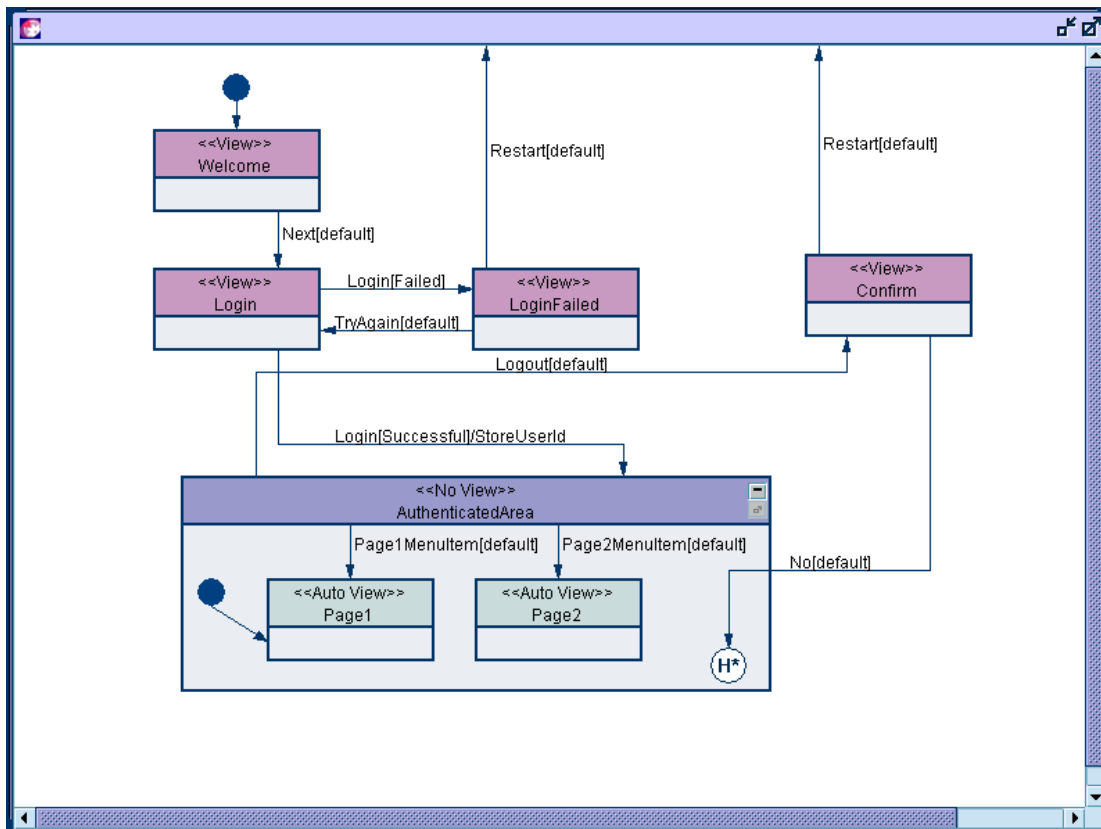


A chart note.

Chart notes can be added anywhere within a statechart and can be used to visually comment the statechart. The state machine makes no use of these notes. They are only used to explain the statechart to other users.

2.5 Simple Statechart Example

In the previous sections we have used a login example to demonstrate the statechart notation. We will extend that example to produce the statechart as shown in the following figure. The purpose of this section is to explain what we mean by this statechart and how a user should interpret it. Later sections introduce more advanced features that the orchestrator is capable of, for now we will concern ourselves only with the basics.



A simple statechart example.

The figure above provides a simple statechart that represents a user interface that allows a user to login to an authenticated area, move around that area and then to restart the application by logging out of the application. Initially, when the user starts the application, the welcome view will be displayed as indicated by the statechart's initial pseudo-state. On the welcome screen or view is a next button. When the next button is pressed the state machine will display the login view to the user. On the login view the user can enter their username and password. When the user clicks the login button a login event is fired. The state machine handles the result of the login and determines which guard condition is met.

If the guard condition *Failed* is met the state machine follows the *Login[Failed]* transition and displays the

LoginFailed screen to the user. On the LoginFailed view is a restart button and a try again button. If the user clicks the try again button then the state machine displays the Login screen again. However, if the user clicks the restart button the state machine displays the welcome screen.

If we go back to the login guard condition and the guard condition *Successful* is met, the state machine then follows the *Login[Successful]/StoreUserId* transition. When the guard condition *Successful* is met the state machine performs the StoreUserId action or side effect. The StoreUserId side effect requires the state machine to take the userId returned from the login process and to store that userId in the user's http session. Once the side effect has been completed the state machine will enter the AuthenticatedArea of the application and will display the Page1 view to the user as indicated by the initial pseudo-state in the AuthenticatedArea.

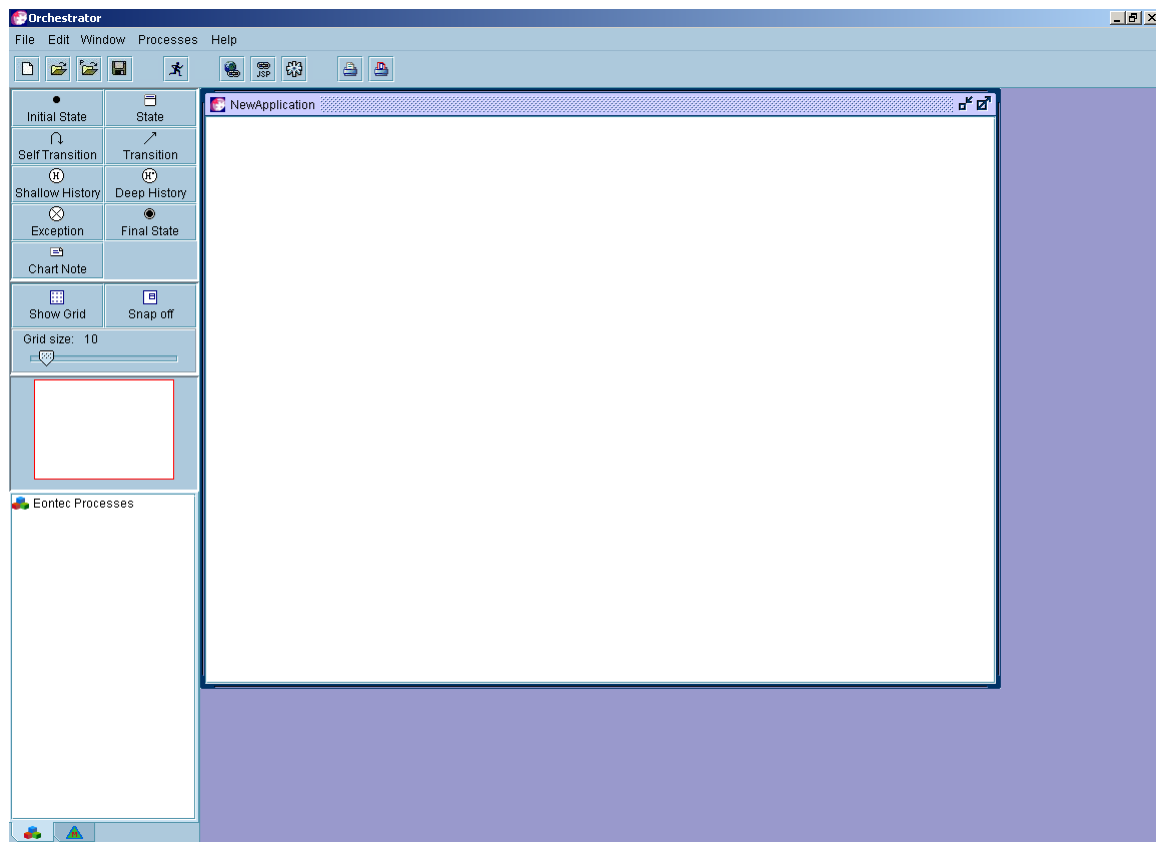
The user can move between Page1 and Page2 in the AuthenticatedArea by menu or form buttons on the AuthenticatedArea screens. The AuthenticatedArea indicates that there is a Logout event. This event is inherited by the AuthenticatedArea's child states. This implies that both the Page1 and Page2 screens will have a logout button so that the user can fire the Logout event. Note how the statechart does not require a logout state transition arrow to be drawn from Page1 and Page2 to the Confirm state. They automatically inherit the event by being child states of the AuthenticatedArea.

3 Basic Orchestrator Drawing

This section will introduce the basic drawing capabilities of the orchestrator tool. After completion of this section you should be able to build a standard statechart with the tool.

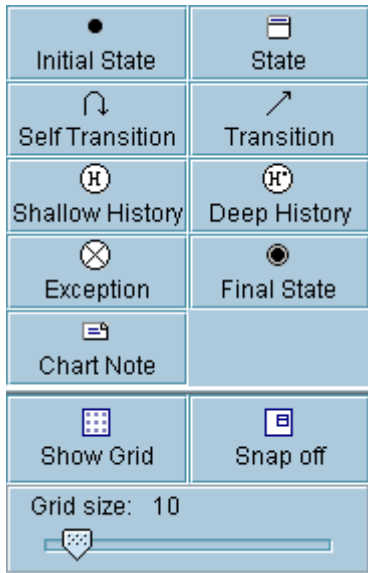
3.1 The Main Orchestrator Window

When the orchestrator window starts, the screen in the following figure will be displayed:



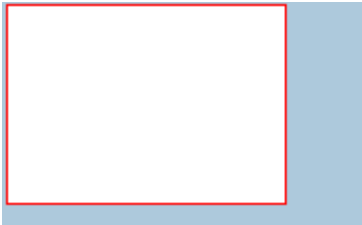
The orchestrator window.

The orchestrator application provides a menu and toolbar to access the major functions of the tool. The main drawing components of the tool with which statecharts are drawn are always displayed in the upper left hand-side of the application. The next figure indicates the statechart drawing components used by the tool.



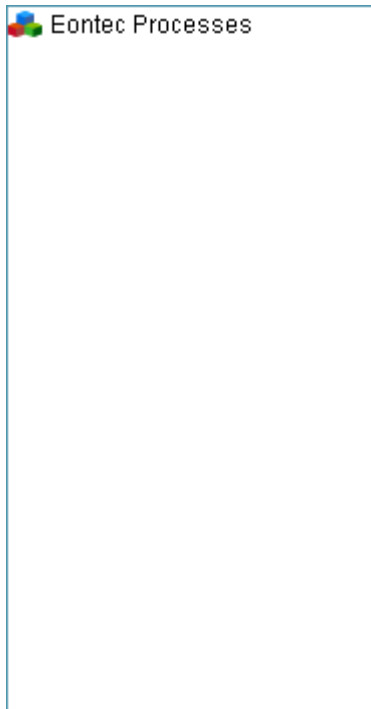
Drawing components used in creating statecharts.

Below the drawing component's panel on the main window is a navigation window that allows the user to see a miniature view of the statechart they are currently drawing. This window can be used to navigate a statechart to a particular area within the currently visible statechart window.



Navigate window.

Below the navigate window panel is a further panel, the "Siebel Processes" panel. This panel is used to display the available list of Siebel processes available to the user to use when populating a statechart with process information.



Siebel processes panel.

Beside the “Siebel Processes” is a further panel, the “Application Metrics” panel. This panel is used to display the number of states, view states, autoviews, events, complex controllers, transitions and complex guard conditions in a statechart. This information can be used to determine the complexity of the statechart, to help with task estimation and to track progress of the application i.e. the number of autoviews left to be coded to actual views.












Application Metrics panel.

As stated previously this section will concentrate on the basic drawing of statecharts using the tool, later sections will explain fully the use of the process panel.

Finally, to the right of these panels is the main drawing area. The orchestrator tool provides an MDI (multiple document interface) desktop in which statecharts can be drawn. The tool allows a user to create or edit a single statechart at a time. However, larger parent states can be opened into smaller sub-chart windows within the desktop. Opening parent states as sub-charts will be explained in detail in the section titled “Advanced Statechart Drawing”.

3.2 The Statechart Drawing Components

The statechart drawing components as shown in the next figure represent the statechart notation as described in the statechart concepts section of the user guide.

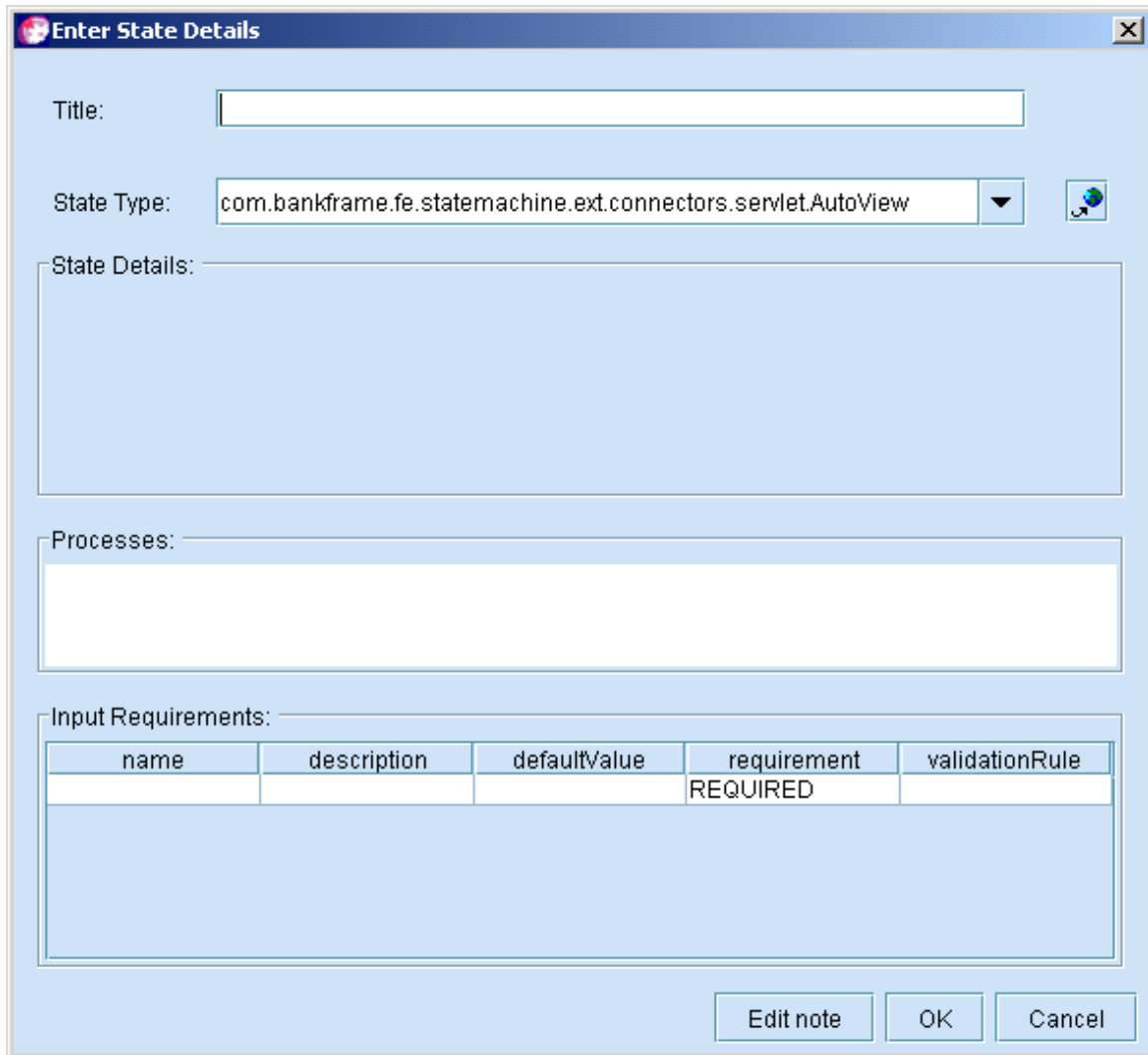
 Initial State	 State
 Self Transition	 Transition
 Shallow History	 Deep History
 Exception	 Final State
 Chart Note	

Statechart notation components palette used for drawing statecharts.

3.2.1 Drawing States or Pseudo-states

To create a state or a pseudo-state on the statechart window the user must click on the state required from the component palette with the mouse and drag the state using the mouse onto the statechart window. The user must release the mouse button over an area in the statechart window where they wish to drop the state. Once released a state or pseudo-state will be created by the tool on the dropped location in the statechart.

Once a state (not a pseudo-state) is dropped and created the tool will display the “Enter State Details” dialog. The user can then enter the state’s details. The following figure shows the “Enter State Details” dialog.



The dialog box is titled "Enter State Details" and contains the following fields and controls:

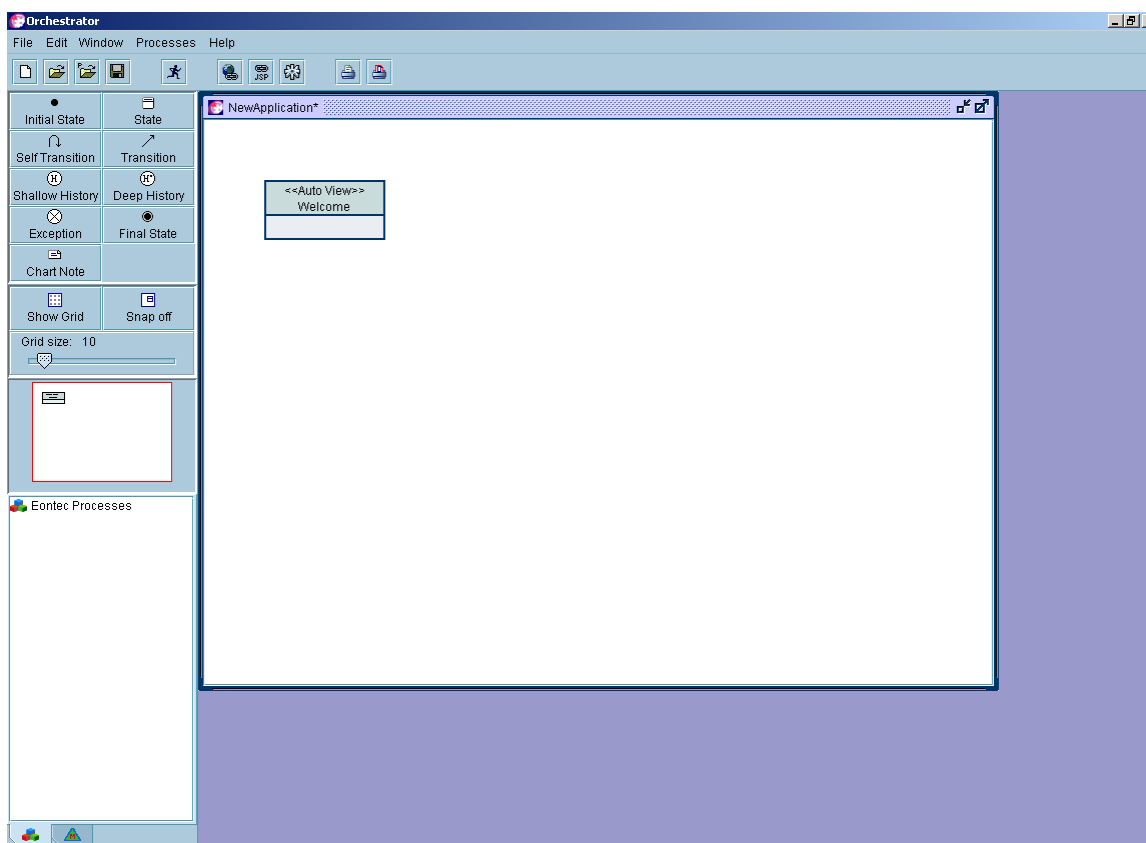
- Title:** A text input field.
- State Type:** A dropdown menu showing "com.bankframe.fe.statemachine.ext.connectors.servlet.AutoView" with a small globe icon to its right.
- State Details:** A large, empty text area.
- Processes:** A large, empty text area.
- Input Requirements:** A table with the following structure:

name	description	defaultValue	requirement	validationRule
			REQUIRED	

At the bottom right, there are three buttons: "Edit note", "OK", and "Cancel".

Enter state details dialog.

For now we will leave the details as is and only enter a title for the state. In this case we will enter a title "Welcome" for the state. When the "OK" button is pressed the dialog will be closed and the state will be displayed with the appropriate title and state type. The next figure indicates the state that would be drawn by following the previous related procedure.

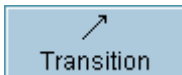


Welcome state drawn.

The “Enter State Details” Dialog will only be displayed when creating or editing states. Pseudo-states do not require titles or require to be configured in any way. The state or pseudo-state can be moved at any time by clicking with the mouse on the state and dragging the state to any new location within the statechart window.

3.2.2 Drawing State Transitions

To connect two states together to represent a state transition the user must click with the mouse on the transition component and then drag the component onto the statechart and drop or release the mouse button directly over the ²header of the state, or anywhere over a pseudo-state that will be the starting state of a state transition.



State transition component.

The tool will immediately change the cursor to that of a transition crosshair – as shown below.

² To understand what the header of a state means, please see the later subsection entitled “State header”.



Transition crosshair.

The user must then move the mouse to the state or pseudo-state that will be the endpoint of the transition. As the mouse is moved about the statechart towards the end state of the transition a temporary state transition line will be drawn for each mouse movement towards the end state. Once the mouse is over the header of the state or anywhere over a pseudo-state the user must click the mouse on the state. The cursor will be returned to the default cursor and the state transition will be drawn. The “Transition Wizard” dialog as shown in the next figure will then be displayed.

Transition Wizard

Enter Transition's Event Details

*Event:

Controller:

Controller Properties:

☐ Validate event's input requirements?

InputRequirements:

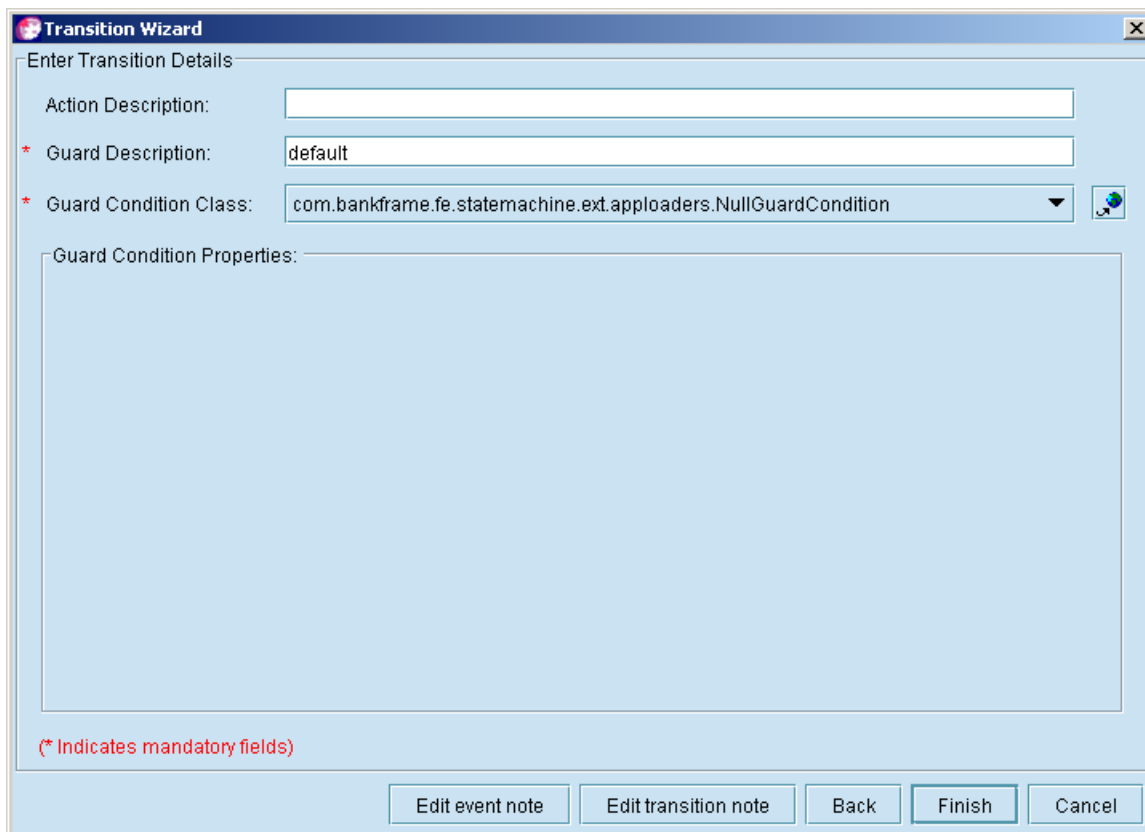
name	description	defaultValue	requirement	validationRule
			REQUIRED	

(*) Indicates mandatory fields

Edit event note Edit transition note Next Cancel

Transition Wizard dialog.

The user must enter an event name of the transition or select an existing event on the starting state of the transition if one already exists. For example, in the login example used in the statechart concepts section, if we were drawing a transition from the welcome state to the login state, then the next event would be created for this transition. Enter the event name, in this case “Next”, into the combo box for the event name. At this stage we will only consider the basic drawing functions of the statechart and so for now we will ignore the other features on the dialog. Press the “Next” button to move the wizard onto the next screen.

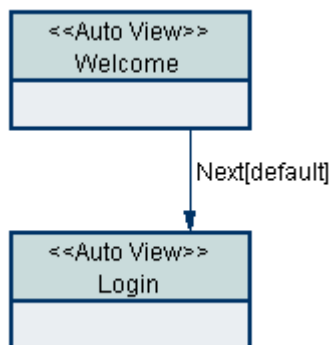


The image shows a 'Transition Wizard' dialog box with the following fields and controls:

- Enter Transition Details** (Section Header)
- Action Description:** (Text input field)
- * Guard Description:** (Text input field containing 'default')
- * Guard Condition Class:** (Dropdown menu showing 'com.bankframe.fe.statemachine.ext.apploaders.NullGuardCondition')
- Guard Condition Properties:** (Large empty text area)
- (* Indicates mandatory fields)** (Red text note)
- Buttons:** 'Edit event note', 'Edit transition note', 'Back', 'Finish', and 'Cancel'.

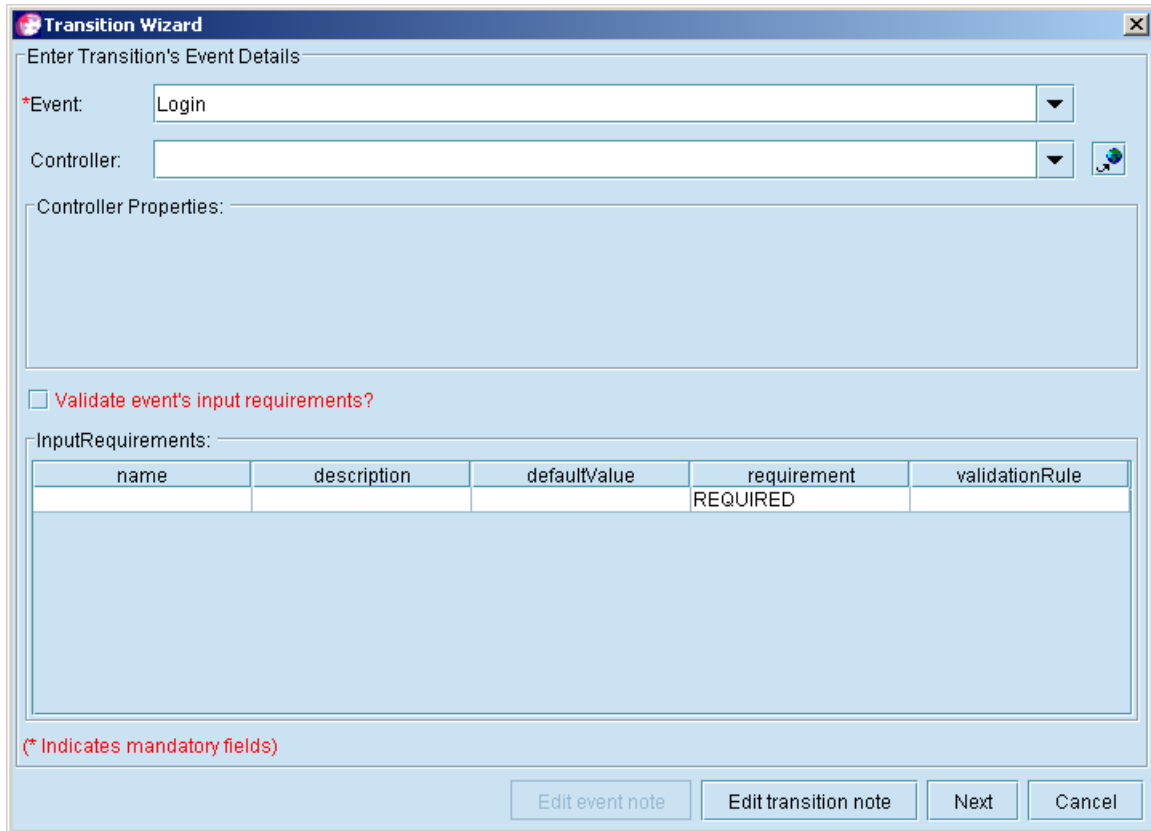
The next screen on the transition wizard dialog.

On this screen in the transition wizard dialog, the user can enter an action description and guard condition for the state transition. In the state transition from the Welcome state to Login state no action or guard condition exists. Again, later sections will describe in more detail the remaining properties of this dialog. Once the “Finish” button is pressed, the dialog is closed and the state transition entitled Next[default] is created as shown in the next figure.



The Next[default] state transition drawn.

If we were drawing the Login[Successful]/StoreUserId state transition then the following information as shown in the figures below would be entered in the transition wizard dialog screens.



The Transition Wizard dialog box is used for configuring transition details. It includes fields for Event and Controller, a section for Controller Properties, a checkbox for validating input requirements, and a table for input requirements. The Event field is currently set to 'Login'.

Transition Wizard

Enter Transition's Event Details

*Event:

Controller:

Controller Properties:

☐ Validate event's input requirements?

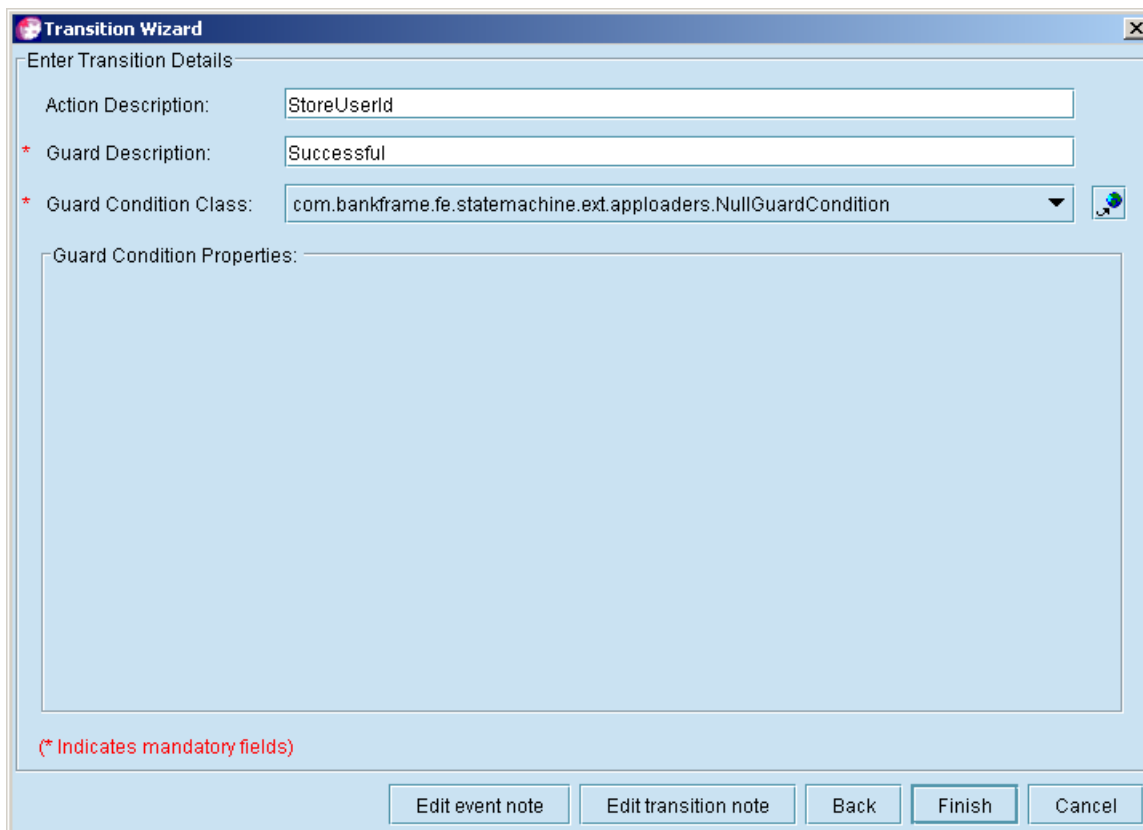
InputRequirements:

name	description	defaultValue	requirement	validationRule
			REQUIRED	

(* Indicates mandatory fields)

Edit event note Edit transition note Next Cancel

Entering the Login event name.



Transition Wizard

Enter Transition Details

Action Description:

* Guard Description:

* Guard Condition Class:

Guard Condition Properties:

(* Indicates mandatory fields)

Buttons: Edit event note, Edit transition note, Back, Finish, Cancel

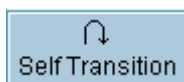
Entering the Login StoreUserId action and Successful guard condition.



The Login[Successful]/StoreUserId state transition

3.2.2.1 Drawing Self-Transitions

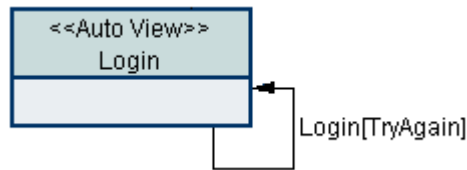
To draw a self-transition, the user only needs to mouse click on the self-transition component and holding the mouse button down, drag the component onto a state header. Pseudo-states do not support self-transitions and will be ignored. The user will be warned if they attempt to drop a self-transition on a pseudo-state.



The self-transition component.

Releasing the mouse button over a state will invoke the tool to draw a self-transition around the state and to display the "Transition Wizard" dialog as described in the previous section. Remember a self-transition has exactly the same properties as a state transition, except its start state and end state are the same. The next figure shows a self-transition drawn around a Login state, when the Login event's guard condition is

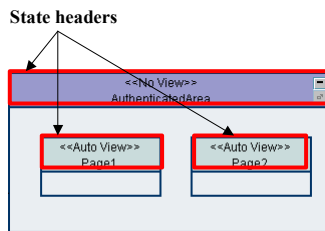
TryAgain. If the login fails, the state machine can re-display the Login state allowing the user to try logging in again.



The Login[TryAgain] self-transition.

3.2.2.2 State Header

When dropping a transition or self-transition on a state you must drop the drawing component on the state's header. Similarly, when specifying the end state of a state transition, the user must click the mouse on the state's header for the tool to identify the state that should be used for the end state of that transition. The state header is the top rectangular box of the state where the state's type and name (or title) is displayed. The following figure highlights the header area of a number of states.



The state header area.

The bottom rectangular box of a state is where child states can be added to the state. Adding child states to a parent state will be discussed in a later section.

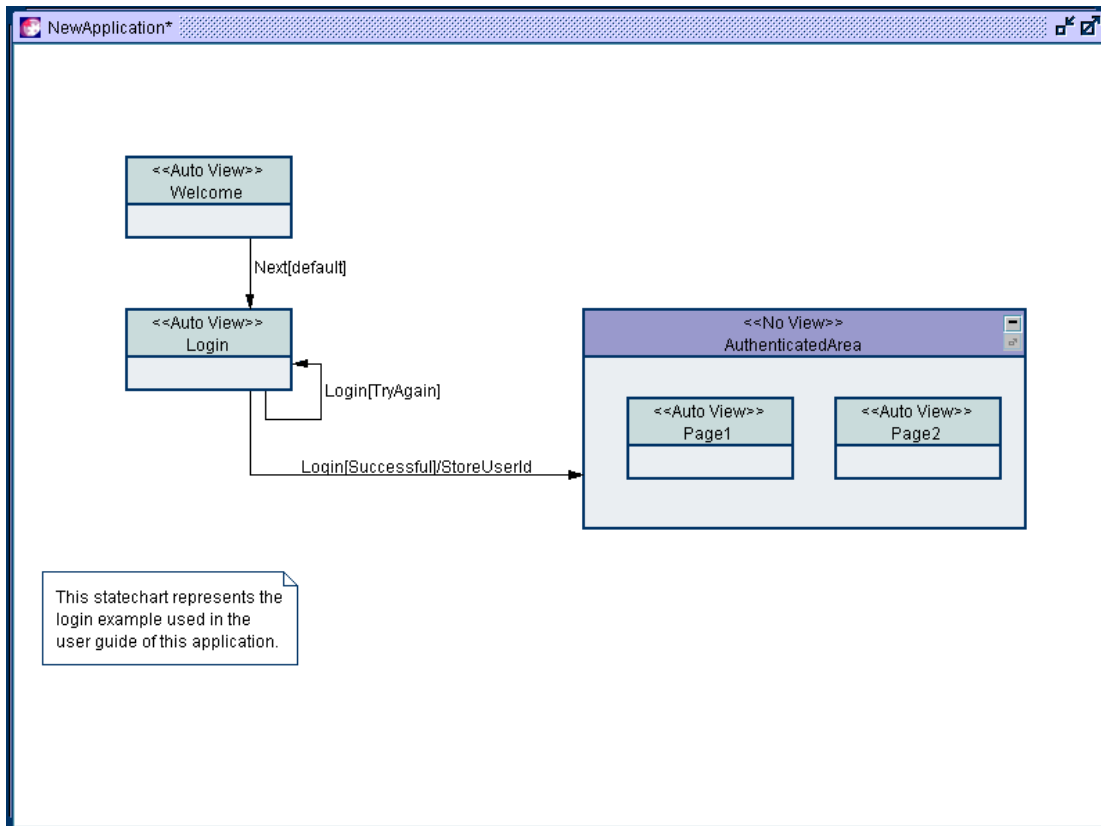
3.2.3 Drawing Chart Notes

Chart notes can be added anywhere to the statechart. The statechart component is shown in the following figure.



Chart note component.

To add a chart note component to the statechart click the mouse button on the component shown in the previous figure and holding the mouse down, drag the component onto the statechart. Release the mouse button where you want the chart note to be located. Click the mouse into the chart note and the user can then type text directly into the note. The note will grow automatically in size as more text is typed. A chart note is shown in the following figure. The statechart can have as many notes as you wish and they can be added to states just as any child state can be added to any parent state.



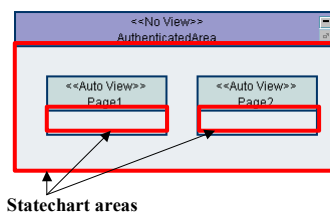
A statechart showing a chart note attached.

3.3 More Drawing State Details

States and pseudo-states can be created or moved anywhere within other states. They can be resized and their details edited - e.g. their names can be changed and state types can be altered. The following subsections indicate how these features can be accessed.

3.3.1 Adding Child States

To add a state or pseudo-state as a child to a parent state, simply drag and drop a state component from the component palette onto the parent state's statechart area. Every state has a statechart area on which child states can be added. The next figure indicates the statechart area available for each state.



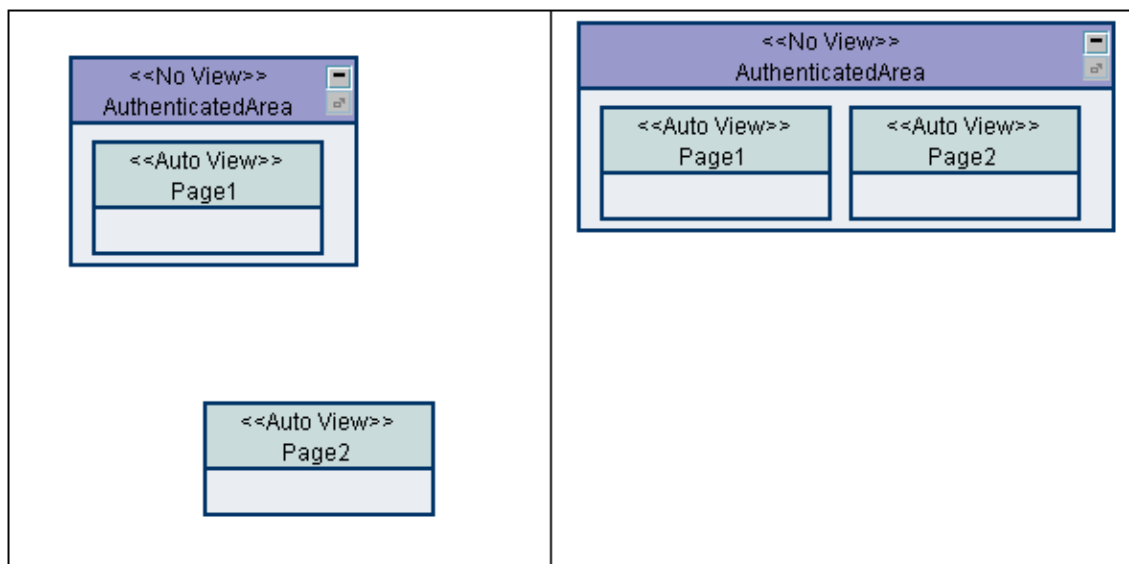
Statechart areas attached to states.

Once a state is dropped onto a parent state's statechart area, the parent state will be resized to fit the child

state and any other child states of the parent. If the child state is then moved further around the parent state, the parent state will continue to be resized if required.

3.3.2 Moving States

Once a state or pseudo-state has been dropped and created, the user is free to move the state to any location in the statechart. To move any state simply mouse click anywhere on the state's header or, if it is a pseudo-state, anywhere on the body of the pseudo-state, and drag the state to its new location. Releasing the mouse button over the new location will move the state. If the state is moved inside a parent state the parent state will be resized automatically if necessary. You can move a state anywhere you wish. If the state is outside a parent state but should be inside it, then dragging the state inside the parent state is allowed. The state will immediately become a child state of the parent state.

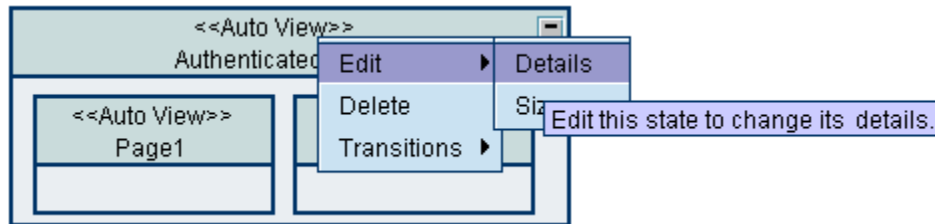


Moving Page2 state into the AuthenticatedArea state.

Similarly the reverse is also true. If a state has been initially located as a child state of some parent state, moving the child state anywhere outside the parent state will immediately release the state as a child state of the parent state.

3.3.3 Editing State Details

The state details, such as changing the state's title, view type, view details or input parameters, can be changed at any time. Right-clicking anywhere on a state's header will bring up the state's popup menu - as shown in the following figure.



The AuthenticatedArea's popup menu.

Clicking the Edit -> Details menu item will bring up the state's "Enter State Details" dialog. Any of the state's details can then be changed using the dialog.

Enter State Details

Title:

State Type: ▼

State Details: (NONE)
 com.bankframe.fe.statemachine.ext.connectors.servlet.AutoView
 com.bankframe.fe.statemachine.ext.connectors.servlet.JSPView
 com.bankframe.fe.statemachine.ext.connectors.servlet.XSLTAutoView
 com.bankframe.fe.statemachine.ext.connectors.swing.AutoView
 com.bankframe.fe.statemachine.ext.connectors.swing.SwingView
 com.bankframe.fe.statemachine.ext.connectors.swing.XSLTSwingView

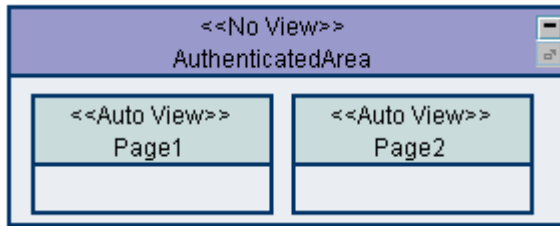
Processes:

Input Requirements:

name	description	defaultValue	requirement	validationRule
			REQUIRED	

Buttons: Edit note, OK, Cancel

Enter State Details dialog.

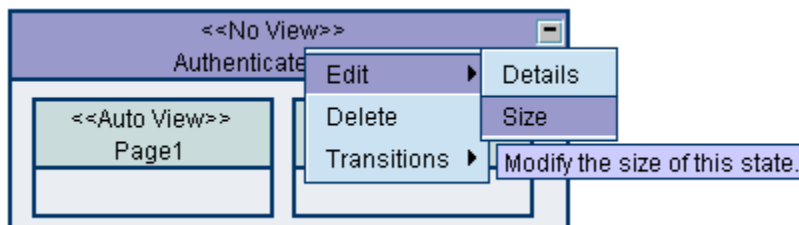


Changing the AuthenticatedArea's view type.

The previous figures show the AuthenticatedArea's state type being changed from an "AutoView" state type to a "No View" or (NONE) state type. Clicking the "Enter State Details" Ok button will save and update any changes made.

3.3.4 Resizing States

The states are automatically resized when adding child states or moving states within a parent state. However, the user can also specify a state's actual size by entering the state's width and height through the Edit -> Size menu item. Right-clicking anywhere on a state's header will bring up the popup menu for the state.



The AuthenticatedArea's Edit, Size menu item.

Clicking the Edit, Size menu item will bring up the "Enter State Size" dialog.

The 'Enter State Size' dialog box has a title bar with a close button. It contains a table with two columns: 'Width' and 'Height'. The table has three rows: 'Current size:', 'Best fit size:', and 'Enter size:'. The 'Current size:' row shows '281' for Width and '110' for Height. The 'Best fit size:' row shows '301' for Width and '110' for Height. The 'Enter size:' row shows '281' for Width and '110' for Height. There are 'OK' and 'Cancel' buttons at the bottom.

	Width	Height
Current size:	281	110
Best fit size:	301	110
Enter size:	281	110

The "Enter State Size" dialog.

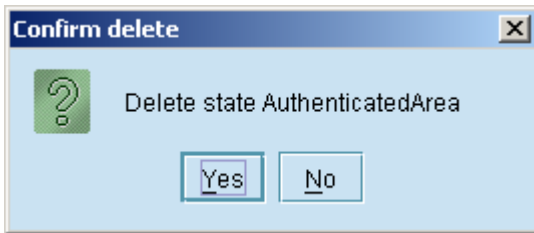
This dialog will indicate the current width and height of the state and indicate the best-fit size for the state. The user can enter the width and height they would like in the "Enter size:" text fields. Clicking the OK button will alter the size accordingly.

3.3.5 Deleting states

Any state or pseudo-state can be deleted by launching the popup menu for the state. If the user selects the Delete menu item, as shown below, then a confirm delete state dialog will be displayed.



The Delete menu item.



The confirm delete dialog.

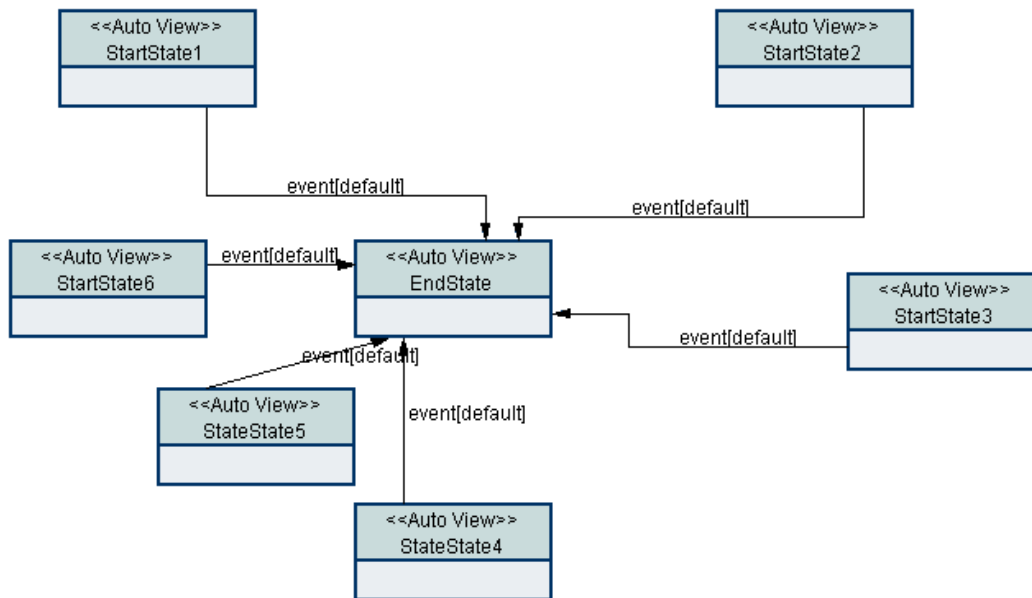
The important thing to remember when deleting a state, particularly if it's a parent state with child states, is that when you delete a parent state all its child states, transitions and child transitions will also be deleted.

3.4 More Drawing Transition Details

Transitions can be drawn between any two states, however only certain types of transitions can be drawn between states and pseudo-states. See the table – “indicating effects pseudo-states have on state transitions” in the statechart concepts section of this user guide to see what transitions are supported by pseudo-states.

3.4.1 Note On Transition Arrows

When a transition is drawn between any two states (regardless if any state is a pseudo-state), the tool will determine how the transition arrow is drawn between them. The user has no real control over how the transition arrow is drawn. However, moving the states that participate in the transition can alter how the transition arrow is drawn. For example, the figure that follows indicates the types of transition arrows that will be drawn given the location of one state in relation to the other state in the transition.



Types of transition arrows drawn by the tool.

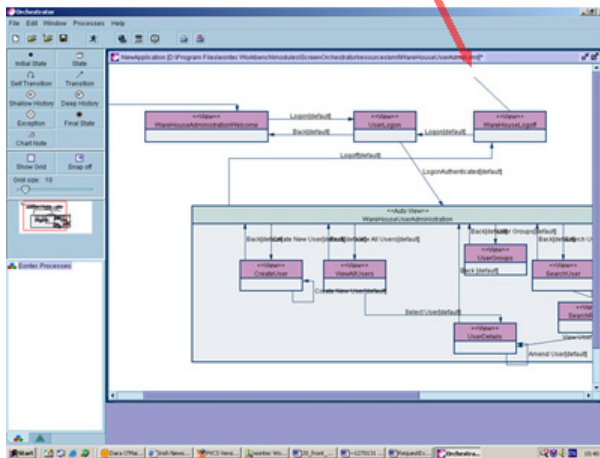
If you need to change a transition arrow and how it's drawn you will need to adjust the location of one of the transition's states. Moving one of the states involved in the transition further away or closer or at a different angle to the other state will force the transition arrow to be drawn differently.

3.4.2 Drawing Transitions To The Master State

The desktop window that displays the statechart is a state itself and as such can have transitions drawn to it. Within the tool this state is known as the master or application state. State transitions are often drawn to the master state to indicate that the application should be restarted. In the login example described in the simple statechart example from the statechart concepts section of the user guide, a state transition arrow is drawn from the LoginFailed state and the Confirm state to the master state to indicate that the application should be restarted as a result of these transitions.

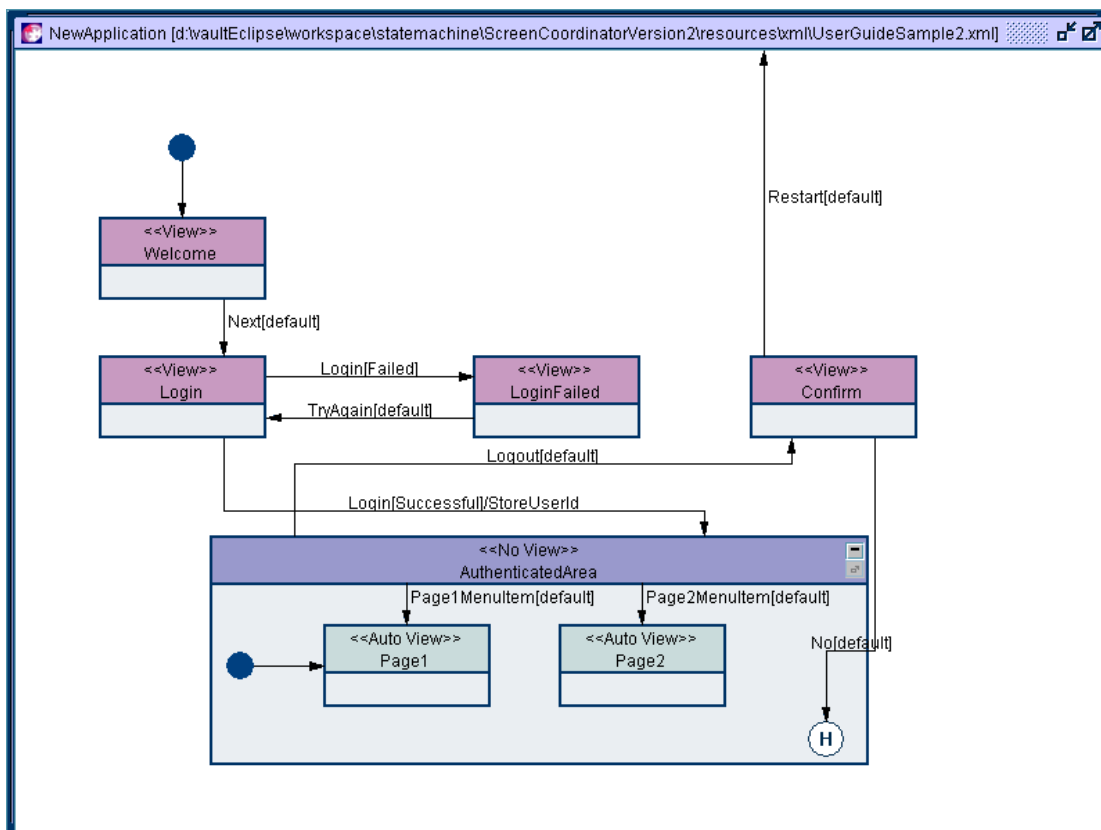
To connect a state to the master state, commence a transition as stated in the drawing state transitions section of this user guide. To end the transition, click the mouse anywhere in the desktop window's statechart area. Ensure that you click in an area where no parent or child state already exists. The following figure indicates where the end transition mouse click was to draw the Restart transition arrow on the Confirm state.

Click mouse on the desktop window's statechart to end the transition to the master state



Creating a transition from a state to the master state.

Once the mouse is clicked the “Transition Wizard” dialog is displayed. The transition’s details can be entered. The next figure shows the completed Restart transition for the confirm state.



The Restart transition from the Confirm state to the master state.

3.4.3 Drawing Transitions To And From Parent And Child States

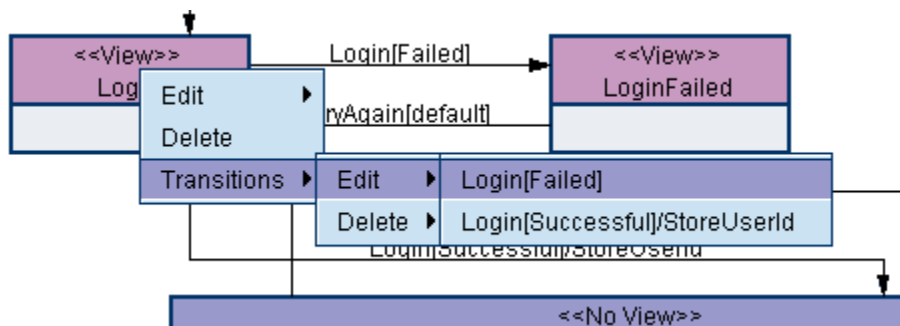
Transitions can be drawn between parents and child states just as a normal state transitions would be drawn. Dragging the transition component onto the parent state's header starts the transition and clicking the mouse on the child state's header ends the drawing of the transition as normal. The previous figure shows a couple of such transitions. The AuthenticatedArea has parent to child transition arrows to the Page1 and Page2 states.

3.4.4 Drawing Transitions To And From Non-Related Child States

Transitions can be drawn to and from states that are non-related child states. What this in effect means is that a transition arrow can be drawn between any two states anywhere on the statechart, regardless of whether they are children of the same parent state, not children or just child states of the statechart itself. The only limit on drawing a transition arrow is when one of the states is a pseudo-state. Again see the table in the statechart concepts of the user guide to confirm what transitions can be drawn with pseudo-states.

3.4.5 Editing Transition Details

Any transition can be edited by right-clicking on the start state of the transition arrow and launching the state's popup menu. Select Transitions -> Edit, followed by the transition's label menu item. The Login[Failed] transition menu item is shown in the next figure.

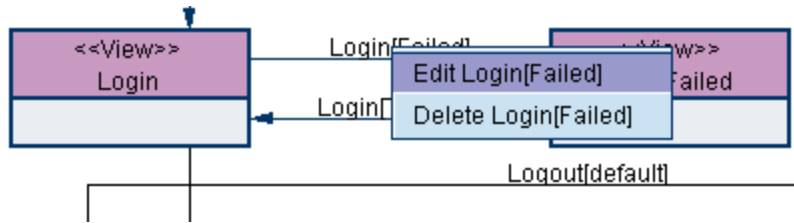


The Login[Failed] transition's edit menu item.

Clicking on the transition's menu item will bring up that transition's "Transition Wizard" dialog, where the transition's details can be changed. Clicking the OK button will close the dialog and update the transition with the edited changes.

3.4.5.1 Selecting Transitions Directly

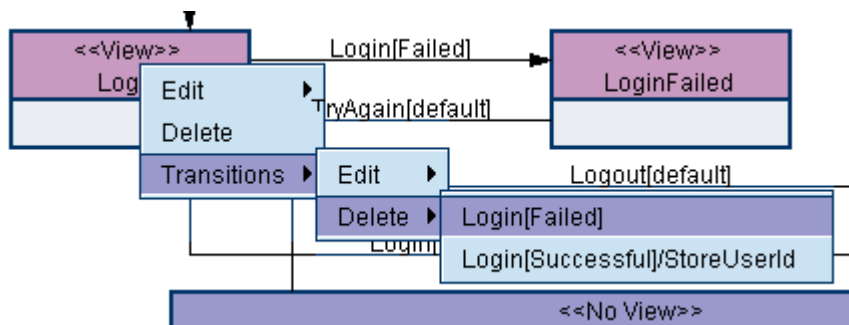
To select a transition so that it can be edited or deleted right-click anywhere on the transition that you wish to edit or delete. Right-clicking on the Login[Failed] transition will bring up that transition's popup edit menu as shown in the following figure.



Directly selecting transitions by right-clicking on them.

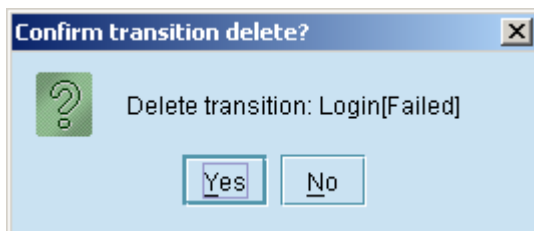
3.4.6 Deleting Transitions

To delete any transition, the user must right-click on the starting state of the transition to bring up the start state's popup menu. The user must then select Transitions -> Delete, followed by the transition's label menu item, to delete the transition. The following figure shows the delete transition menu item for the Login[Failed] transition.



The Login[Failed] transition's delete menu item.

Once the transition's delete menu item is selected, the user will be asked to confirm the deletion of the transition.



The confirm delete transition dialog.

When the user has confirmed the delete transition, the transition will be removed from the statechart. The transition can also be selected for deletion by right-clicking on the transition directly and displaying the transition's edit popup menu as described in the "Selecting Transitions Directly".

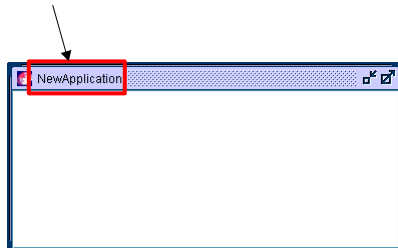
3.5 More Drawing Statechart Details

Additional to drawing states and transitions, the user can edit details of the statechart itself. The following sub-sections indicate features directly related to the statechart itself.

3.5.1 The Statechart Name

Each statechart represents an application. The name of the statechart is also the application name and the name of the master state of the statechart. The name of the statechart is displayed in the title bar for the desktop window of the statechart.

The statechart name

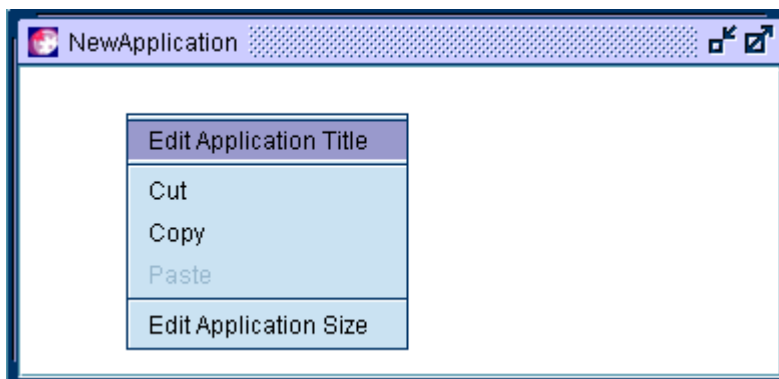


The statechart or application name highlighted.

The previous figure indicates where the statechart name is displayed.

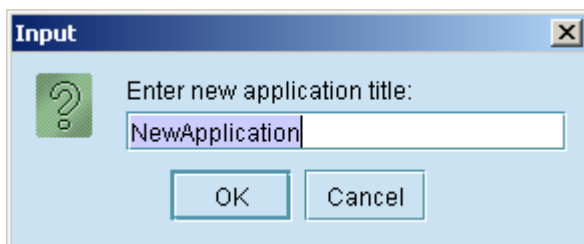
3.5.2 Renaming The Statechart

Right-clicking anywhere on the statechart where no state is currently located will display the statechart's popup menu. The following figure shows the statechart's popup menu.



The statechart's popup menu.

Selecting the "Edit Application Title" menu item will display the input dialog for editing the application's title.



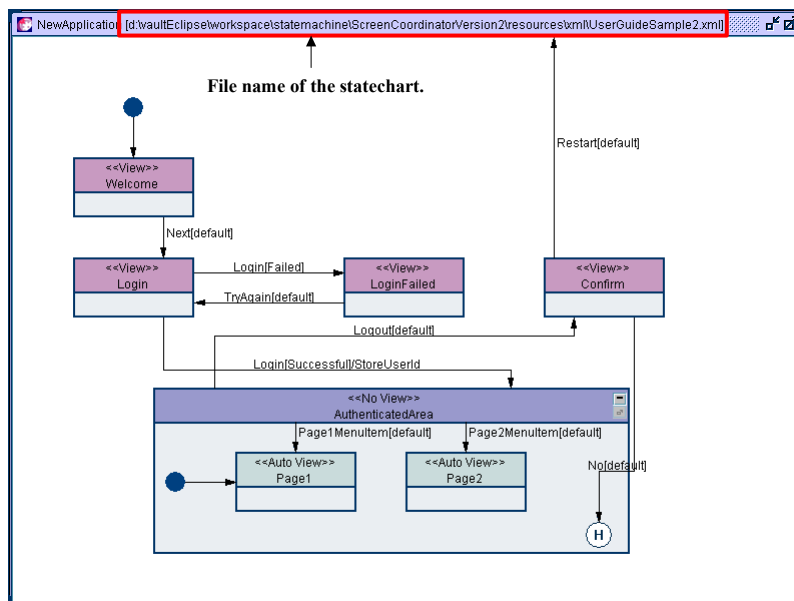
Input dialog for editing the application title of the statechart.

Once the OK button is pressed on the input dialog the title bar of the statechart window is changed to reflect the new application title.

3.5.3 Saving A Statechart

A statechart can be saved at any time by pressing the save button on the toolbar or by selecting File -> Save XML from the menu. If the statechart hasn't been saved before then the user will be asked to enter a filename for the statechart. By default the statechart's filename will be the same as the statechart's application title. The user can enter whatever filename they wish, however we suggest that you leave the filename to be same name as the statechart's application name.

Once the statechart has been saved to a file, the title bar of the statechart window will be updated to include the filename of the statechart. The following figure shows the highlighted title bar of an application saved to a file.



The highlighted title bar indicating the filename of the statechart.

The statechart can also be saved to another file at any time, by selecting File -> "Save XML As.." from the menu. A file chooser dialog will be displayed and the user will be allowed to enter a new name for the file name of the statechart.

As you add items to a statechart, such as states, pseudo-states, transitions, or move any state in the statechart, as well as editing any states or transition details, the tool will indicate that a file save is required by adding an asterisk to the statechart's window title bar.

3.5.4 Renaming a Saved Statechart

If you started a statechart and built it up over some period of time, you will have already named the statechart and saved it in a file. Over time you may then wish to rename the statechart to a more appropriate name. Renaming the statechart as shown in the “Renaming the statechart” section earlier in this document can do this. When you rename the statechart’s application title, the tool will also rename the file associated with the statechart to the same name automatically. The tool tries to maintain a match between the statechart name and the file name in which the statechart is saved. By changing the statechart’s application name you will also change the filename in which the statechart is saved.

3.5.5 Opening A Statechart

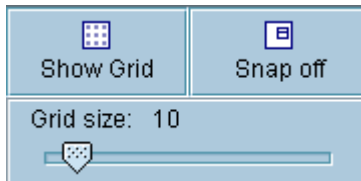
You can open a statechart at any time by pressing the open button on the toolbar, or by selecting File -> Open or, if the statechart was opened recently, by selecting the file from the recently opened file list on the File menu. If you already have a statechart opened you will be asked to save it (if it requires saving) before opening the selected statechart.

3.6 Miscellaneous Drawing Features

The following sub-sections detail the remaining important features required for basic drawing with the tool. More advanced drawing features will be covered in a later section entitled “Advanced statechart drawing”.

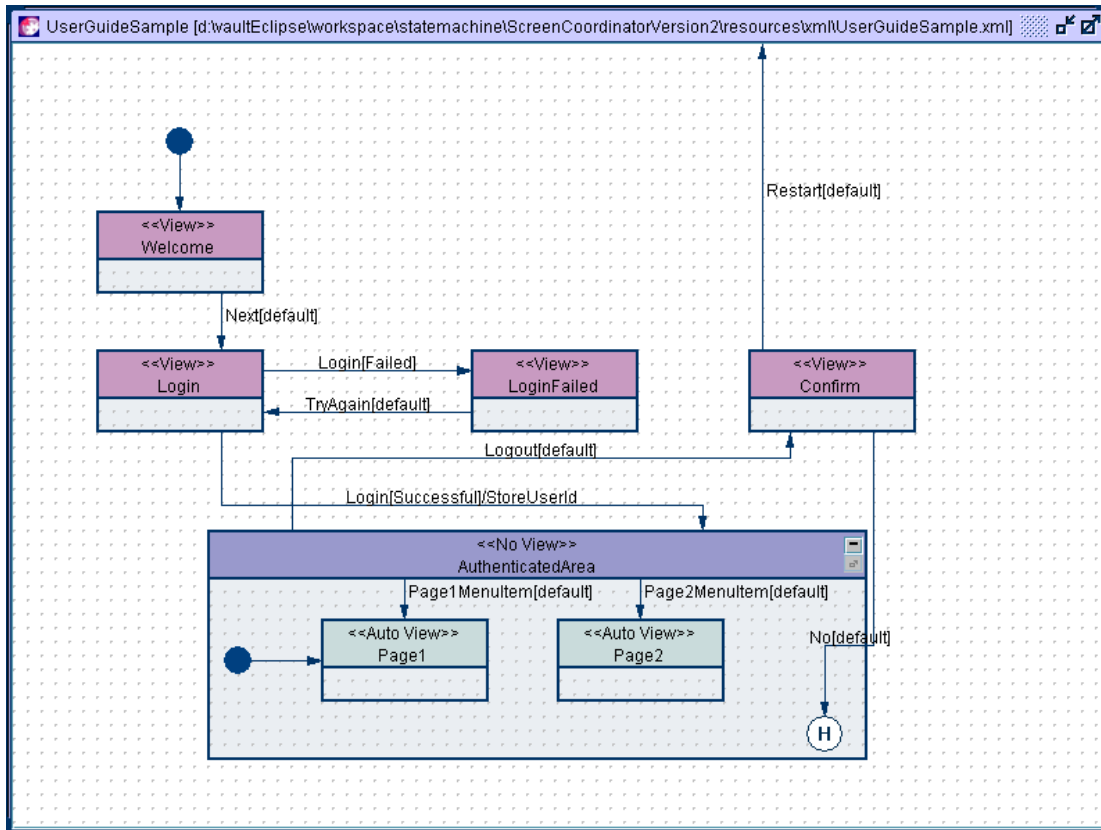
3.6.1 Using The Grid And Snap To Features

The tool provides a grid and snap to features for drawing your statechart application. The grid tool allows a visible grid to be used to position and align states drawn on the statechart. The snap to tool allows states to be located on actual grid points regardless of whether the grid is visible or not to the user. If a state is created or moved while the snap to tool is on, then the state will be automatically located to the nearest grid point. The following figure indicates the grid and snap features available on the orchestrator’s component palette.



The grid and snap to palette buttons.

By default the grid is off, while the snap to feature is on, when you start the Orchestrator tool. If you wish to switch the grid on, press the “Show Grid” button. Pressing the “Show Grid” button will switch the grid on and the statechart windows will show a grid as shown in the next figure.



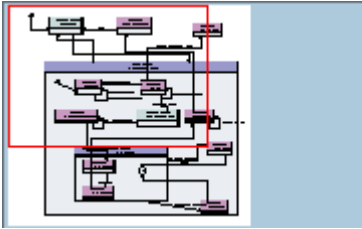
The grid switched on.

The “Show Grid” button will also be toggled to a “Hide Grid” button, so that the user can hide the grid when required. One small result of switching on the grid is that specifying transitions is slightly slower. This is a known bug that will be fixed in a later release. The grid size can be adjusted by moving the grid size slider left or right. Moving it left will decrease the grid size down to a minimum of 5 pixels per grid square. Moving the slider to the right will allow the grid size to be increased to a maximum of 50 pixels per grid square. The default grid size is 10 pixels per grid square.

Pressing the “Snap Off” button will switch the snap to feature off. When you create or move states they will be located exactly where they are dropped. Pressing the “Snap Off” button will toggle the button to a “Snap On” button, which will allow the snap to feature to be switched back on.

3.6.2 Using The Navigation Panel

As a statechart gets larger and moves outside the size of the desktop statechart window, that window will become scrollable. To help navigate around the window, a navigation panel has been incorporated into the tool. The following figure shows the navigation panel for a scrollable statechart.



The navigation panel for a scrollable statechart.

The navigation panel displays a miniature representation of the statechart. The red box in the panel indicates the currently visible area in the statechart's desktop window. If inside the red box is clicked the cursor will change to a hand and while the mouse button is held down the user can drag the box to a new location within the navigation panel. As the mouse is dragged the statechart window will be moved and located to correspond with the area visible in the navigation panel.

3.6.3 Printing Statecharts

Statecharts can be printed at any time by pressing the print buttons on the toolbar or by selecting File -> Print from the menu. Pressing the "Print All" button will print the currently selected desktop window statechart. The statechart will be printed to scale and if it is larger than a single page, all the pages will be printed.

Pressing the "Print On One Page" button will print the currently selected desktop window statechart. However, if the statechart is larger than a single page the statechart will be scaled to fit on a single page.

3.6.4 Export StateChart as a GIF

Statecharts can be exported as GIF files at any time by selecting 'File' -> 'Export as a GIF file' from the menu. The GIF file can then be imported into design documents or sent electronically which allows the statechart to be viewed and analysed without having a Screen Orchestrator installation.

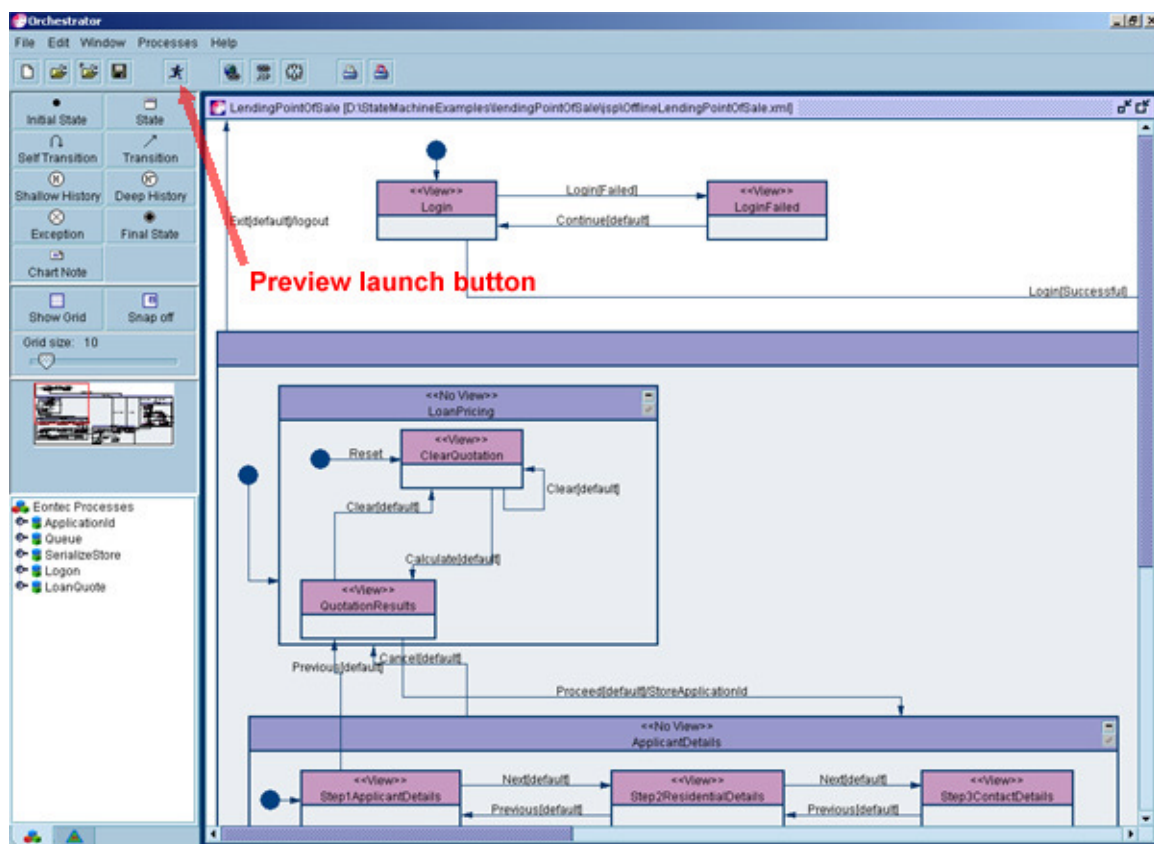
4 Preview And Web Deployment Capabilities

4.1 Introduction

The state machine framework can run the application that is produced by the orchestrator tool. The application is typically web based and must therefore be packaged into a web archive (WAR) file and deployed on a suitable Web server. The orchestrator tool provides an in-built swing based preview window for testing and verifying the statechart design. It also provides an in-built mechanism for creating and deploying a war file for web based applications. The following sections show how these mechanisms can be used.

4.2 Preview Capability

As the statechart is being developed, the designer or developer can preview the statechart at any time by invoking the preview feature of the orchestrator tool. The following figure indicates where the button to launch the preview feature is located.



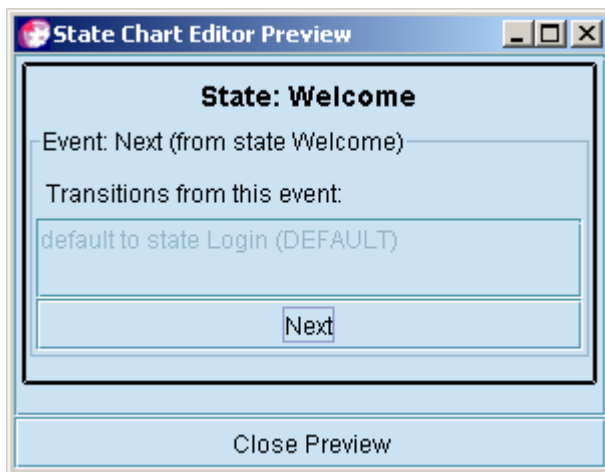
Highlighted preview launch button.

The preview window will display the initial state and allow the user to fire events to the next state in the drawn statechart. The preview window does not use the specified view state type but instead uses a swing-based autoview to display the state. If the state's events have the correct controllers and guard condition

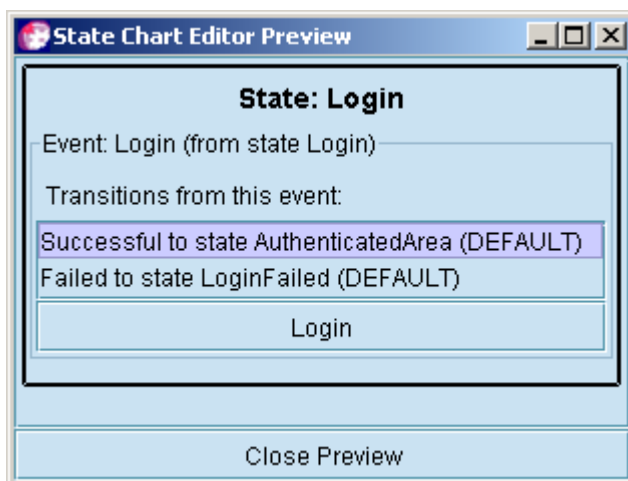
classes then the preview window can be used to follow the actual transitions that would occur in the real application.

The screen orchestrator tool also allows you to use a special controller with the preview window. If the user when drawing the statechart uses the AutoViewController for events then the preview window will allow the user to actually select the transition they want to follow. This is extremely useful for verifying all the statecharts transitions including those that might represent rare behaviour that is difficult to duplicate. The following table with figures show the earlier login example with AutoViewController used for events with more than one transition and the way in which the transitions can be followed using the preview feature.

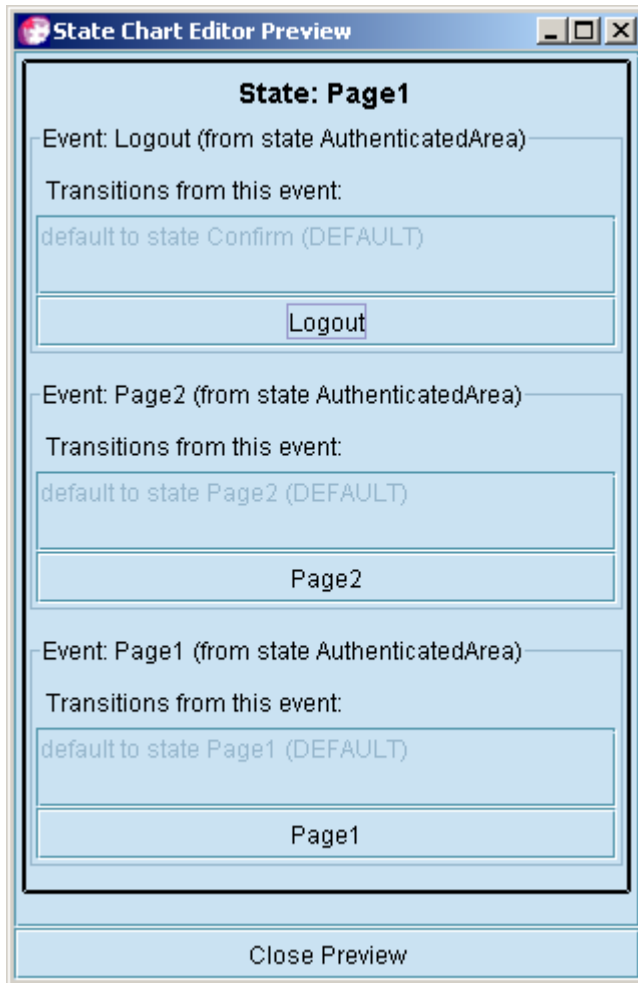
Initial state of the example application is the Welcome state. The Next event will be fired by pressed the Next button.

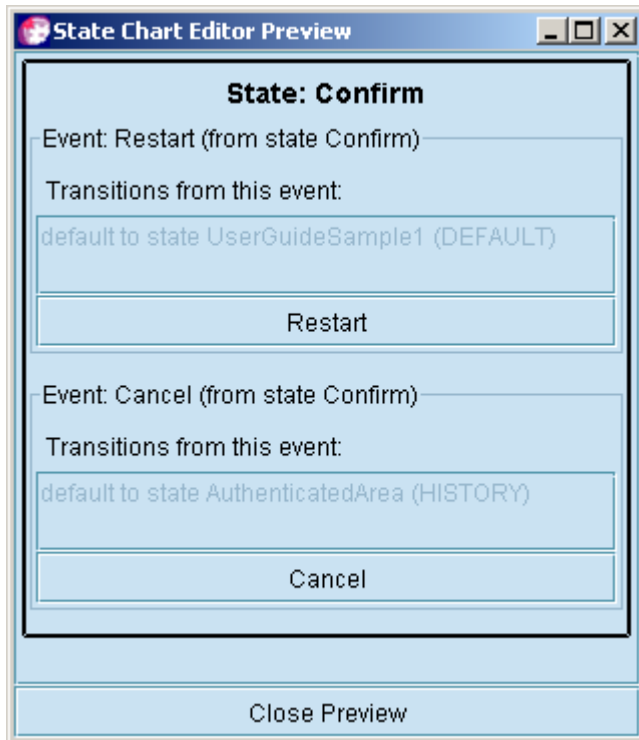


The Next event takes you to the Login state. This state's Login event has 2 possible transitions. The event is either successful or it fails. The AutoView Controller is used for this event which allows the user to select the Successful transition to be followed when the Login button is pressed.



The Login was successful and so the initial state of the AuthenticatedArea is the Page1 state. We will now press the Logout button to fire the Logout event.





4.3 Web Deployment Capability

As stated in the introduction of this section the orchestrator tool can be used to produce a WAR file for deployment of web-based applications. The following sub-sections indicate how this feature can be accessed and used.

4.3.1 Install an appropriate web server

The state machine framework can be deployed on any suitable http server that supports Java servlets and JSPs. The state machine and the war file produced by the orchestrator tool has being tested on the following application servers:

- JBoss 3.0.x and higher
- JBoss 3.0.x with Tomcat and higher
- WebLogic 6.1 and higher
- WebSphere 5.x and higher (may need to update the jdom.jar installed with this server for the state machine to work correctly).

4.3.2 WAR properties saved per statechart

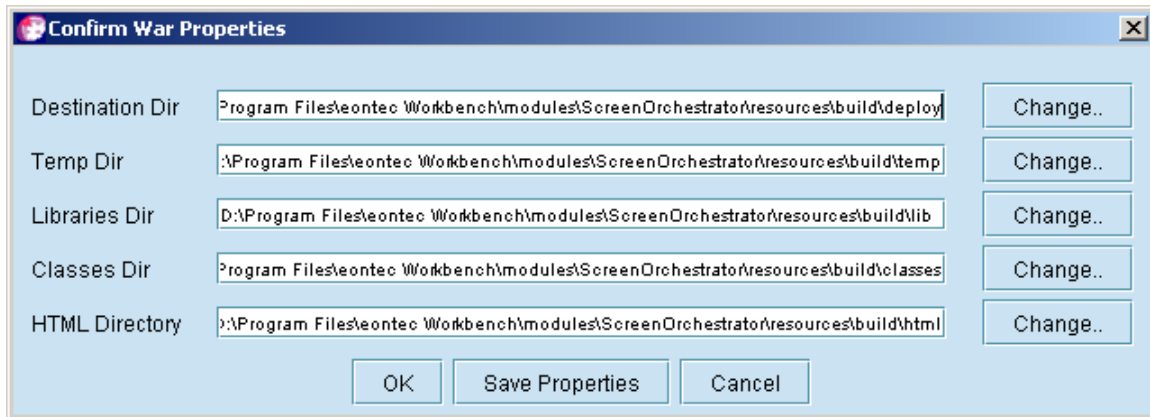
War properties were previously only set for the tool rather than for individual statecharts. This has been changed so that war properties can now be set per statechart. The tool maintains a set of default values for the war properties. When a new chart is created the default war properties are automatically assigned.

These properties can then be changed for that specific statechart. When the statechart is saved these war properties will be saved with the statechart. When the statechart is re-opened the charts war properties are

then set according to the values obtained from the statechart. This allows each statechart to have different war properties.

4.3.3 Configuring the WAR file within the tool

In the orchestrator tool go to *File>>>War Properties*. You will then be presented with the following dialog:



This dialog will be populated with default values for the war file. You can change them if you wish.

4.3.3.1 Destination Dir

This value will probably be changed most often. It is where the application will put the packaged war file. You should point this to the “webapps” folder of the web server e.g. if you were using JBoss then you might point it to a “.\\jboss-3.0.x\\server\\default\\deploy” directory. Once the war file has been generated it will be transferred to this location for deployment.

4.3.3.2 Temp Dir

This is the folder where the WAR will be generated and packaged before it is copied to the \$Destination Dir value.

4.3.3.3 Classes

This is where the application’s additional classes should be located to be included in the WAR. If the application uses any additional controllers or guard condition classes in the application then they should be placed here so that they will be included in the application’s WAR file.

4.3.3.4 HTML Directory

Any html or JSP files that the WAR needs should be located in this directory. Any additional directories and files such as images and style sheets should also be located under this directory.

4.3.4 Deploying the war file

When you are happy with the properties for the war file you can deploy the webapp by choosing *File >>> Save XML and regenerate war*.

This will create the war file and place it the directory specified in the \$Destination Dir variable in the war properties. Some application servers will have the ability to hot re-deploy the war, for those that do not then the application server will have to be re-started.

Generally, the URL to run the application will be of the following format.

http://<<serverName>>:<<portNumber>>/<<ApplicationName>>/StateMachine

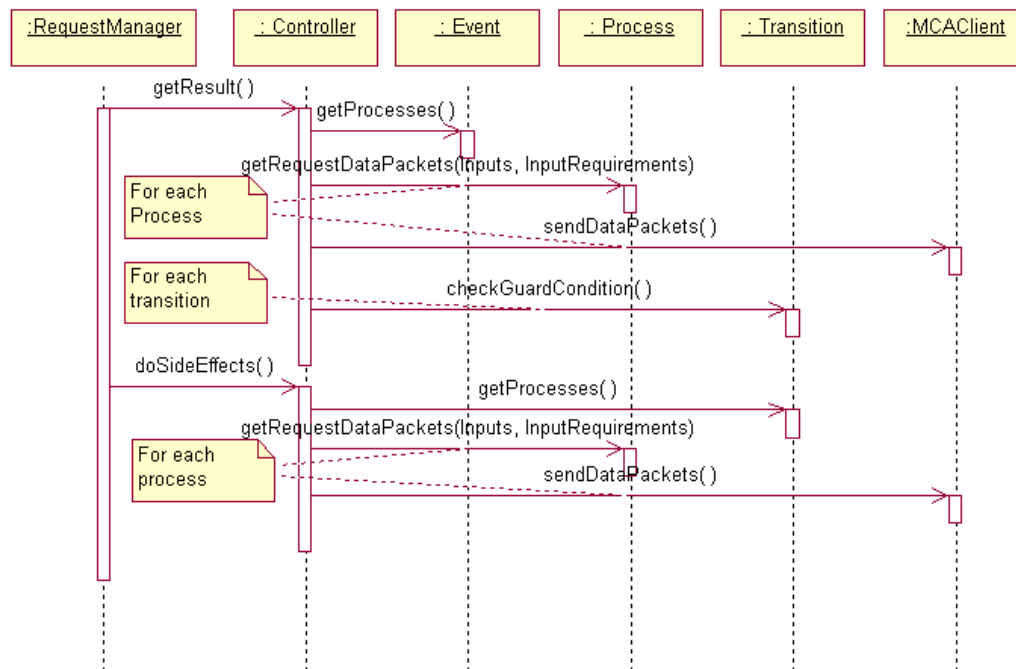
e.g. if a [EbankingExample.xml](#) file was open and deployed on an application server installed locally and on a port number of 8080, then the URL would be: <http://localhost:8080/EBankingExample/StateMachine>.

5 Designing Events With Processes And Guard Conditions

This document describes how to define complex events in the editor including processes, guard conditions and side effects. It goes through the sequence followed by the state machine when handling an event, and how to add processes and useful guard conditions to that sequence to control a real user interface.

5.1 Handling an Event

When the state machine receives an event, it follows the sequence in this diagram.



When the event is received the state machine calls the Controller's `getResult` method. The controller calls all the processes associated with the event and then calls the `checkGuardCondition` method for each of the transitions. One of the guard conditions is going to evaluate to `TRUE`. The `getResult` method will return that transition.

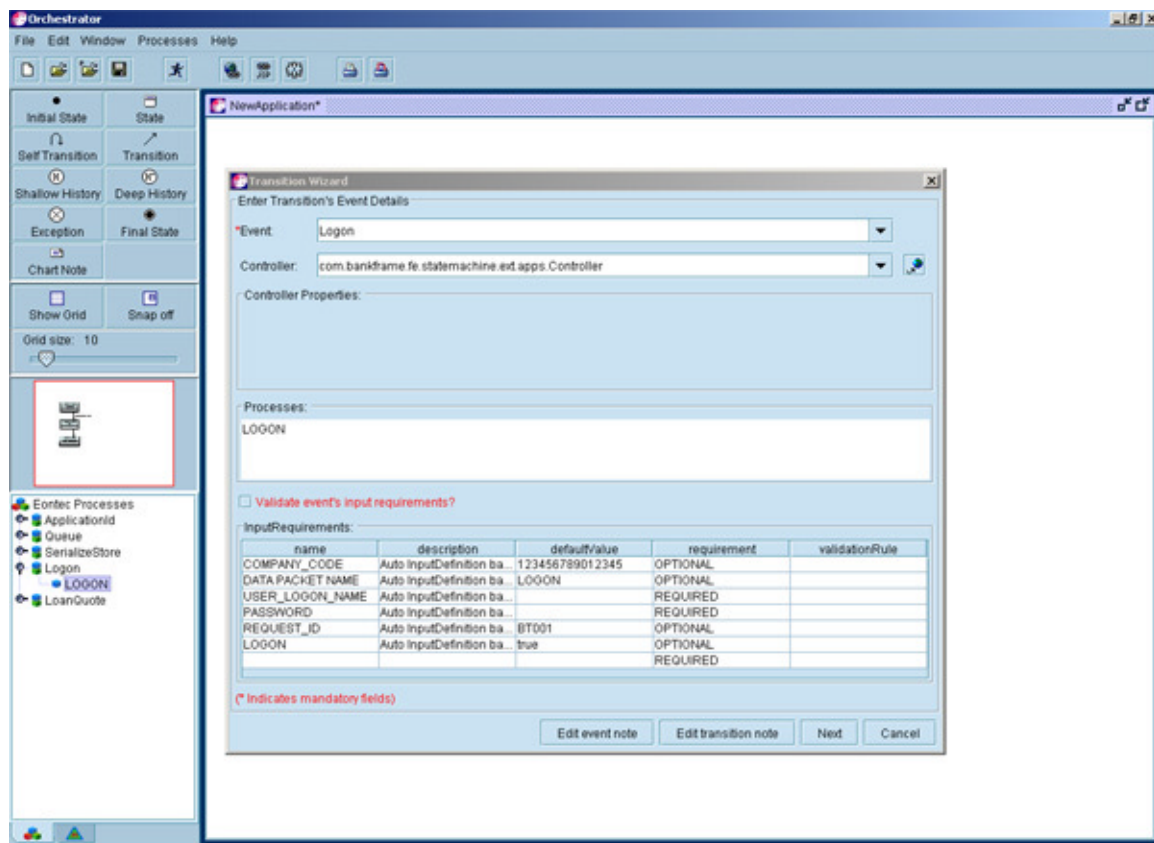
The state machine will then call the `doSideEffects` method. The controller will call all the processes associated with the transition.

(Note: This is the sequence followed by the `com.bankframe.fe.statemachine.ext.apps.Controller` class. Other controller classes may behave differently)

5.2 Associating Processes with Events and Transitions

At its simplest, associating processes with events and transitions is just a matter of dragging the process from the process tree to the transition wizard dialog.

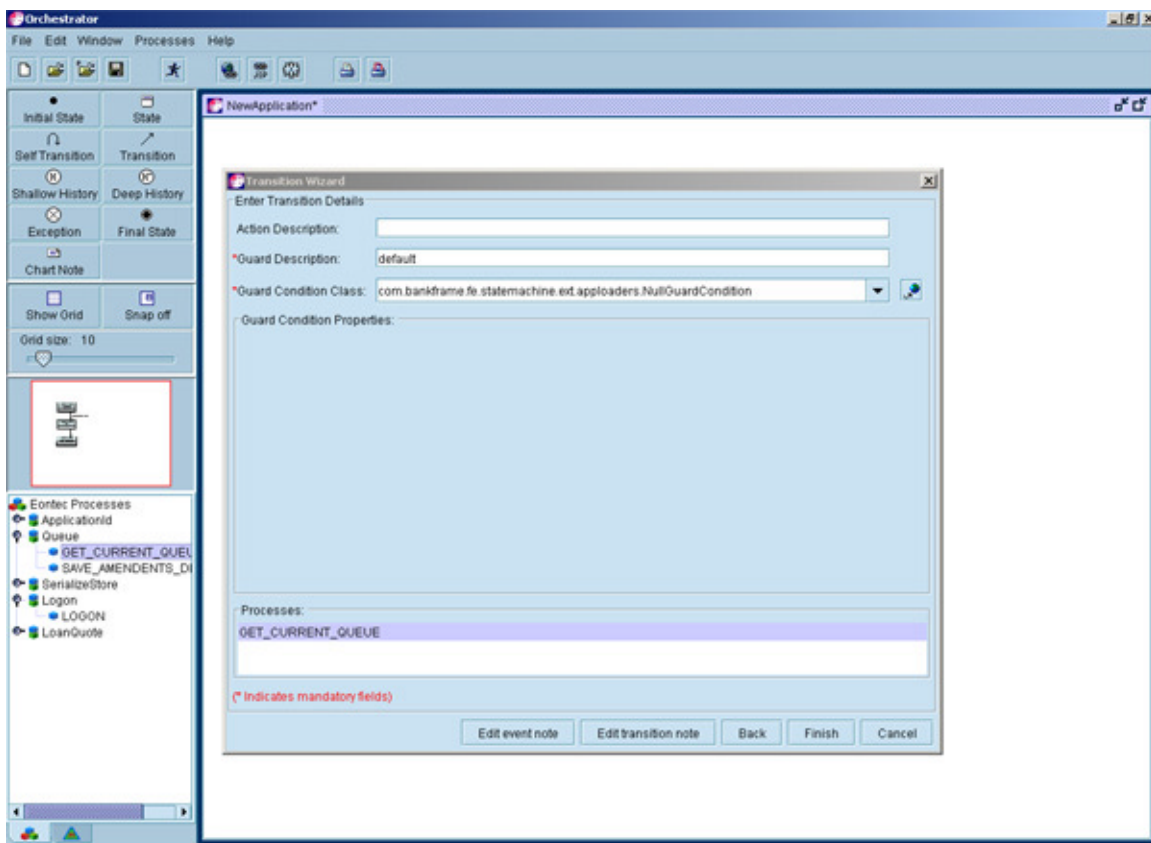
Designing Events With Processes And Guard Conditions ■ Associating Processes with Events and Transitions



See the “Process Integration” section for details on loading and editing processes on the process tree.

The transition wizard dialog has two pages. To associate a process with an event, drag the process onto the first page of the wizard, as shown above. To associate a process with a transition drag the process onto the second page of the dialog, as below.

Processes on the event (the first page) will be called when the event is received from the user before any guard conditions are tested, as described in the ‘Handling an Event’ section. Processes on the transition will be called after the transition’s guard condition is tested and only if the guard condition evaluates to ‘true’.



5.3 Setting the Input Requirements

When you add a process to an event or transition, the parameters to that process become input requirements to the event. (The transition does not have independent input requirements, so all parameters to all processes on an event and all its transitions are considered input requirements to the event.) For each input requirement there are four attributes to be set: the parameter name, the (optional) parameter description, the (optional) default value, and the requirement type. The parameter name and requirement type must be provided. In most cases, the parameter name will be a DataPacket key, as used by a process the event references.

The requirement type must take one of the following five values:

- **REQUIRED:** indicates that the parameter value must be supplied by the user
- **OPTIONAL:** indicates that the parameter value may be supplied by the user
- **CONSTANT:** indicates that the parameter value will always be the supplied default value
- **CODED:** indicates that the parameter value will be supplied by customised code (this option will be used very rarely)
- **PROCESS:** indicates that the parameter value will be supplied by the process definition (this option will be used for parameters such as the DATA PACKET NAME and REQUEST_ID)

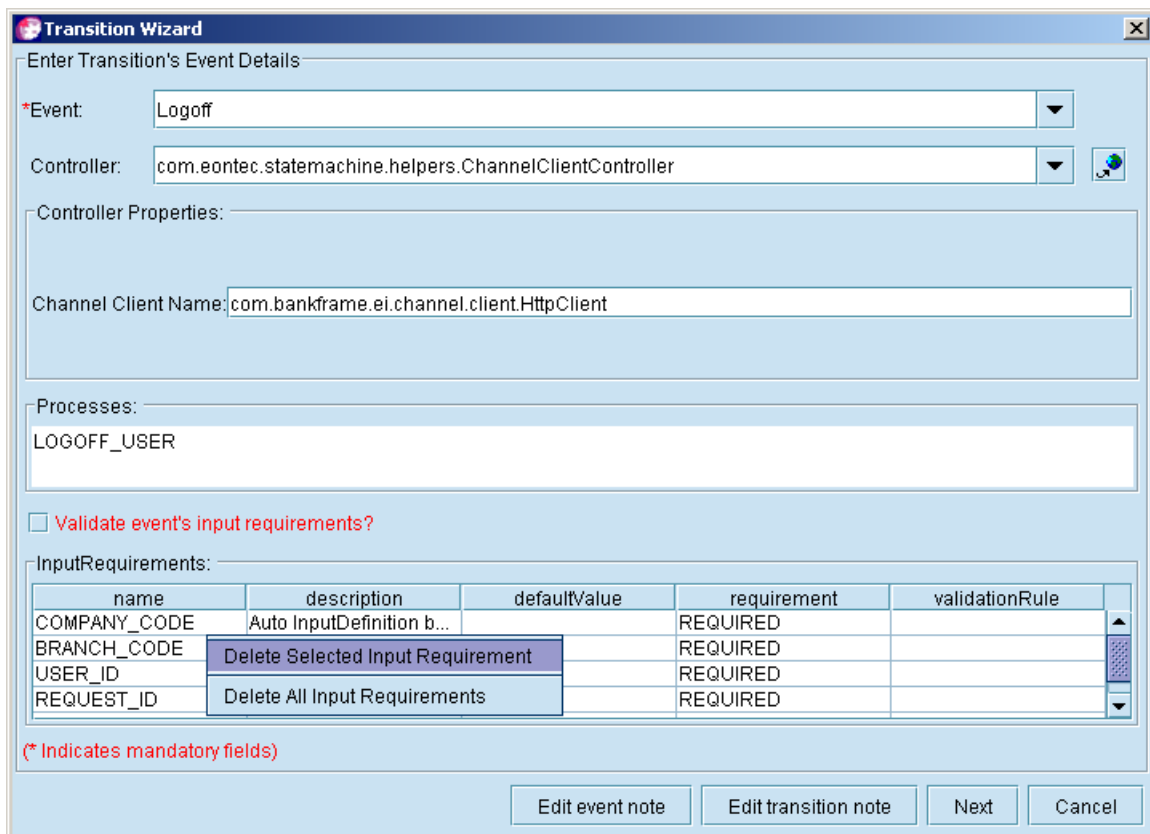
The input requirements are set in the table on the first page of the transition dialog.

Note that you may add input requirements that are not required by any process. For example: if you are

using an InputBasedGuardCondition (described below) you can add the parameter being tested by that guard condition to the input requirements.

5.4 Deleting Input Requirements

When events or states no longer use certain input requirements, the tool provides the means to remove these parameters by deleting them from the input requirements table by right hand clicking on the selected parameter in that table.



The screenshot shows the 'Transition Wizard' dialog box. The 'Event' is 'Logoff' and the 'Controller' is 'com.eontec.statemachine.helpers.ChannelClientController'. The 'Channel Client Name' is 'com.bankframe.ei.channel.client.HttpClient'. The 'Processes' list contains 'LOGOFF_USER'. The 'Validate event's input requirements?' checkbox is unchecked. The 'InputRequirements' table is as follows:

name	description	defaultValue	requirement	validationRule
COMPANY_CODE	Auto InputDefinition b...		REQUIRED	
BRANCH_CODE			REQUIRED	
USER_ID			REQUIRED	
REQUEST_ID			REQUIRED	

A context menu is open over the 'BRANCH_CODE' row with two options: 'Delete Selected Input Requirement' and 'Delete All Input Requirements'. A red note at the bottom states '(*) Indicates mandatory fields'. At the bottom of the dialog are buttons for 'Edit event note', 'Edit transition note', 'Next', and 'Cancel'.

Either the selected input requirement is deleted or all the input requirements are deleted from the InputRequirements table.

5.5 How the request DataPackets are built

To be sure the correct data is being sent to the processes, it is important to understand how the Controller and Process objects build up the DataPackets that are sent through the MCA Services client.

The main deciding factor is the requirement type specified in the input requirements for each parameter, according to these rules:

- **REQUIRED:** The value will be taken from the request received from the user. If the value is not in the request, a null value will be used.
- **OPTIONAL:** The value will be taken from the request received from the user if possible. If the request doesn't contain a parameter of the correct name, the value will be the default value for this input requirement.

- **CONSTANT:** The value will always be the default value for this input requirement.
- **CODED:** The value must be provided by custom-written Java code in the controller.
- **PROCESS:** The value will be taken from the DataPacket definition in the Process.

The DataPackets will be built to contain all the keys specified in the DataPacket definitions in the Process.

5.6 Defining Guard Conditions

With the processes defined and receiving the correct data from the user interface, the next detail is to define the guard conditions on the event's transitions.

The basic rule you have to remember is: "For every event, no matter what inputs are provided from the user or received from the processes, exactly one of the guard conditions on the event's transitions must be true. All other guard conditions must be false."

There is one exception to this rule: if an event has only one transition, that transition will always be followed no matter what the guard condition.

There are four types of guard condition available by default in the editor:

5.6.1 NullGuardCondition

This is a guard condition that always returns an undefined value, neither true nor false. You should use this guard condition only if the event has only one transition, or you are using a controller that does not test guard conditions (such as the AutoViewController or SimpleController).

5.6.2 FixedValueGuardCondition

This is a guard condition that will always return either true or false as set in the transition dialog. You can use this guard condition during testing of an application to force it along a particular route to an area you need to test.

Transition Wizard

Enter Transition Details

Action Description:

* Guard Description:

* Guard Condition Class:

Guard Condition Properties:

Guard Condition Value:

true

false

Processes:

(* Indicates mandatory fields)

Edit event note Edit transition note Back Finish Cancel

5.6.3 InputBasedGuardCondition

The InputBasedGuardCondition tests some value received from the user or in the user session.

The options you can set are:

- Whether or not the input must contain or not contain the specified value
- Whether or not the test should be case sensitive
- Where or not the value is for testing
- The name of the input
- The value to test for

Transition Wizard

Enter Transition Details

Action Description:

* Guard Description:

* Guard Condition Class:

Guard Condition Properties:

Input must:

Case Sensitive:

Input Source:

Parameter Name:

Parameter Value:

Processes:

(* Indicates mandatory fields)

5.6.4 ResultBasedGuardCondition

The ResultBasedGuardCondition tests the result from a process associated with the event.

The options you must set on this are:

- Whether or not the result should contain or not contain the value
- Whether or not the test should be case sensitive
- The name of the process that was called
- The name of the DataPacket in the result DataPackets that should be tested
- The key/value pair within that DataPacket to be tested.

Note that the DataPacket name and key will always be considered case sensitive, only the value would be tested for case sensitivity.

Transition Wizard

Enter Transition Details

Action Description:

* Guard Description:

* Guard Condition Class:

Guard Condition Properties:

Result must:

Case Sensitive:

Process Name:

Response DataPacket Name:

DataPacket Key:

Value:

Processes:

(* Indicates mandatory fields)

5.6.5 TimeoutGuardCondition

The TimeoutGuardCondition tests whether a timeout has resulted from a process associated with the event.

The variables to be set are:

- The timeout parameter to test the process against, e.g. `TIMEOUT_STARTED` would start counting the timeout from the time the process started.
- The timeout threshold is the number of milliseconds that would constitute a timeout.

Transition Wizard

Enter Transition Details

Action Description:

* Guard Description:

* Guard Condition Class:

Guard Condition Properties:

Timeout Parameter:

Timeout Threshold (millisecs):
 (For example 150000 millisecs is 2.5 minutes)

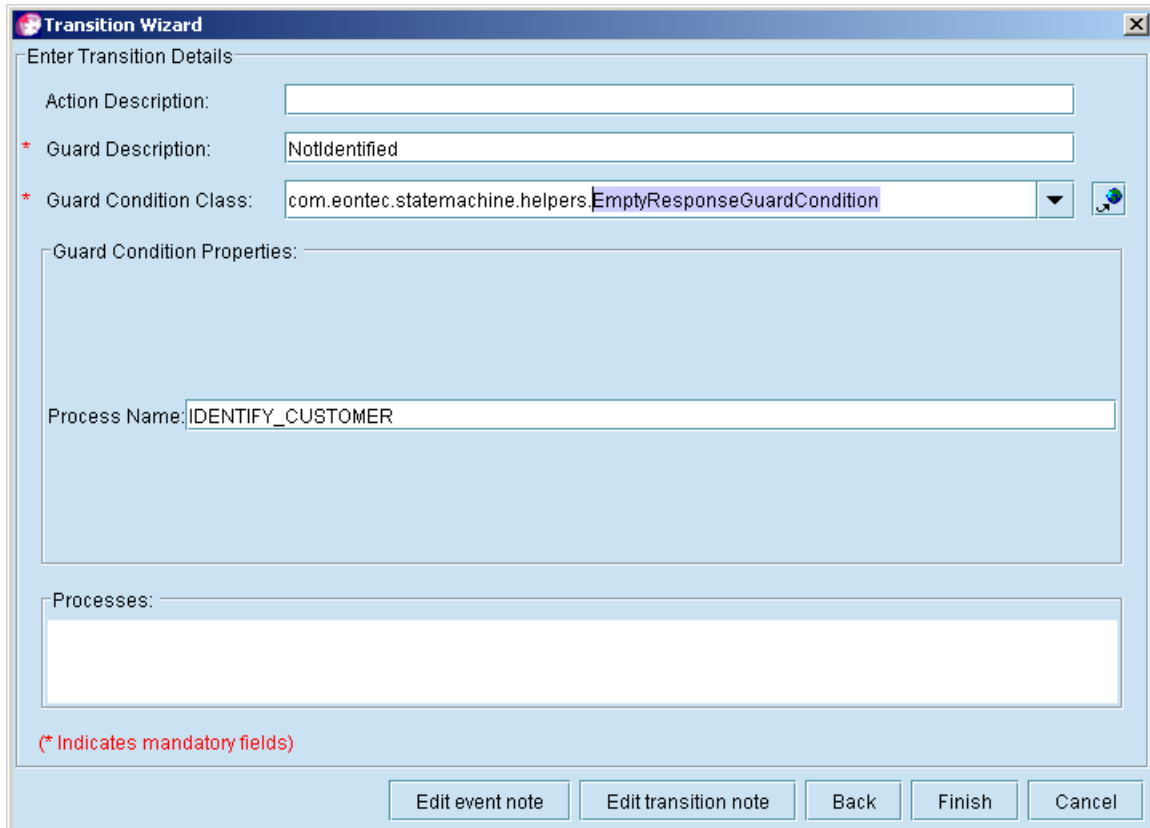
Processes:

(* Indicates mandatory fields)

5.6.6 EmptyResponseGuardCondition

The EmptyResponseGuardCondition tests whether the response from a process associated with the event is an empty response. The variable to be set is:

- The name of the process that was called



The image shows a 'Transition Wizard' dialog box with the following fields and controls:

- Enter Transition Details:**
 - Action Description:
 - * Guard Description:
 - * Guard Condition Class: (with a dropdown arrow and a globe icon)
- Guard Condition Properties:**
- Process Name:**
- Processes:**

At the bottom left, there is a red note: (* Indicates mandatory fields). At the bottom right, there are five buttons: 'Edit event note', 'Edit transition note', 'Back', 'Finish', and 'Cancel'.

5.7 Other Controller classes

All of the above assumes you are using the `com.bankframe.fe.statemachine.ext.apps.Controller` class as the event controller.

There are other controller classes that you can use in different circumstances.

5.7.1 The SimpleController

The SimpleController class, `com.bankframe.fe.statemachine.base.apps.SimpleController`, can be used to handle all trivial events. All events with just one transition and no associated processes can be handled by the SimpleController. The SimpleController will handle these trivial events faster than the main Controller class.

5.7.2 The AutoViewController

The AutoViewController class, `com.bankframe.fe.statemachine.base.apps.AutoViewController`, can be used in conjunction with the AutoView, XSLTAutoView, or preview features to allow you choose which transition to follow based on selecting from a list of available transitions.

5.7.3 Additional Controllers

The following controllers are available in the `com.eontec.statemachine.helpers` package:

- **ChannelClientController:** This controller provides a mechanism for specifying what channel client is used when executing processes. By default is it set to use HttpClient and in this mode it behaves in the exactly same way as the standard main controller.
- **DataCollectorController:** This controller is a subclass of the ChannelClientController, but it adds very special behaviour in handled DataPackets. This controller can build multiple DataPackets from the input request values and can append these DataPackets to any request that will be executed by any process specified by the event or transition.
- **MultipleRequestController:** A new controller, the MultipleRequestController has been developed to handle more complex multiple DataPacket requests. The state machine has also been modified to handle multiple DataPacket requests and these modifications in association with the MultipleRequestController are designed to better enable the state machine to handle these complex requests.
- **ClearUserSessionController:** This controller is a subclass of the ChannelClientController and can be used to clear values in the Inputs object
- **AddToUserSessionController:** This controller is a subclass of the ChannelClientController and can be used to add values to the Inputs user session

Please read the JavaDocs for these classes for further information on how they should be used.

5.7.4 Custom Controllers

There may be some times when the standard processes, controllers and guard conditions are not enough to meet the requirements of the user interface. In that case it is possible to write controller classes with custom code to meet the requirements. See the document called 'Writing Controller classes' for more information on how this is done.

If a custom Controller class has been written for an event, enter the class name (including package name) in the 'Controller' box on the transition dialog.

5.8 Add Common Fields to Every Request

The state machine has now the ability to add common items to every request sent to a Siebel Retail Finance server. For example, an application that uses Entitlements will require the following values to be contained within every request:

`ENTITLEMENTS_CHANNEL_ID`

`ENTITLEMENTS_ACTOR_ID`

`ENTITLEMENTS_ACCESS_PROVIDER_ID`

`ENTITLEMENTS_ACCOUNT_NUMBER`

`ENTITLEMENTS_BRANCH_CODE`

The state machine can be configured to add these items to every request by setting the following keys and values in the BankframeResource.properties file:

```
'# Common Request Items

#####

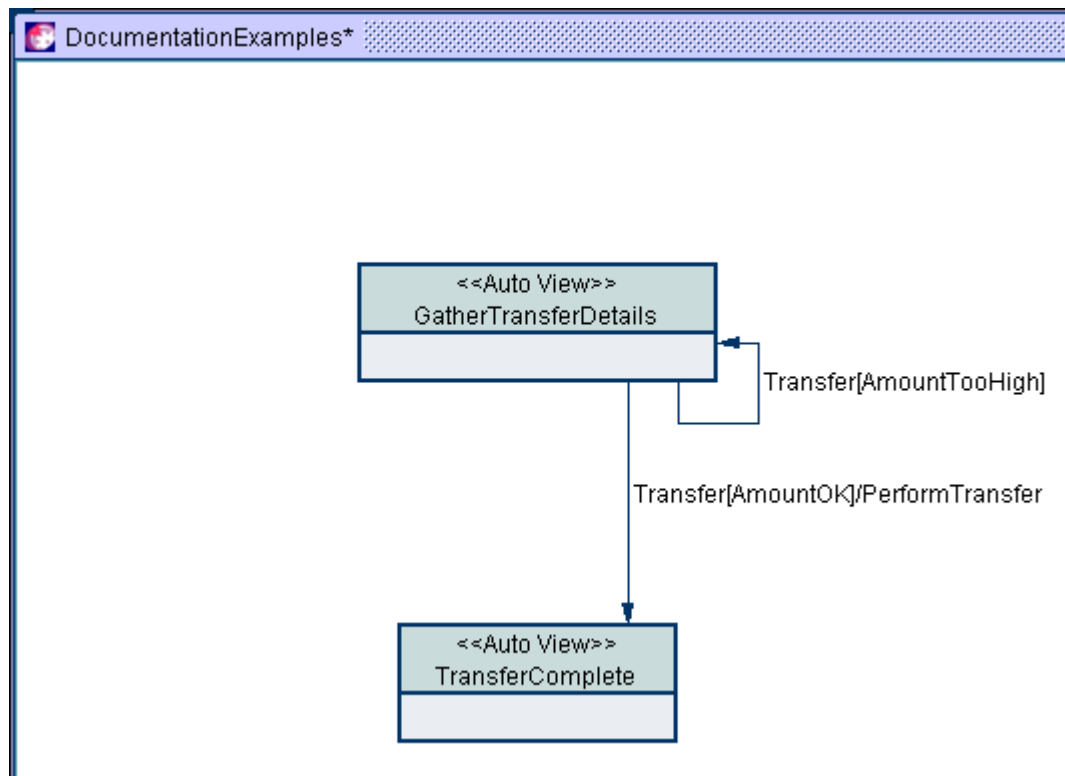
common.request.items.enable=true

common.request.items.fields=ENTITLEMENTS_CHANNEL_ID;
ENTITLEMENTS_ACTOR_ID;ENTITLEMENTS_ACCESS_PROVIDER_ID;
ENTITLEMENTS_ACCOUNT_NUMBER;ENTITLEMENTS_BRANCH_CODE'
```

These items are not required to be in the process definition but must have their key names listed in the "common.request.items.fields" value in the BankframeResource.properties file and the "common.request.items.enable" value must be set to true. The values for these fields must be available in the inputs object used by the state machine to hold user data. If the value for these fields is not in the inputs object already then they are put into the request as blank values.

5.9 Worked Example

This worked example describes how you might code an event with a process call, a result-based guard condition, and a side effect on one of the transitions. This would appear as follows:



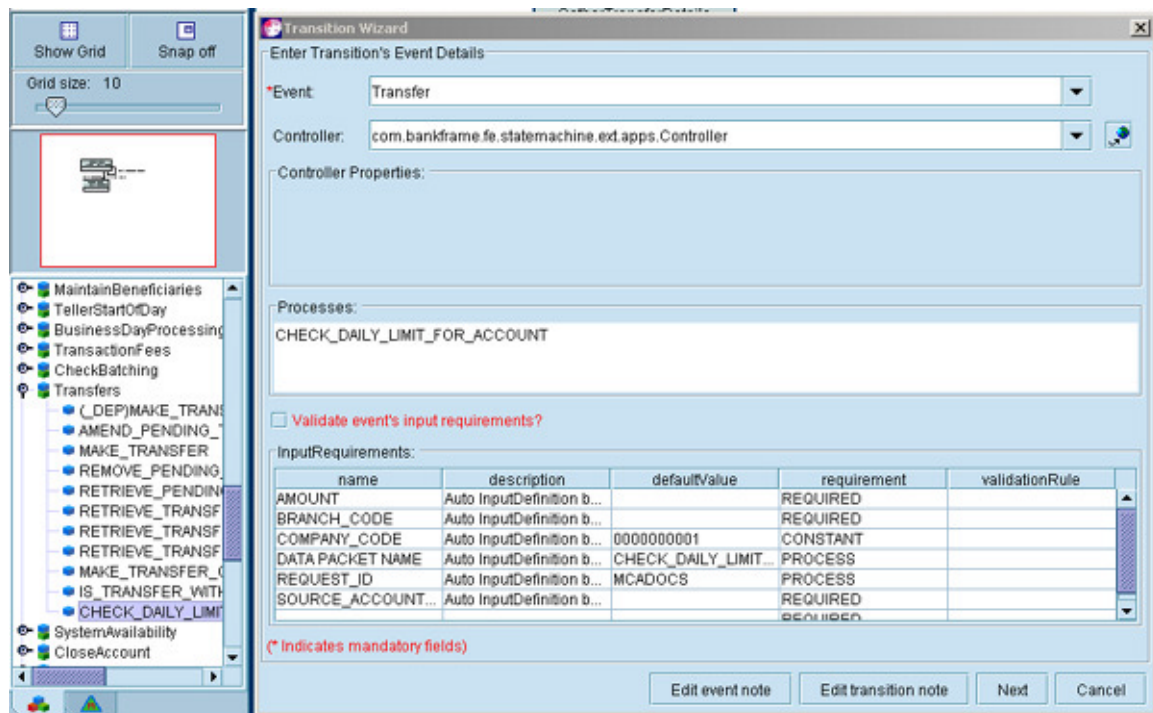
The user interface is intended to allow a user request a funds transfer between two accounts they own. Users are restricted in the amount they can transfer in any given day, so the system must check whether the amount chosen is above the limit. If it is not above the limit the transfer must be performed and the user given a transaction record number.

To support this, there are two tier-1 methods defined. The first determines whether the amount specified is within the allowed range, the second performs the transfer and returns the record number.

Starting with just the two transitions, the first step is to add one of the transitions. I'll start with the [AmountTooHigh] transition.

Drag from the self transition button on the palette bar onto the GatherTransferDetails state, and set the event name ('Transfer') and controller name ('com.bankframe.fe.statemachine.ext.apps.Controller').

Drag the 'CHECK_DAILY_LIMIT_FOR_ACCOUNT' process onto the event.



You will see the InputRequirements box filled with values taken from the process definition. You need to set the requirement type for each of these inputs correctly.

The amount, branch code and source account number will all be supplied by the user, and so are 'REQUIRED'. The company code will not change from the value supplied, and so can be set 'CONSTANT'. The data packet name and request ID are both particular to the process, so should be set 'PROCESS'.

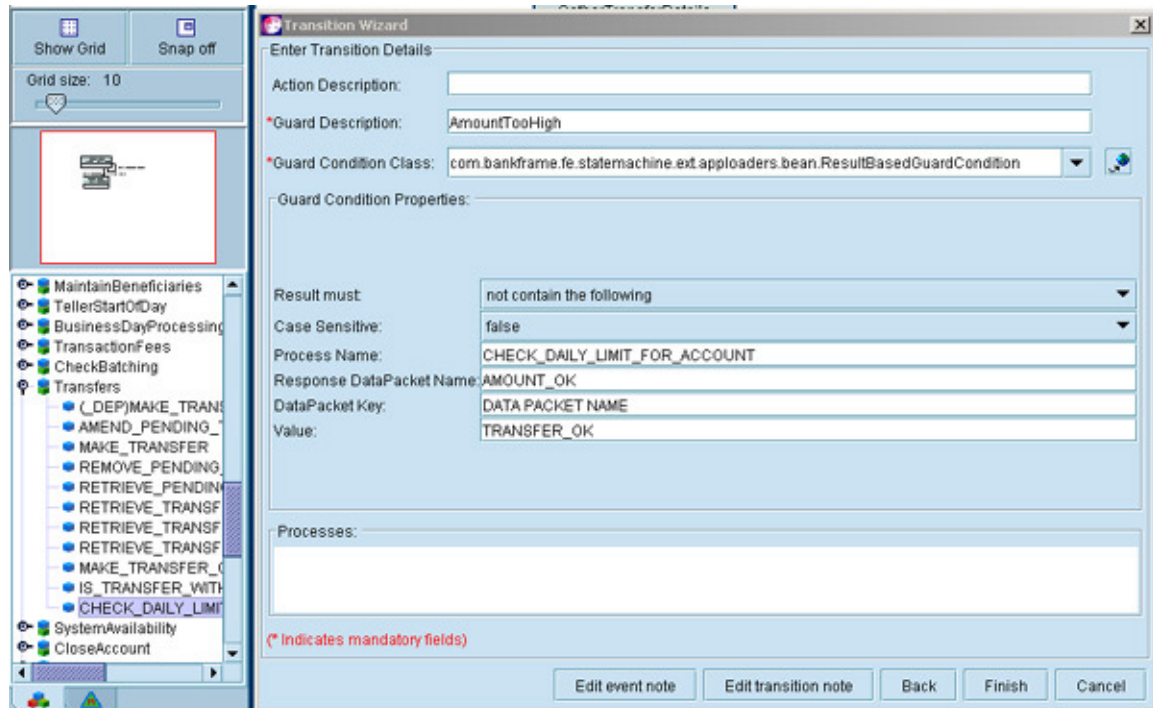
Now set the guard condition for this transition. We want this transition to be followed if the process does not return a DataPacket called 'AMOUNT_OK'. A ResultBasedGuardCondition can make that test.

Select the ResultBasedGuardCondition and enter the following values for each corresponding guard condition property – as per the screen shot below:

- Result must: not contain the following
- Case Sensitive: false
- Process name: CHECK_DAILY_LIMIT_FOR_ACCOUNT
- Response DataPacket name: AMOUNT_OK
- DataPacket key: DATA PACKET NAME
- Value: TRANSFER_OK

(Since we are testing the data packet name, the key we need to test is 'DATA PACKET NAME' and the value is the data packet name to test for.)

You should also set the guard description to something meaningful, such as AmountTooHigh.



[AmountTooHigh] transition.

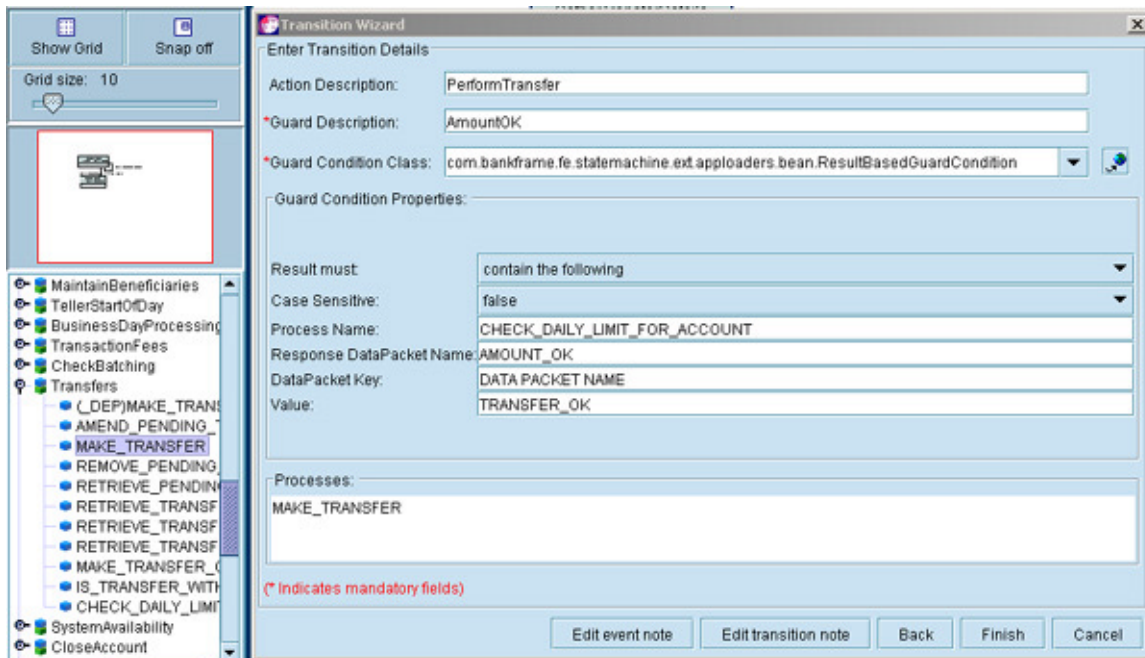
Finish that transition and start the [AmountOK] transition from the GatherTransferDetails state to the TransferComplete state. Drag from the transition button on the palette bar to the GatherTransferDetails state, then click on the TransferComplete state.

Select the 'Transfer' event from the event drop-down list, and you should see the controller, process and input requirements data all filled as in the previous transition. Move on to the second page to set the guard condition.

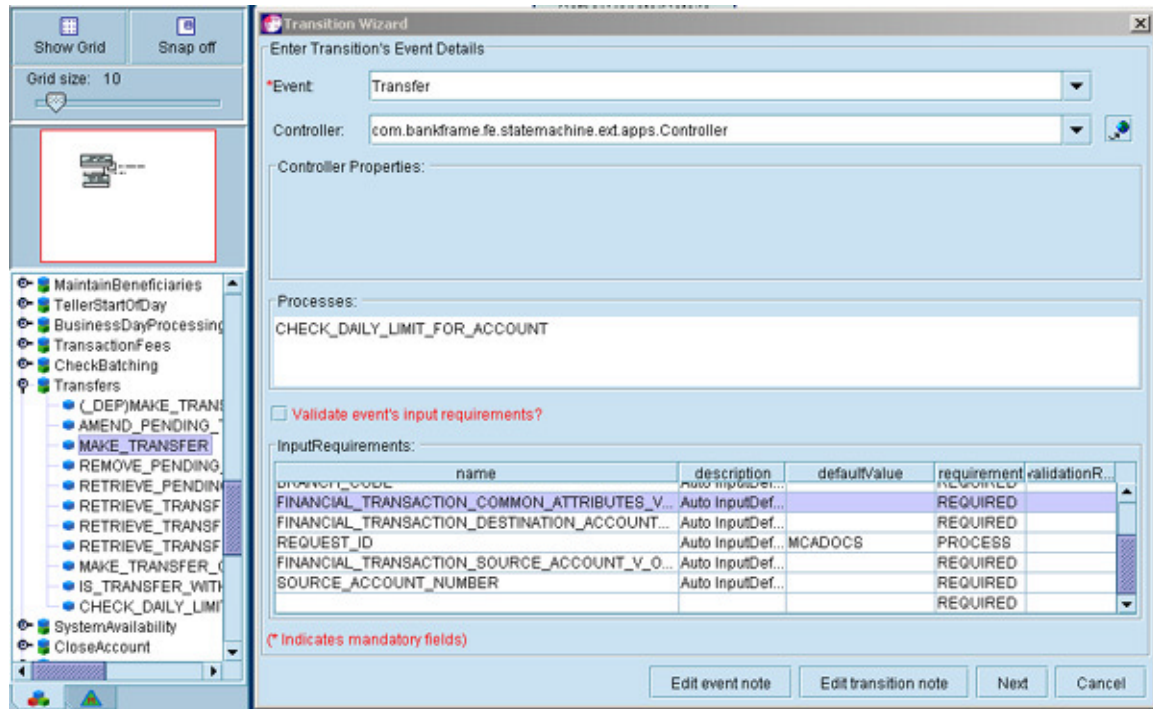
The guard condition on this transition is the opposite of the [AmountTooHigh] transition. Enter exactly the

same details, changing just the first to read 'Result must: contain the following'. That will ensure no matter what is returned from the CHECK_DAILY_LIMIT_FOR_ACCOUNT process one of the two guard conditions will be true. Set the guard description to something meaningful, such as 'AmountOK'.

This transition also has an action associated with it, as it must actually complete the transfer requested by the user. Enter an action description such as 'PerformTransfer', and drag the MAKE_TRANSFER process onto the transition dialog.



Press the 'Back' button now and you should see that the [InputRequirements](#) table has an extra line. The [MAKE_TRANSFER](#) process requires three more parameters than the [CHECK_DAILY_LIMIT_FOR_ACCOUNT](#) process, the destination account number VO (FINANCIAL_TRANSACTION_DESTINATION_ACCOUNT_VO_IMPL), some common system attributes VO (FINANCIAL_TRANSACTION_COMMON_ATTRIBUTES_VO_IMPL) and the source account VO (FINANCIAL_TRANSACTION_SOURCE_ACCOUNT_VO_IMPL). These are other parameters that must be supplied by the user, so make sure the last column for these three InputRequirements read 'REQUIRED'.



Finish that transition.

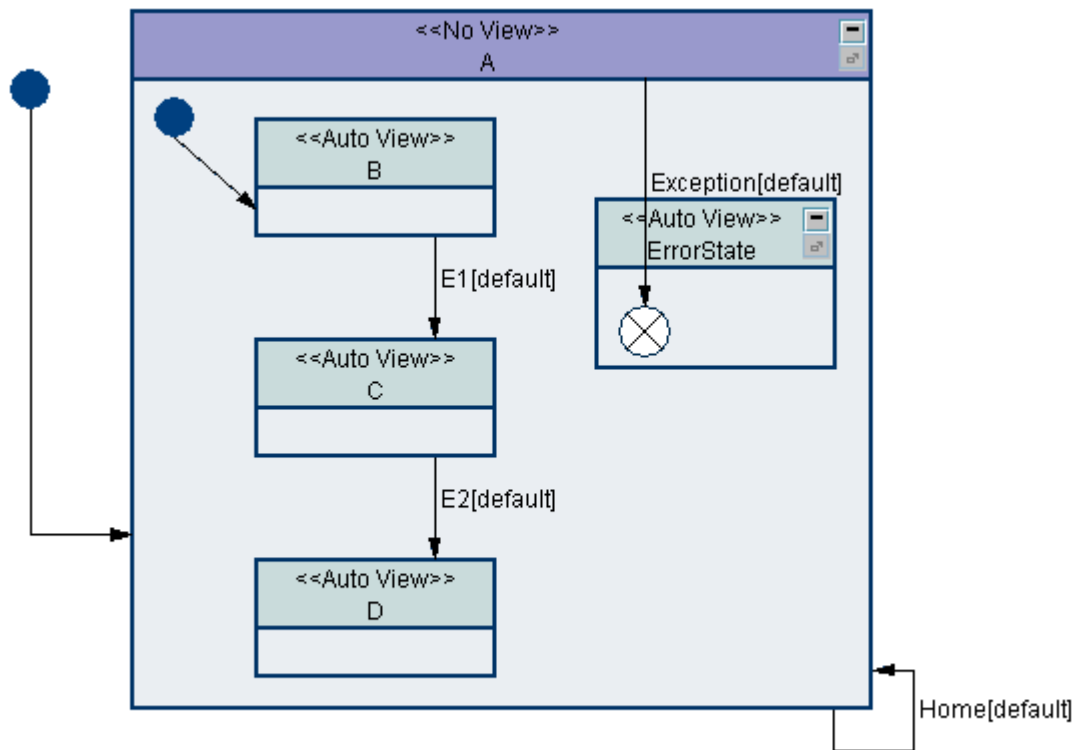
In this worked example you have created an event with two transitions, set guard conditions such that the correct transition will be followed, added a process to be called before the guard conditions are checked and another process to be called if one of the transitions is followed.

5.10 Blocking Events from States

In order to block events from states other than the current state the most important thing is that the Exception State is used in conjunction with setting the `block.states` key to `true` in the `BankframeResources.properties` file.

```
block.states=true
```

In the diagram below the states B, C and D will all inherit the "Exception" event. If an event is to be drawn to an exception state it must be called "Exception" (otherwise it will not work). When the C state is displayed and the back button is pressed on the browser the B state will be displayed. If the E1 event is then pressed the statemachine will throw an exception which will force the application to follow the "Exception" event and hence will display the ErrorState view. From the ErrorState the user can be brought back into the application via the "Home" event. The "ErrorState" can be anything you want, as long as the ExceptionState is a child of a viewable state.



Blocking Event from States Example

There is one scenario that cannot be blocked and that is when the user is on a state, then presses the back button and then the forward button on the browser. In this scenario the user can fire the event. The browser is back displaying the correct state and therefore the event will be allowed.

The `ScreenOrchestrator\resources\BankframeResources.properties` file will have to set the `block.states` to `true` at the end of the file, but when building the WAR double check that the WAR properties settings can find the relevant properties file. Also ensure that the WAR properties lib jars have also been updated with the current `mca.jar`

The `NewApplication.xml` file in the `StateMachineExamples.zip` shows how to use the event blocking feature.

6 Writing Controller Classes

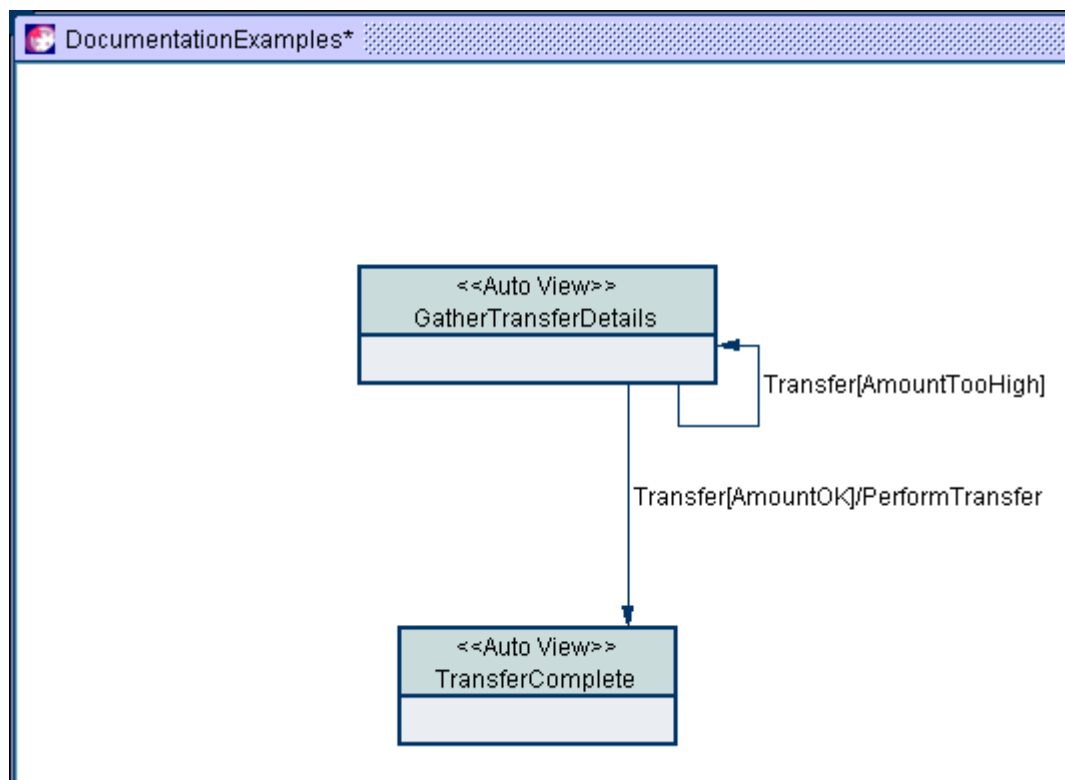
This section describes how to write Controller classes. It goes through the responsibilities of such classes, the APIs to be used, and some of the classes that are provided by default. It also includes a section on adding your controller classes to the editor.

6.1 The responsibilities of a Controller

Controller classes are the implementation of the control logic of the user interface. Controllers are the event handlers, with every event having a Controller configuration of its own.

Controllers have two basic responsibilities:

- To choose one of the event's transitions to follow.
When the Controller receives an event it must look at the data that has been supplied by the user and perhaps request information from the model. Based on this it must choose one of the event's transitions to follow. A Controller can choose a different transition each time the event is received, based on the business logic, the data supplied by the user and the state of the model.
- To perform any side effects required by that transition.
Having chosen a transition, the Controller must perform any side effects that are required by that transition.



For example: an application might allow a user to make a funds transfer between two accounts if the amount transferred in any single day is less than some limit. The Controller would first test that the amount is under

the limit and choose one of the two transitions based on that. If it chose the transition to the TransferComplete state it would then perform the side effect of completing the transfer.

6.2 The IController Interface

All Controller classes must conform to the `com.bankframe.fe.statemachine.base.apps.IController` interface.

This interface defines two methods reflecting the two primary responsibilities of the Controller classes:

- `IStateTransition getResult(RequestContext requestContext, IEvent event) throws StateMachineUserException`

This method represents the Controller's responsibility to choose between the transitions on the event. The `RequestContext` object contains all the information about the current user and request, while the `IEvent` object is the event that is to be processed. Typically this method is implemented as a series of calls to the model or tests on the input parameters, followed by a call to `event.getTransition(transitionName)` in order to get the `IStateTransition` object to return.

- `void doSideEffects(RequestContext requestContext, IStateTransition transition) throws StateMachineUserException`

This method represents the Controller's responsibility to perform any side effects required by the transition. The `requestContext` will be the same object as given to the `getResult` method, while the transition object will be the one returned by the `getResult` method.

Typically this will be implemented as a series of if/then/else if/ blocks testing the name of the transition. Within each should be the code to perform the required side effects.

Please refer to the MCA Services JavaDocs for further API details.

Assuming appropriate processes were deployed, the event above might be handled by the code below:

```
public IStateTransition getResult(RequestContext requestContext,
    IEvent event) throws StateMachineUserException {
    try {
        ChannelClient client =
ChannelClientFactory.getChannelClient();

        DataPacket requestData = new
DataPacket("CHECK_DAILY_LIMIT_FOR_ACCOUNT");

        requestData.put(DataPacket.REQUEST_ID,
TRANSFERS_REQUEST_ID);

        requestData.put("COMPANY_CODE",
requestContext.getRequest().getParameterValues("COMPANY_CODE")[0]);

        requestData.put("BRANCH_CODE",
requestContext.getRequest().getParameterValues("BRANCH_CODE")[0]);
```

```

        requestData.put("SOURCE_ACCOUNT_NUMBER",
requestContext.getRequest().getParameterValues("SOURCE_ACCOUNT_NUMBER")[0]
);

        requestData.put("AMOUNT",
requestContext.getRequest().getParameterValues("AMOUNT")[0]);

        Vector requestVector = new Vector(1);

        requestVector.add(requestData);

        Vector responseData = client.send(requestVector);

        if
(((DataPacket)responseData.firstElement()).getName().equals("AMOUNT_OK"))
{

            return event.getTransition("AmountOK");

        } else {

            return event.getTransition("AmountTooHigh");

        }

    } catch (ProcessingErrorException ex) {

        throw new StateMachineUserException(ex);

    }

}

public void doSideEffects(RequestContext requestContext,
IStateTransition transition) throws StateMachineUserException {

    if (transition.getName().equals("AmountOK")) {

        try {

            ChannelClient client =
ChannelClientFactory.getChannelClient();

            DataPacket requestData = new
DataPacket("TRANSFER_FUNDS");

            requestData.put(DataPacket.REQUEST_ID,
TRANSFERS_REQUEST_ID);

            requestData.put("COMPANY_CODE",
requestContext.getRequest().getParameterValues("COMPANY_CODE")[0]);

```

```

        requestData.put("BRANCH_CODE",
requestContext.getRequest().getParameterValues("BRANCH_CODE")[0]);

        requestData.put("SOURCE_ACCOUNT_NUMBER",
requestContext.getRequest().getParameterValues("SOURCE_ACCOUNT_NUMBER")[0]
);

        requestData.put("DEST_ACCOUNT_NUMBER",
requestContext.getRequest().getParameterValues("DEST_ACCOUNT_NUMBER")[0]);

        requestData.put("AMOUNT",
requestContext.getRequest().getParameterValues("AMOUNT")[0]);

        Vector requestVector = new Vector(1);

        requestVector.add(requestData);

        Vector responseData = client.send(requestVector);

    } catch (ProcessingErrorException ex) {

        throw new StateMachineUserException(ex);

    }

}

}

```

6.3 The SimpleController Class

The `com.bankframe.fe.statemachine.base.apps.SimpleController` class is an implementation of `IController` that is intended to control events with only one transition and no side effects. In many applications, particularly web applications, there will be many events that are simply navigation events. That is: they take the user from one screen to another and do nothing else. The `SimpleController` can handle all of these events.

6.4 The Main Controller Class

The Controller class that will be most commonly used is the `com.bankframe.fe.statemachine.ext.apps.Controller` class. This class is a complex and complete implementation of the `IController` interface that can use process, input requirements and guard condition data entered in the Orchestrator to carry out all the responsibilities of a Controller.

This class follows the steps below:

- The `getResult` method will call all of the processes defined for the event in the correct order, using data from the user input, input requirements and process definitions as appropriate to build up the `DataPackets` to be sent through MCA Services. Results from the process calls will be added to the user session.

- For each transition on the event, the Controller will call the `checkGuardCondition` method. Depending on the configuration of the transition, this could check the user inputs, the results from the processes, or other data in order to decide whether the transition should be followed.
- One of the transition `checkGuardCondition` methods should return `IGuardCondition.TRUE`. This transition will be returned from the Controller's `getResult` method.
- The `doSideEffects` method will call all of the processes defined for the transition in the correct order, using data from the user input, input requirements and process definitions as appropriate to build up the `DataPackets` to be sent through MCA Services. Again, results from the process calls will be added to the user session.

These steps should be enough to handle the majority of all events that will be included in an application user interface.

For those events that this will not handle, it is possible to extend the Controller class in various ways to add extra functions.

6.5 The Modified Controller Contract

The contract defined by the `IController` interface is a very general contract that can be used in any environment. The Controller class in the `com.bankframe.fe.statemachine.ext.apps` package provides a different definition of the `getResult` and `doSideEffects` methods geared more specifically to the Automated Methodology:

- `IStateTransition getResult(IEvent event, Inputs inputs, RequestContext requestContext) throws StateMachineUserException, ProcessingErrorException`
- `void doSideEffects(IEvent event, IStateTransition transition, Inputs inputs, RequestContext requestContext) throws StateMachineUserException, ProcessingErrorException`

Please refer to the MCA Services JavaDocs for further API details.

The `IEvent`, `IStateTransition` and `Inputs` objects passed into these methods have all been customized.

The `IEvent` and `IStateTransition` objects have a `getProcesses` method that returns an Iterator over all the processes associated with the event or transition. The `IEvent` object has a `getInputRequirements` method that makes available all the requirements and default values entered by a designer in the Orchestrator. `IStateTransition` includes a `checkGuardCondition` method, to test whether the transition's guard condition has been met.

6.5.1 The Inputs Object

The Inputs object provides a single view of all the data provided by the user or recorded previously in the current user's session. It combines three different data sources:

- The Request - contains the data entered by the user in the user interface before firing the current event.
- The Visit - will generally be empty, but may contain data placed there by another Controller or View. The visit is intended to store data that might be needed by a View. The visit is stored by the state machine so that it can be reloaded if we return to the same result state via a History or Deep History pseudo-state.
- The User Session - can store data about the user that might be required anywhere in the application. This may include details like the user's name, active role, actor ID, etc.

Inputs provides five methods for getting and setting parameter values:

- `Enumeration getParameterNames()` - this method provides an Enumeration over all the names of all the parameters in the three data sources.
- `Object getparameter(String parameterName)` - this method provides the value of the named parameter. It will look first in the request, then the visit, and finally the user session.
- `Object getparameter(String parameterName, int inputSource)` - this method provides the value of the named parameter in the specified input source. The input source must be one of `INPUT_SOURCE_ANY`, `INPUT_SOURCE_REQUEST`, `INPUT_SOURCE_VISIT` or `INPUT_SOURCE_USER_SESSION`.
- `void setParameter(String parameterName, Object parameterValue)` - this method sets a parameter value in the request.
- `void setParameter(String parameterName, Object parameterValue, int inputSource)` - this method sets a parameter in the specified input source.

6.6 Extending The Controller Class

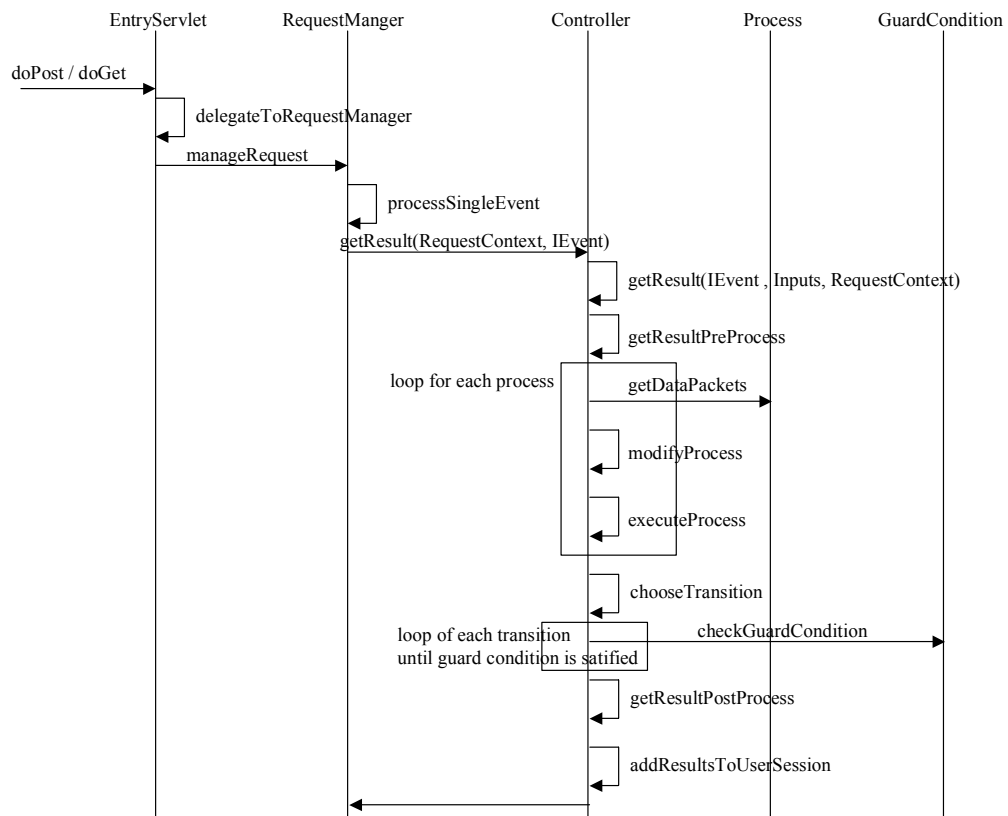
In addition to the steps described above, the Controller class calls a number of empty methods at different times during the processing. You can override these empty methods to add extra functionality.

The full sequence of method calls is:

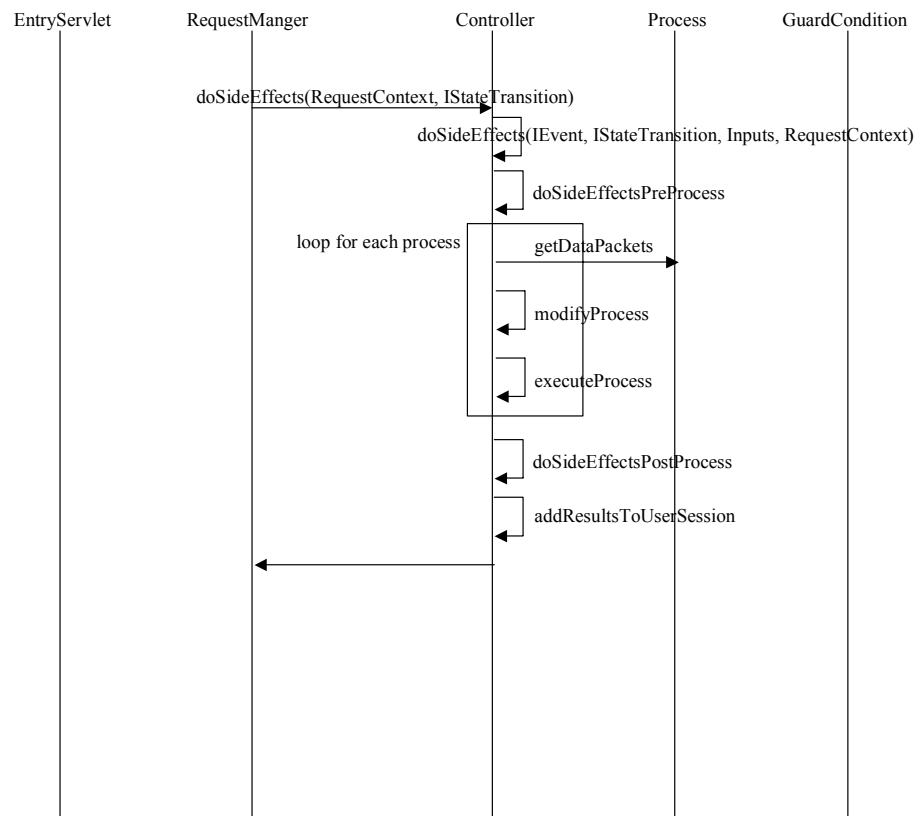
- The framework calls `getResult`.
- `getResult` calls `getResultPreProcess` - override `getResultPreProcess` if you need to manipulate the user inputs or perform any other tasks before the controller does anything.
- For each process in the event:
 - `getResult` calls `modifyProcess` - override `modifyProcess` to change the automatically-produced request DataPackets. Note that `modifyProcess` may be called many times by `getResult` and by `doSideEffects`, so make sure you are modifying the correct process call!
 - `getResult` calls `executeProcess`.

- getResult calls chooseTransition - override chooseTransition if you want to choose the transition yourself, instead of using the transition.checkGuardCondition methods. You must override chooseTransition if any of the transition guard conditions might return IGuardCondition.UNDEFINED.
- getResult calls getResultPostProcess - override getResultPostProcess if you need to extract certain pieces of information from the process responses, or if you want to change the default behaviour of adding the response data to the user session.
- getResultPostProcess calls addResultsToUserSession.
- getResult returns the chosen transition to the framework.
- The framework calls doSideEffects.
- doSideEffects calls doSideEffectsPreProcess - override doSideEffectsPreProcess if there is anything you need to do before the side effects are performed.
- For each process in the transition:
 - doSideEffects calls modifyProcess - this is the same modifyProcess method as is called by getResult, so be careful when overriding it to modify only the processes you need to.
 - doSideEffects calls executeProcess.
- doSideEffects calls doSideEffectsPostProcess - this is your last chance to change the behaviour of the controller.
- doSideEffectsPostProcess calls addResultsToUserSession.

The following sequence diagrams show the previously mentioned methods and where they are called when a `com.bankframe.fe.statemachine.ext.apps.Controller`'s `getResult` and `doSideEffect` methods are invoked.



The com.bankframe.fe.statemachine.ext.apps.Controller sequence diagram.



The com.bankframe.fe.statemachine.ext.apps.Controller sequence diagram continued.

Full details on all of these methods, including the method signatures, can be found in the MCA Services JavaDocs.

6.7 Adding A New Controller To The Orchestrator

When you create a new controller class, there are several ways to include it in the editor:

6.7.1 Do Nothing

It is possible to do nothing at all. When a designer needs to use the new controller they type the name into the 'Controller' combo box on the transition dialog.

6.7.2 Add the controller to the statechart.properties file

The controller classes loaded into the transitions dialog box are listed in the statechart.properties file. The file contains three controllers by default:

- `com.bankframe.fe.statemachine.base.apps.SimpleController`
- `com.bankframe.fe.statemachine.base.apps.AutoViewController`
- `com.bankframe.fe.statemachine.ext.apps.Controller`
- `com.eontec.statemachine.helpers.ChannelClientController`

- `com.eontec.statemachine.helpers.DataCollectorController`
- `com.eontec.statemachine.helpers.ClearUserSessionController`
- `com.eontec.statemachine.helpers.AddToUserSessionController`

To add a new controller, type the controller class name in the 'Controller' combo box on the transition dialog. On the right-hand side of the combo box is a register controller button as highlighted in the next figure.

Transition Wizard

Enter Transition's Event Details

*Event: Login

Controller: com.eontec.statemachine.helpers.DataCollectorController

Controller Properties:

Channel Client Name: com.bankframe.ei.channel.client.HttpClient

Data store: User session

Key names:

Enter a "*" delimited list of data packet names that should be fired in the request excluding the header datapacket. Leave blank to send all data packets in the request.

Processes:

Register controller class permanently

☐ Validate event's input requirements?

Input Requirements:

name	description	defaultValue	requirement	validationRule
			REQUIRED	

(* Indicates mandatory fields)

Edit event note Edit transition note Next Cancel

Highlighted register controller button.

The 'Controller' combo box on the transition dialog will now contain your new controller class in the default list.

6.7.3 Create a Customizer for the Controller

The 'Controller Properties' box on the transitions dialog is managed by loading bean customizer classes for the controller class selected. If you create a customizer for your controller class and add it to the classpath

for the editor, the dialog will load your customizer. This allows you to completely control how your controller looks in the editor. The `controllerProperties` will contain an entry for each attribute exposed by the bean. For information on creating a customizer, see the [JavaBeans API](#) and [documentation](#).

7 Writing Guard Condition Classes

This section describes how to write guard condition classes. It goes through the responsibilities of such classes, the APIs to be used, and some of the classes that are provided by default. It also includes a section on adding your guard condition classes to the editor.

7.1 The Responsibility Of A Guard Condition

A guard condition class has a very simple responsibility: to decide whether a transition should be followed in any given case.

7.2 The IGuardCondition Interface

All GuardCondition classes must implement the `com.bankframe.fe.statemachine.ext.apploders.IGuardCondition` interface. This interface defines two methods for you to implement:

- `int checkGuardCondition(Inputs inputs, Vector processExecutionRecords, RequestContext requestContext, IStateTransition stateTransition)` - this method must return either `IGuardCondition.TRUE` or `IGuardCondition.FALSE`. If it returns `TRUE`, the transition will be followed, if `FALSE`, the transition will not be followed.
The `Inputs`, `RequestContext` and `IStateTransition` objects, supplied are the same as described above for the `Controller` class. The `processExecutionRecords` vector contains `com.bankframe.fe.statemachine.ext.apps.ProcessExecutionRecord` objects, listing the details of all the processes executed by the `Controller` before calling `checkGuardCondition`.
- `void setGuardConditionProperties(Properties guardConditionProperties)` - this method will be called before `checkGuardCondition`. The `guardConditionProperties` will contain any information provided by the designer.

7.3 Adding A New Guard Condition To The Orchestrator

When you create a new guard condition class, there are several ways to include it in the editor:

7.3.1 Do Nothing

It is possible to do nothing at all. When a designer needs to use the new guard condition they type the name into the 'Guard Condition Class' combo box on the transition dialog. They will be presented with a standard Properties editor in which they can enter the guard condition properties.

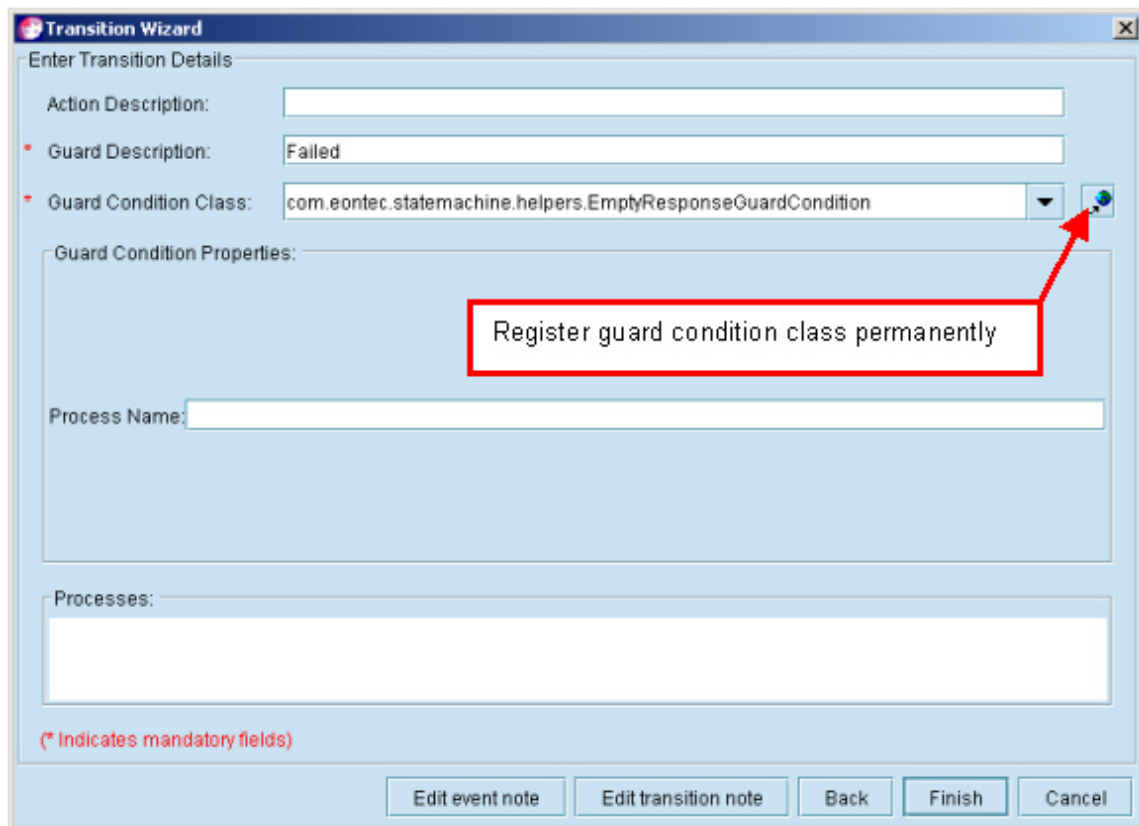
7.3.2 Add the guard condition to the statechart.properties file

The guard condition classes loaded into the transitions dialog box are listed in the `statechart.properties` file. The file contains three guard conditions by default:

- `com.bankframe.fe.statemachine.ext.apploders.bean.ResultBasedGuardCondition`

- `com.bankframe.fe.statemachine.ext.apploders.bean.InputBasedGuardCondition`
- `com.bankframe.fe.statemachine.ext.apploders.bean.FixedValueGuardCondition`
- `com.eontec.statemachine.helpers.TimeoutGuardCondition`
- `com.eontec.statemachine.helpers.EmptyResponseGuardCondition`

To add a new guard condition, type the guard condition class name in the 'Guard Condition Class' combo box on the transition dialog. On the right-hand side of the combo box is a register guard condition button as highlighted in the next figure.



Highlighted register guard condition button.

The 'Guard Condition Class' box on the transition dialog will now contain your new guard condition class in the default list.

7.3.3 Create a customizer for the guard condition

The 'Guard Condition Properties' box on the transitions dialog is managed by loading bean customizer classes for the guard condition class selected. If you create a customizer for your guard condition class and add it to the classpath for the editor, the dialog will load your customizer. This allows you to completely control how your guard condition looks in the editor.

The `guardConditionProperties` will contain an entry for each attribute exposed by the bean. For information on creating a customizer, see the JavaBeans API and documentation.

8 Writing JSPs

This section describes the responsibilities of a JSP in the orchestrator framework, and goes through the beans and tags available to help build JSPs.

8.1 Responsibilities of a JSP

In this framework a JSP has two very simple responsibilities:

- Display data to the user - the JSP is required to format and display the data the user expects to see, including all the formatting, framing, branding and general prettiness that is required in the user interface.
- Give the user the opportunity to fire events - for every state the user interface is in, there will be events that the user can fire. The JSP must provide buttons, links or similar widgets for the user to allow events to be fired.

Within these two simple responsibilities, how you code the JSP is very flexible. There is, however, one thing that you must not do:

- You must not include any tests in the JSP that result in loading new pages, forwarding or redirecting the JSP. If there is ever a circumstance where you want to redirect or forward to another JSP based on some test in the JSP, you must change the state chart design so that the test is handled in a Controller class.

8.2 Getting data into the JSP

The JSP is required to display data to the user. This data is made available to the JSP through a set of beans placed in the servlet request context.

8.2.1 Inputs bean

The Inputs bean contains all the data included in the request from the user, the current user session and the current state visit. It is loaded using the tag:

```
<jsp:useBean id="Inputs" scope="request"
class="com.bankframe.fe.statemachine.ext.apps.Inputs" />
```

It is also useful to import the Inputs class to reference static members:

```
<%@ page import="com.bankframe.fe.statemachine.ext.apps.Inputs" %>
```

Inputs provides five methods for getting and setting parameter values:

- `Enumeration getParameterNames()` - this method provides an Enumeration over all the names of all the parameters in the three data sources.
- `Object getParameter(String parameterName)` - this method provides the value of the named parameter. It will look first in the request, then the visit, and finally the user session.
- `Object getParameter (String parameterName, int inputSource)` - this method provides the value of the named parameter in the specified input source. The input source must be

one of `INPUT_SOURCE_ANY`, `INPUT_SOURCE_REQUEST`, `INPUT_SOURCE_VISIT` or `INPUT_SOURCE_USER_SESSION`.

- `void setParameter(String parameterName, Object parameterValue)` - this method sets a parameter value in the request.
- `void setParameter (String parameterName, Object parameterValue, int inputSource)` - this method sets a parameter in the specified input source.

See the JavaDocs for further details on the API

8.2.2 ProcessExecutionRecords bean

The ProcessExecutionRecords bean is a Vector of ProcessExecutionRecord objects, containing all of the processes executed while handling the current event. The responses from these processes, available as Vectors of DataPackets, will contain all the data retrieved from the server by the Controller or View classes.

Load the ProcessExecutionRecords bean with the tag:

```
<jsp:useBean id="ProcessExecutionRecords" scope="request"
class="java.util.Vector" />
```

8.2.3 State bean

The State bean is the state that is to be displayed by the JSP. It is possible to use the same JSP to display different states, and the State bean will give you the current stateId or title.

Load the state bean with the tag:

```
<jsp:useBean id="State" scope="request"
class="com.bankframe.fe.statemachine.ext.apploaders.IState" />
```

8.2.4 View bean

The View bean is the View class that is including the JSP.

This can be loaded with the tag:

```
<jsp:useBean id="View" scope="request"
class="com.bankframe.fe.statemachine.ext.connectors.servlet.JSPView" />
```

8.2.5 RequestContext bean

The RequestContext bean contains other miscellaneous objects, including the statemachine configuration, the application loader, user session, user session manager and logger. You will probably not need this bean in most cases.

The RequestContext can be loaded with the tag:

```
<jsp:useBean id="RequestContext" scope="request"
class="com.bankframe.fe.statemachine.base.RequestContext" />
```

8.3 Firing an event from a JSP

The JSP will need to supply the user with buttons or links to fire events. There are two distinct mechanisms you can use to fire these events. Do not mix these two mechanisms. If you start using one of these two approaches, keep using that one. Any attempt to mix them will cause events to fail.

8.3.1 Using the .jsm URL extension

The statemachine servlet is configured to respond to all requests that end with .jsm. It expects the stateId and event name to be supplied in the URL in the form <stateId>.<event name>.jsm.

This URL format can be used on both simple links and forms, using the following code:

```
<a href="<jsp:getProperty name="State" property="id" />.event.jsm"> event
</a>
```

```
<form action="<jsp:getProperty name="State" property="id" />.event.jsm">
```

```
...
```

```
</form>
```

Replace 'event' in these code samples with the correct event name.

8.3.2 Using the StateMachine URL

The statemachine servlet is also configured to respond to requests with the URL /StateMachine (relative to the web application root). You can retrieve the absolute URL from the View bean.

You must supply two parameters with this URL called 'statemachineEventName' and 'statemachineStateName'. Use the following code as a guide:

```
<a href="<jsp:getProperty name="View" property="requestURL"
/>?statemachineStateName=<jsp:getProperty name="State" property="id"
/>&statemachineEventName=event">event</a>
```

```
<form action="<jsp:getProperty name="View" property="requestURL" />">
```

```
<input type="hidden" name="statemachineStateName" value="<jsp:getProperty
name="State" property="id" />">
```

```
<input type="hidden" name="statemachineEventName" value="event">
```

```
...
```

```
</form>
```

Replace 'event' in these code samples with the correct event name.

9 Orchestrator Process Integration

9.1 Introduction

The Orchestrator supports the ability to hook financial processes to the front end components generated by the tool.

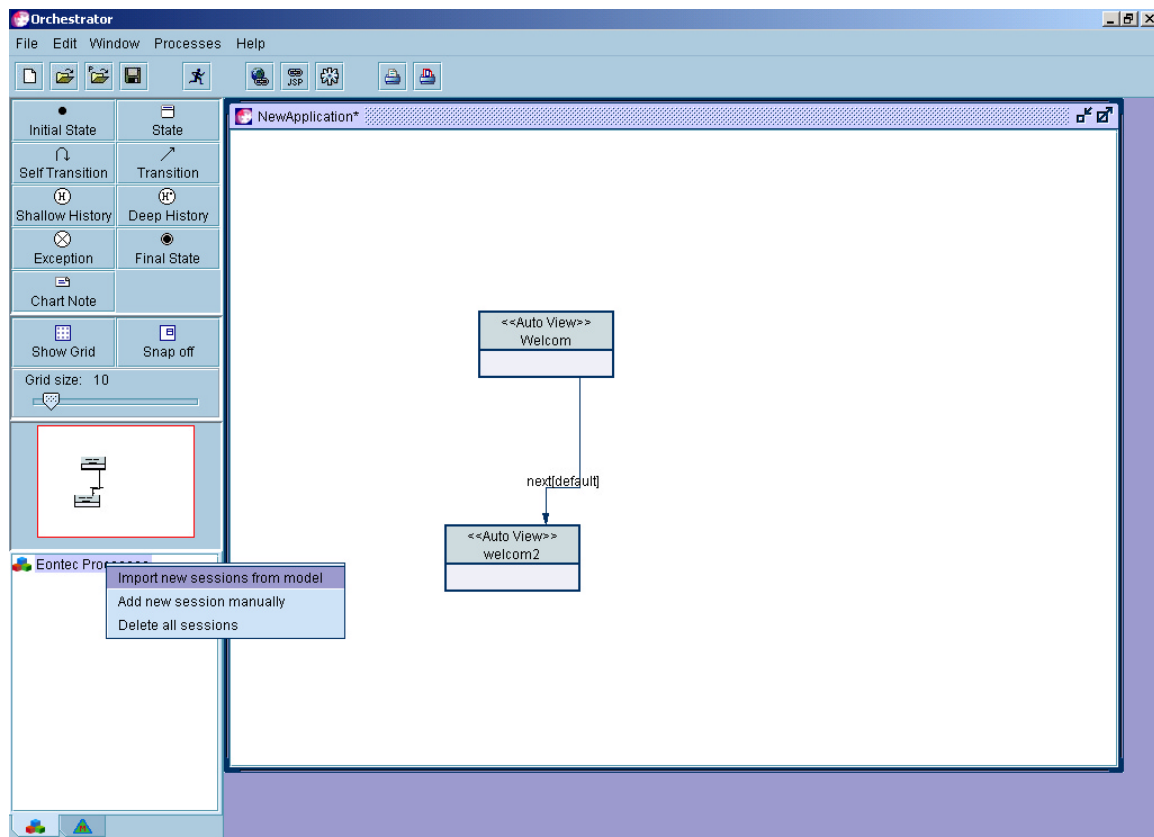
In terms of this tool, a process is a unit of work performed within a deployed session bean. There can therefore be many processes within one session. The manner in which the methods are called being tweaked by the value of the DATA PACKET NAME key being passed in. In this sense the tool regards a session as an encapsulation of one or many financial processes.

Process integration within the tool generally consists of two steps

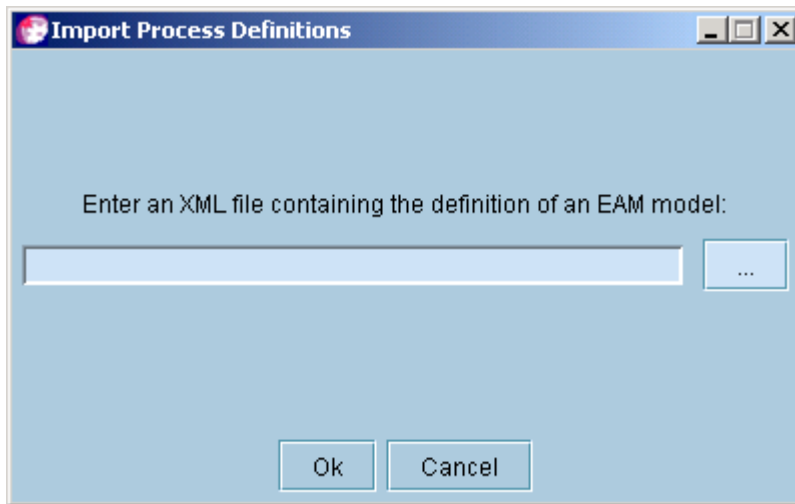
- Importing in processes from an external source, either manually or from an Automated Methodology model.
- Associating these processes with events and/or transitions.

9.2 Importing in processes from an Automated Methodology model

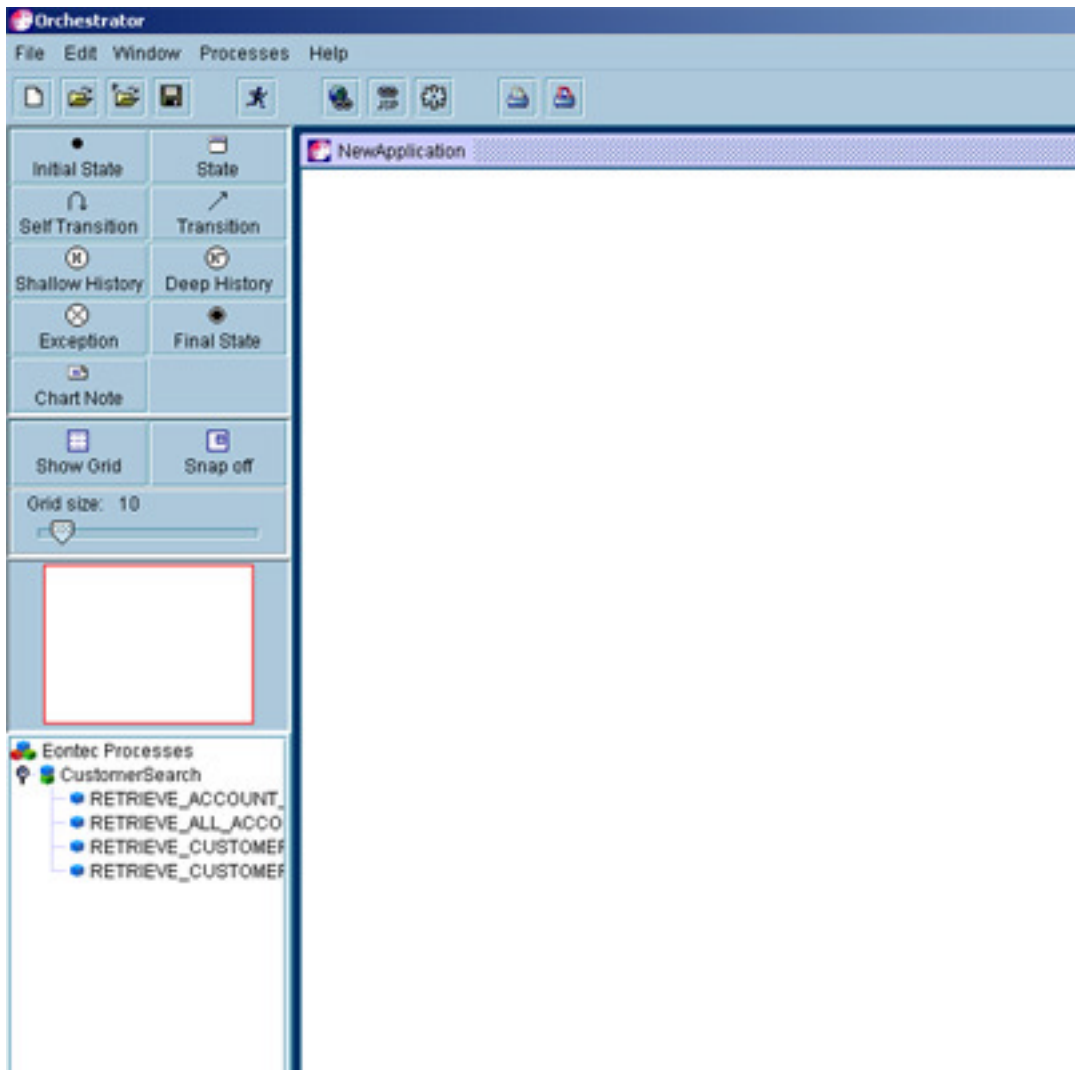
To import in processes from an Automated Methodology model, right click on the root node of the “Siebel Processes” node and select “Import new sessions from model”.



You should then be presented with the following dialog:



Click the button to the right of the text field and point the file chooser to an xml file representing an Automated Methodology model. If you wish, you can use the sample xml file shipped with the application. It lies in the "xml" folder that the file chooser defaults into and is called "RetailAccount.xml". When you have done this, the process tree should be populated with process information and resemble the following:



When processes are imported into the Screen Orchestrator from an Automated Methodology model, the session name, process name, process signature and process return type are automatically converted to the key/value pairings required for Request and Response DataPackets. Functional parameter objects, non-functional parameter objects, banking objects and primary key classes used in the signature of a process are also automatically converted to the expected Request and Response DataPacket format. If a parameter is not defined as a result of the import process, it may be that the class type is not defined correctly in the Automated Methodology model.

9.3 Manually inputting process information

To manually input a process right click on the root node of the process tree and select “Add new session manually”. You should then be presented with the following dialog:

Enter session name

Session Name

Back Next Finish

Key in a session name and then click next. You should then be presented with the following:

Process 1, set details

Request Data Packet

+ -

Process Name

KEY	SAMPLE VALUE
DATA PACKET NAME	
REQUEST_ID	

Response Data Packet

+ - Vector ☐

KEY	DATA PACKET NAME

Back Next Finish

Input all relevant information. To add or remove fields use the +/- buttons. To state whether the response is a vector or single DataPacket type, use the checkbox.

The above dialog refers to one process contained in the session. As explained earlier there can be many processes to a session. Therefore, if one wishes to add another process, click the next button and you will see another copy of this panel representing another process on this session. Note also that you must supply a DataPacket name value with each process. This is used to name the node on the tree. When you are finished, click OK. The process information should be added to the process tree.

9.4 Removing all sessions from the Siebel Process list

To remove all the sessions in the Siebel Process list, right click on any process node and select “Delete all sessions”. A dialogue box will appear to confirm if you wish to remove all the process definitions, click ‘Yes’ to confirm.

9.5 Editing/Deleting processes

To edit the details of any process, right click on any process node and select “Edit Process”. You should get a dialog similar to the following, with different process information:

Edit Process: retrieveAccountDetailsForBranchCodeAndAccountNumber

Request Data Packet

KEY	SAMPLE VALUE
companyCode	
branchCode	
accountNumber	
userId	
DATA PACKET NAME	retrieveAccountDetailsFo...

Response Data Packet

Vector ☐

KEY
COMPANY_CODE
ACCOUNT_NUMBER
ACCOUNT_CURRENCY
ACCOUNT_BALANCE
BRANCH_CODE
ACCOUNT_NAME
PRODUCT_ID
UNCLEARED_FUNDS_VALUE
JOINT_ACCOUNT_INDICATOR
OPEN_ACCOUNT_DATE
NEXT_CHEQUEBOOK_FACILITY_NUMBER
MANDATE_TYPE
NEXT_MOVEMENT_NUMBER
NEXT_CARD_CONTROL_NUMBER
CLOSURE_REASON

Ok

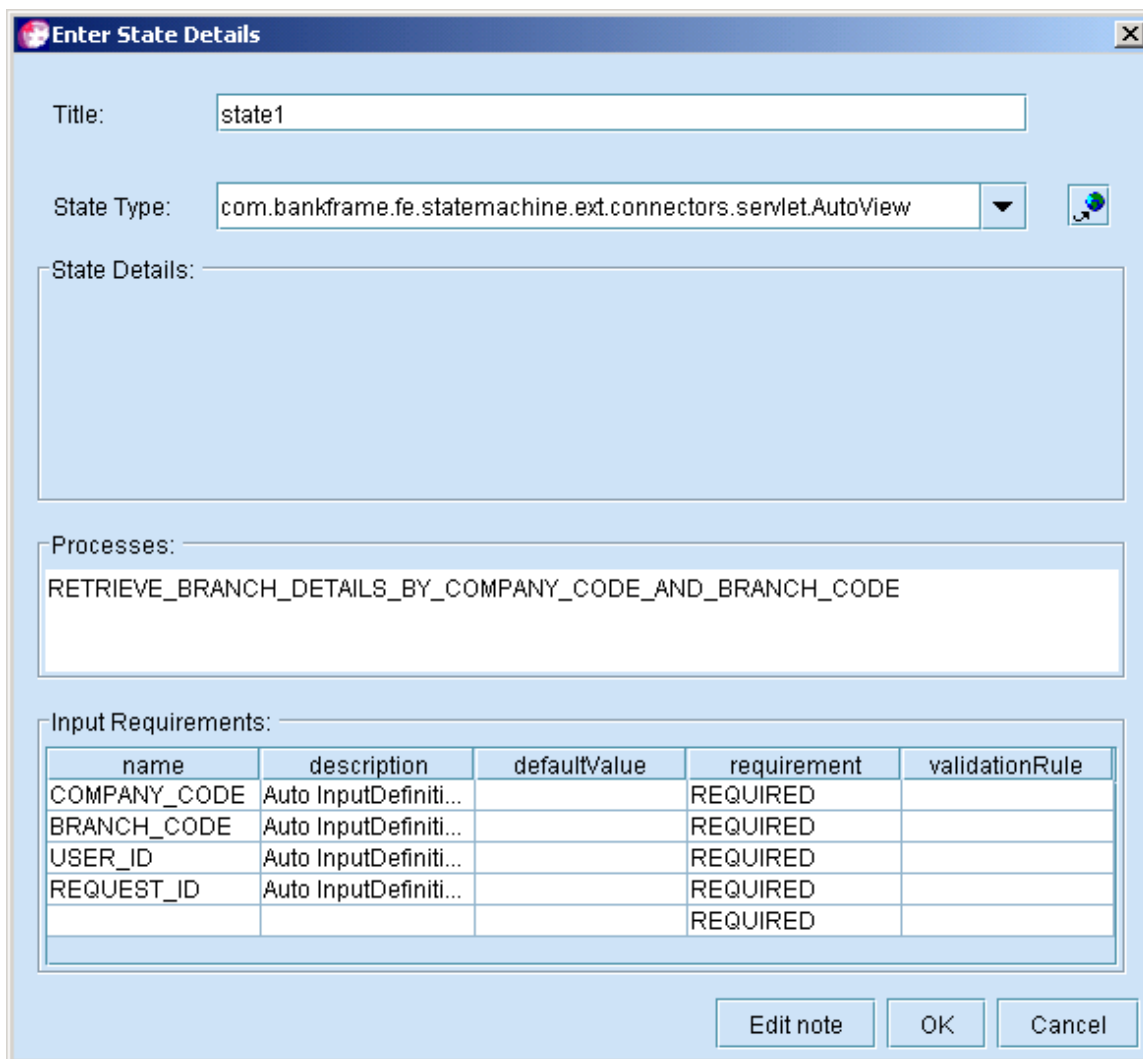
Edit the process details and then click ok.

To delete a process, right click on it on the tree and select “Delete Process”.

9.6 Assigning processes to the state chart


9.6.1 Assigning processes to a state

To assign a process to a state, go to the process tree, hold down the left mouse button and drag the process over a state on the chart, then release the left mouse button. You will then be presented with a dialog resembling the following:



Enter State Details

Title:

State Type: 

State Details:

Processes:

RETRIEVE_BRANCH_DETAILS_BY_COMPANY_CODE_AND_BRANCH_CODE

Input Requirements:

name	description	defaultValue	requirement	validationRule
COMPANY_CODE	Auto InputDefiniti...		REQUIRED	
BRANCH_CODE	Auto InputDefiniti...		REQUIRED	
USER_ID	Auto InputDefiniti...		REQUIRED	
REQUEST_ID	Auto InputDefiniti...		REQUIRED	
			REQUIRED	

If you wish to add further processes to this state, drag them on to this dialog from the process tree and drop them either onto the white text area entitled “Processes” or drop them onto the table entitled “Input Requirements”. Click OK to save the process details.

9.6.2 Adding Processes to a State Transition

To add processes to a state transition, make a new transition between two states. You will be presented with the following dialog:

Transition Wizard

Enter Transition's Event Details

*Event:

Controller:

Controller Properties:

☐ Validate event's input requirements?

InputRequirements:

name	description	defaultValue	requirement	validationRule
			REQUIRED	

(* Indicates mandatory fields)

Enter the event name and select a controller from the drop-down list (other than [SimpleController](#) or [AutoViewController](#), which are used only for simple state navigation and do not invoke processes).

Transition Wizard

Enter Transition's Event Details

*Event:

Controller:

Controller Properties:

Processes:

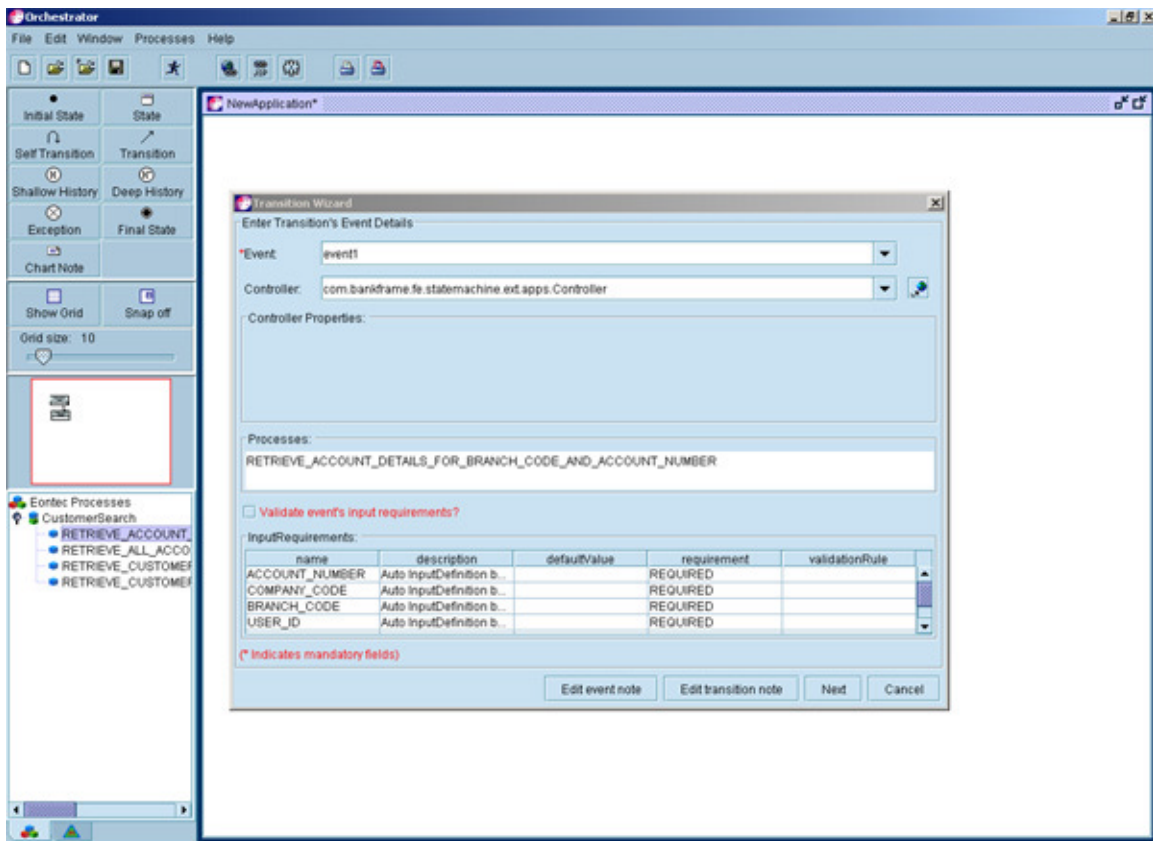
☐ Validate event's input requirements?

InputRequirements:

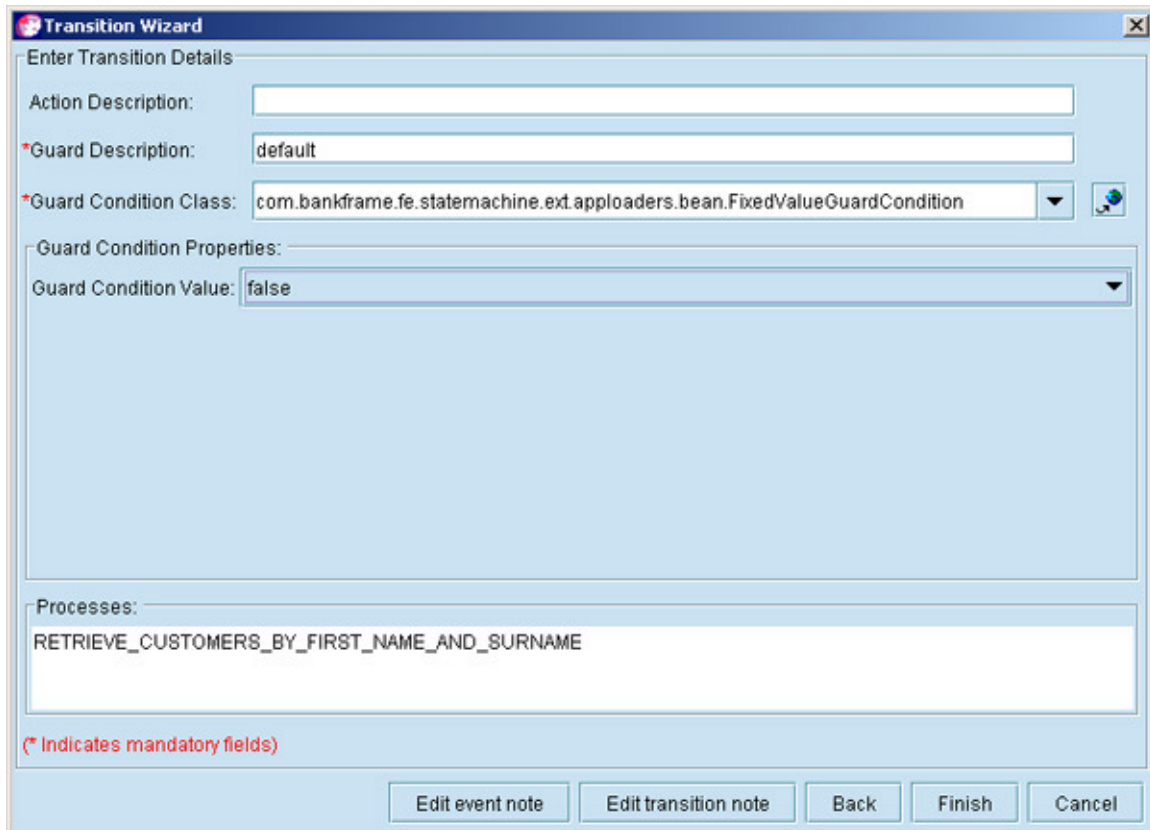
name	description	defaultValue	requirement	validationRule
			REQUIRED	

(* Indicates mandatory fields)

To add processes to this transition, drag and drop them from the tree, into the text area entitled "Processes". This will add them to the transition's event details.



You can also add processes to the transition details (as opposed to the transition event details). This will invoke the process as a side effect on the transition. To do this, click next on the above dialog and drag and drop the processes on to the text area entitled "Processes".



The image shows a 'Transition Wizard' dialog box with a title bar and a close button. It contains several input fields and a list of processes. The 'Enter Transition Details' section includes an 'Action Description' field, a '*Guard Description' field with the value 'default', and a '*Guard Condition Class' dropdown menu showing 'com.bankframe.fe.statemachine.ext.apploders.bean.FixedValueGuardCondition'. Below these is a 'Guard Condition Properties' section with a 'Guard Condition Value' dropdown menu set to 'false'. The 'Processes' section contains a list box with the text 'RETRIEVE_CUSTOMERS_BY_FIRST_NAME_AND_SURNAME'. At the bottom, there is a red note '* Indicates mandatory fields' and five buttons: 'Edit event note', 'Edit transition note', 'Back', 'Finish', and 'Cancel'.

Transition Wizard

Enter Transition Details

Action Description:

*Guard Description:

*Guard Condition Class:

Guard Condition Properties:

Guard Condition Value:

Processes:

RETRIEVE_CUSTOMERS_BY_FIRST_NAME_AND_SURNAME

(* Indicates mandatory fields)

Edit event note Edit transition note Back Finish Cancel

Transition Details screen

For more information on how processes are managed by the state machine framework after they have been assigned, please refer to the “Designing Events With Processes And Guard Conditions” section.

10 Advanced Drawing

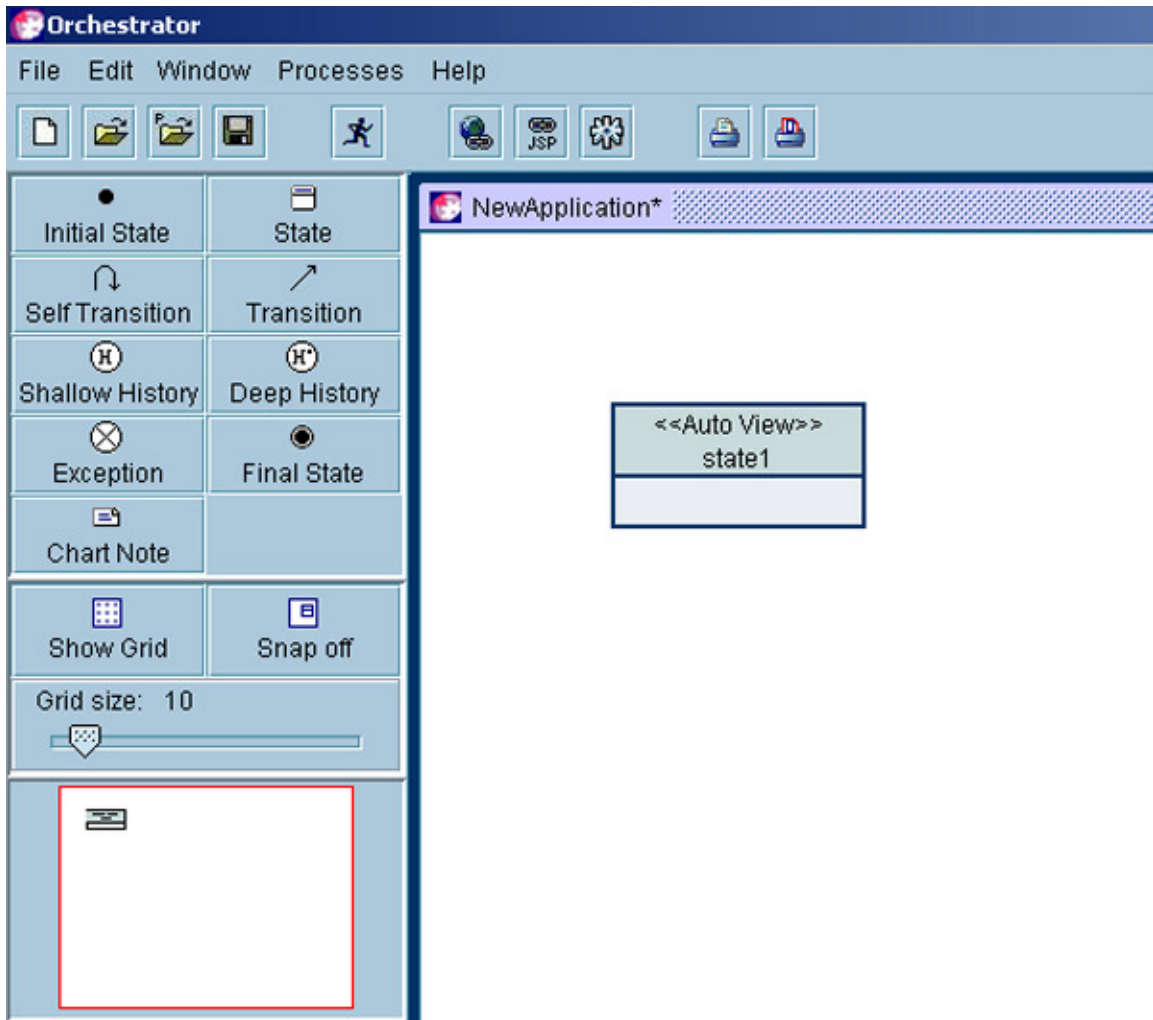
The following section provides details of the advanced drawing capabilities of the orchestrator tool.

10.1 Undo / Redo Features

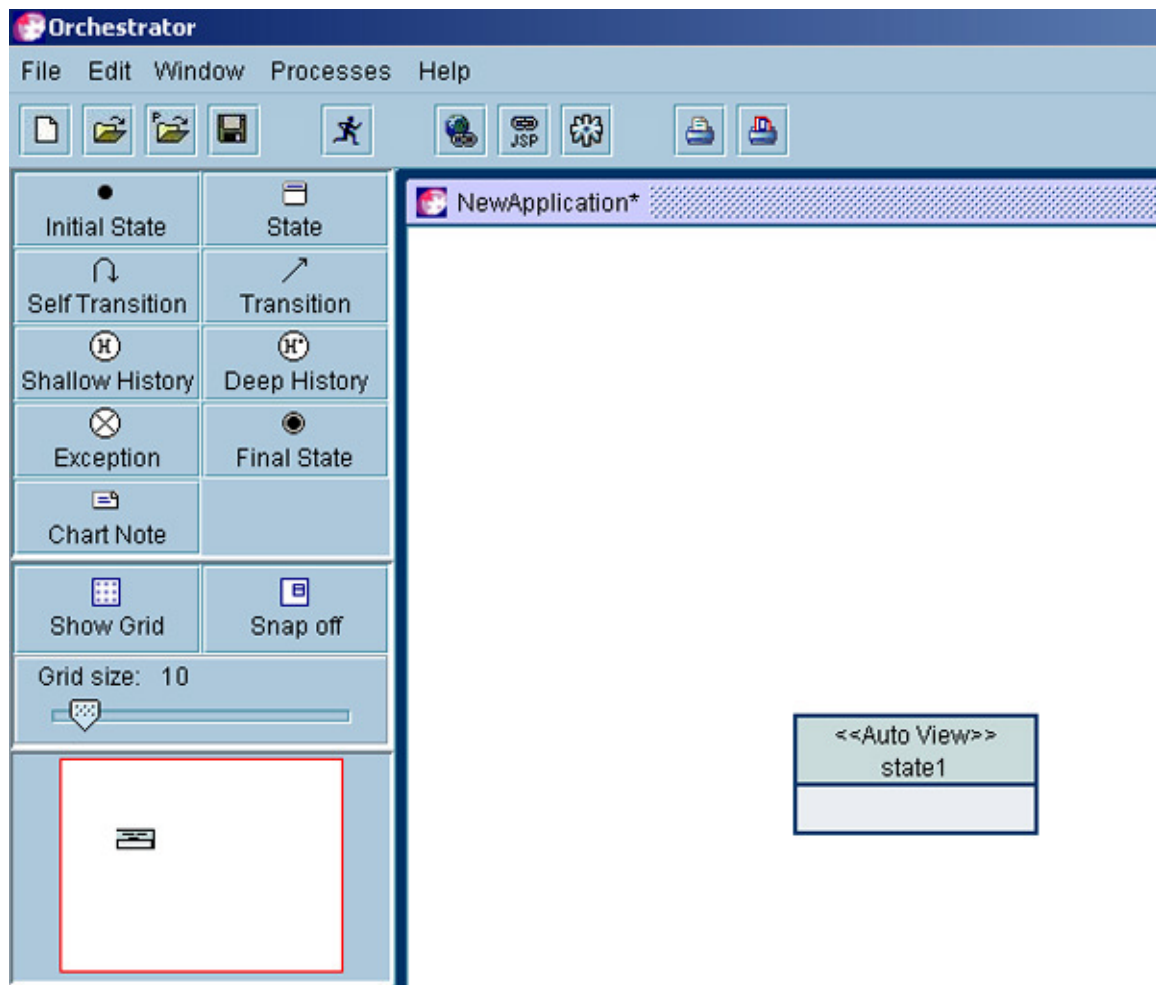
The Orchestrator tool provides the ability to undo and redo drawing instructions. If the user creates, moves, edits or deletes a state then the tool can undo that change. The tool can also undo any transition create, edit or delete. Only the last 5 drawing instructions can be undone. The tool provides an Edit menu with Undo and Redo menu items to access these features. The textual description of the Undo and Redo menu items change as the user creates undo and redo instructions. The textual description is useful for the user to determine what instruction will be undone or redone if the menu item is selected.

10.1.1 Undo Example

This section will show an example of using the undo feature to undo a move instruction. The first figure that follows shows the original location of the test1 state. The second figure indicates the new position of this state.

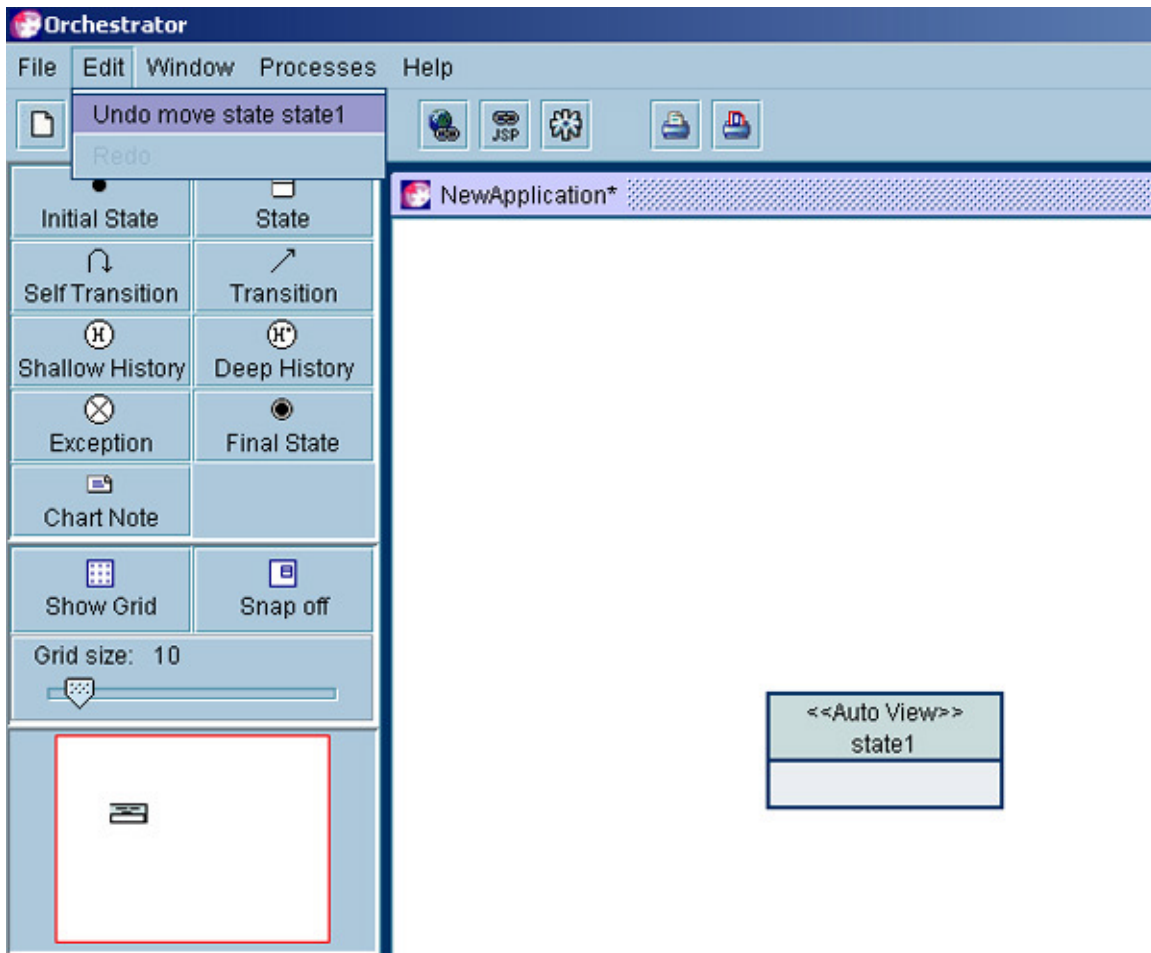


The original location of the test1 state.



The new location of the test1 state.

This move instruction will now be undone by selecting **Edit -> Undo move state test1** as highlighted in the next figure.

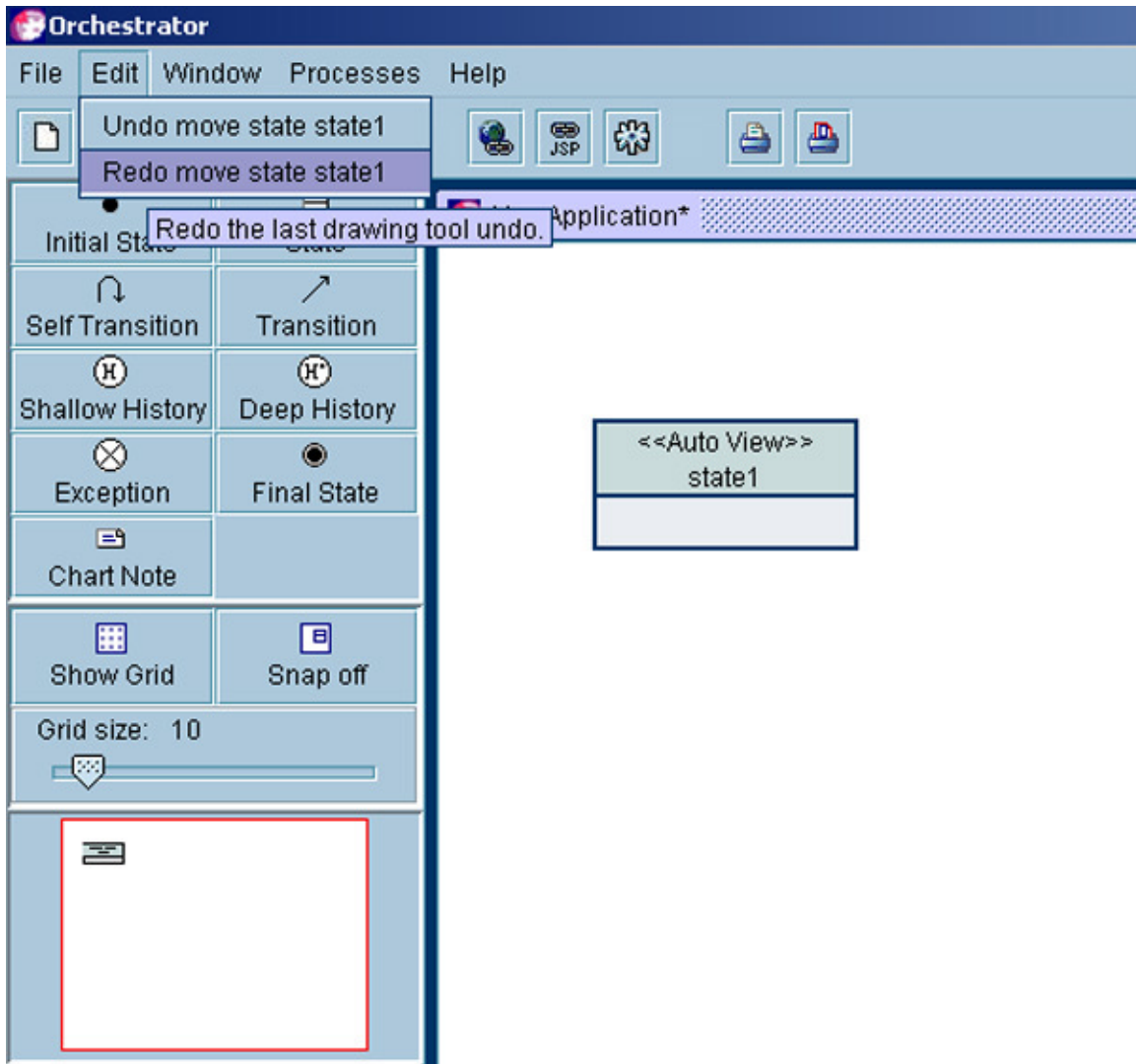


The “Undo move state test1” menu item highlighted.

Once the “Undo move state test1” menu item is selected the test1 state will be returned to its original position as shown in the first diagram of this section.

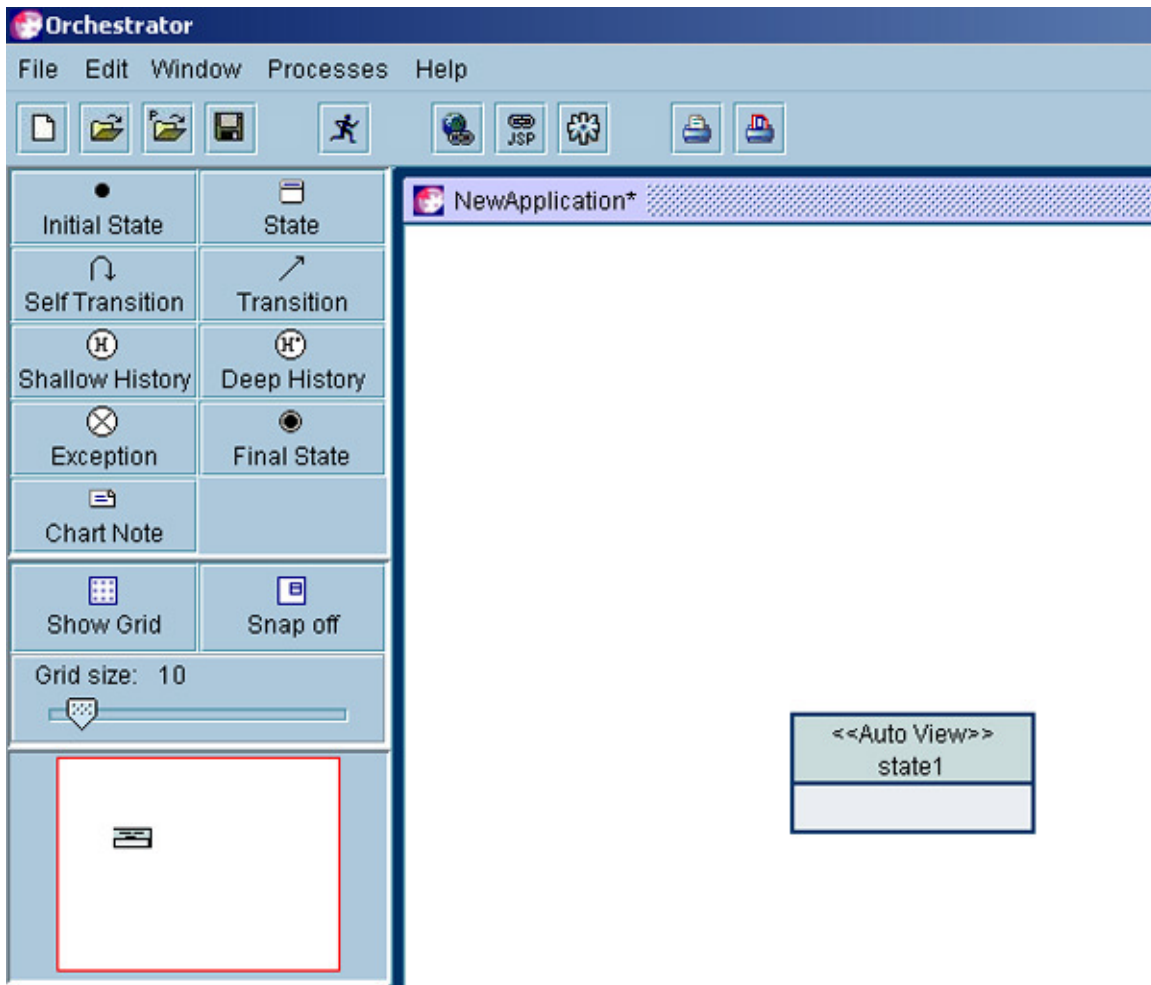
10.1.2 Redo Example

The undo example as described in the previous section will be used as the basis of this example. We will now redo the previous undo so that the move instruction is done as originally specified. When an instruction is ‘undo’, the redo menu item is changed so that its textual description details what undo instruction will be redone. The next figure indicates the “Redo move state test1” menu item.



The “Redo move state test1” menu item highlighted.

Once the “Redo move state test1” menu item is selected the test1 state move will be done and the statechart will look as shown below.



The test1 state move redone.

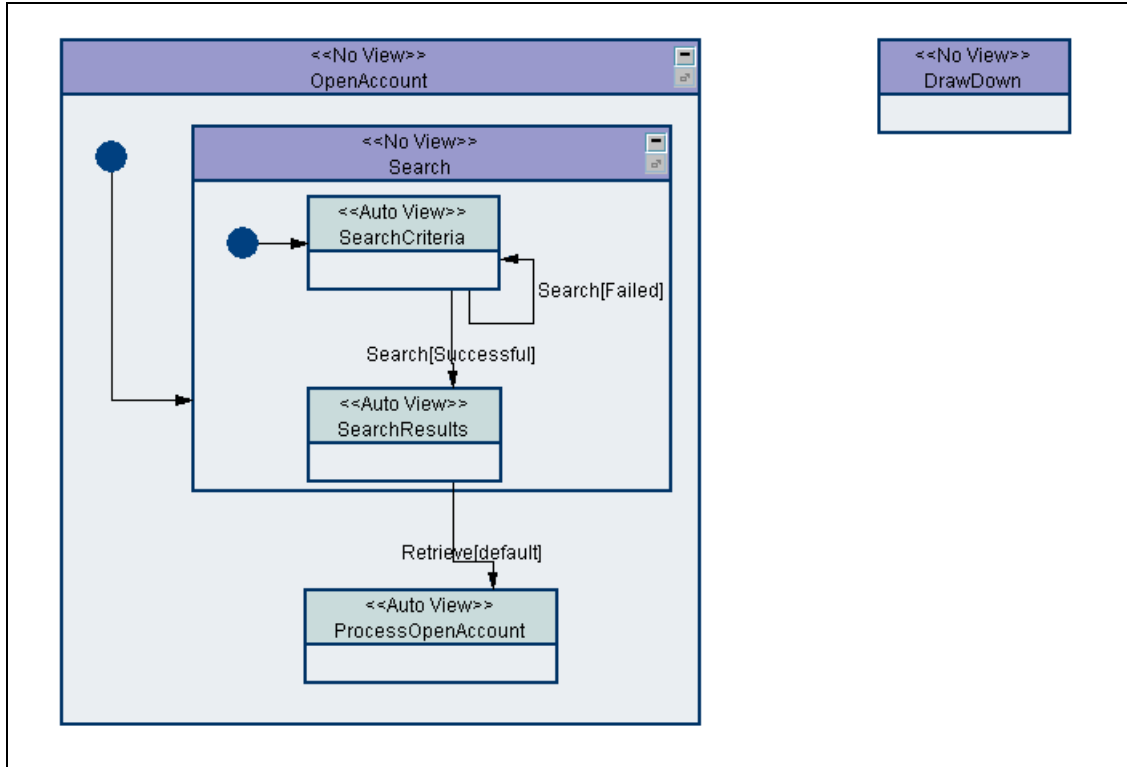
10.2 Copy, Cut & Paste Features

The Orchestrator tool can also copy and cut states to different parts of the statechart and from different statechart files. When copying or cutting a state all its child states and transitions, apart from any transitions either entering or leaving the state, will also be copied or pasted. To copy or cut any state right-click on the state to bring up its popup edit menu. On the popup edit menu will be a menu item to copy or cut the state. Select either one according to what you want to do. Once selected, move the mouse to an area in the statechart (this can be within any state on the statechart) and right-hand click on that area. The popup menu will appear and you can then select the paste menu item to copy or cut the state to that area.

You can copy and cut and paste from one statechart to another. To do this you must first open the statechart that you wish to copy or cut the state from. Select the state and press the copy or cut menu item. Then open the statechart you wish to copy or paste the state into. Once the statechart is opened press the paste menu item where you want to state to be pasted too.

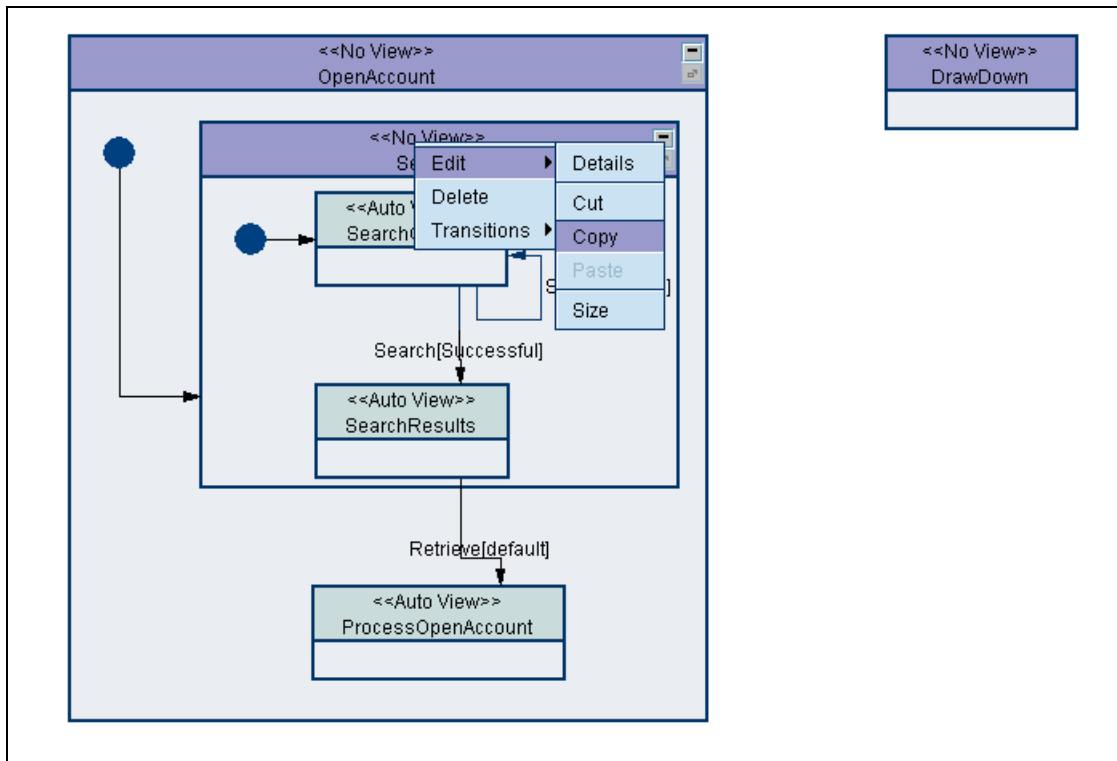
10.2.1 Copy Example

In this example we will copy a search parent state and its child states to another parent state in the statechart. The next figure shows a small statechart for a sample application. We will copy the search state in the OpenAccount parent state to the DrawDown parent state.



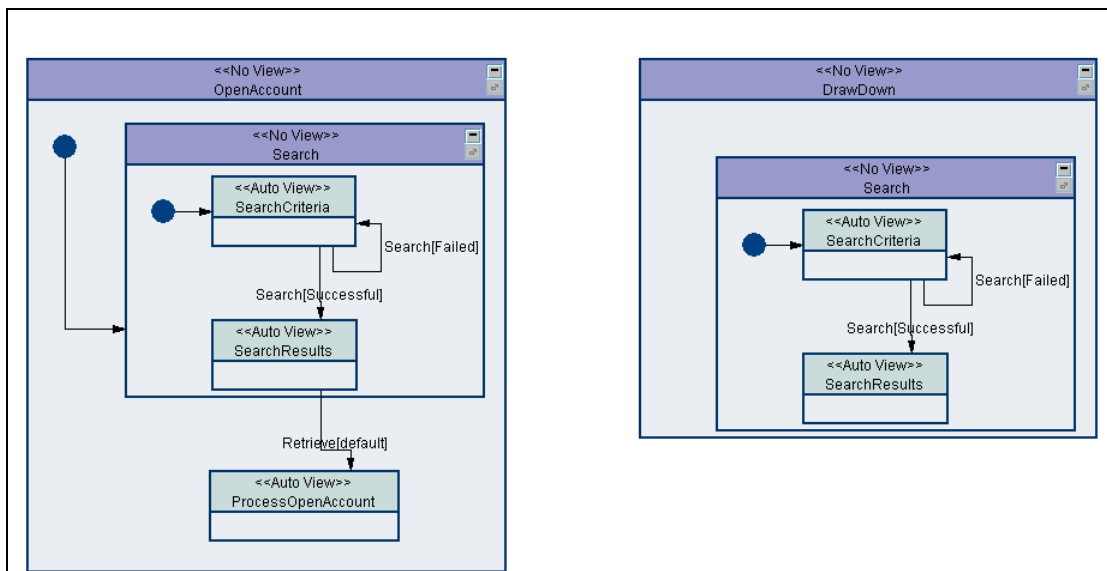
The OpenAccount Search state.

Right-click on the OpenAccount's Search state to bring up the state's popup menu. Select the Edit, Copy menu item as shown in the next figure.



The Search state's Edit, Copy menu item selected.

Move the mouse to an area in the DrawDown state where child states can be added. Right-click the mouse in this area to bring up the state's popup menu. Select the Edit, Paste menu item and the Search state will be copied in this state as shown in the next figure.



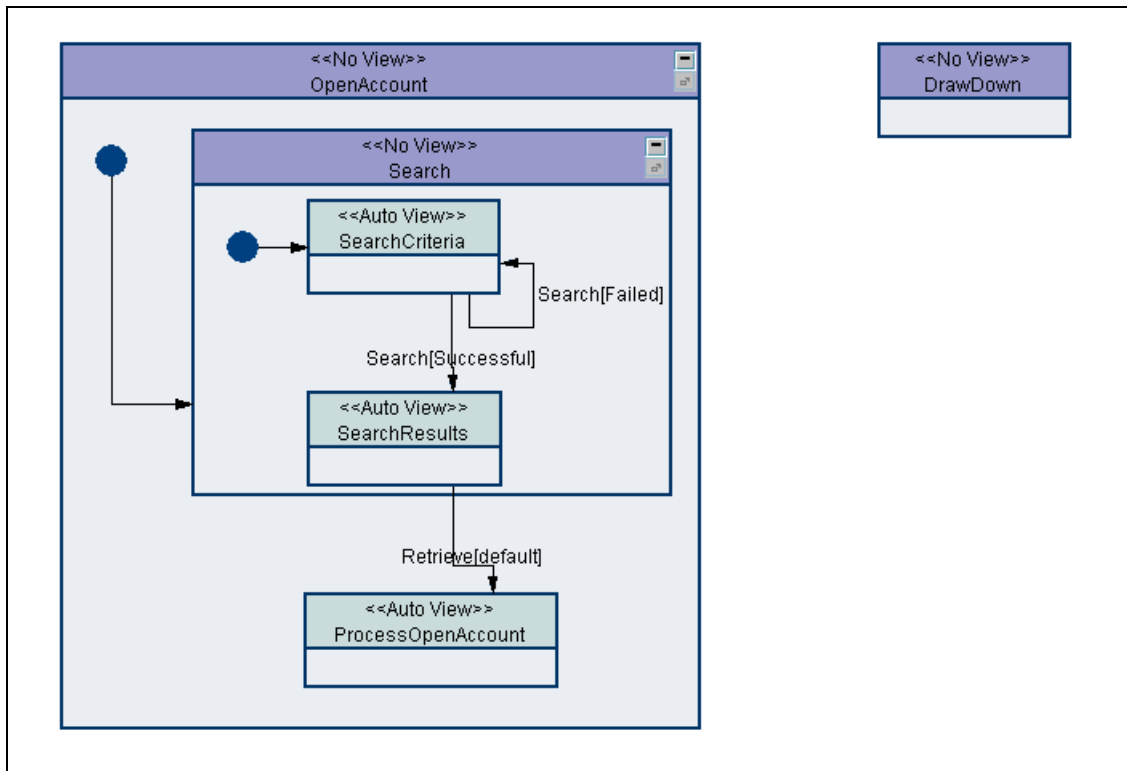
The Search state copied to the DrawDown state.

Please note that the Retrieve[default] transition was not copied as this transition leaves the Search state and

hence is ignored when the copy is done. Also note that the initial state transition coming into the Search state is also ignored by the copy.

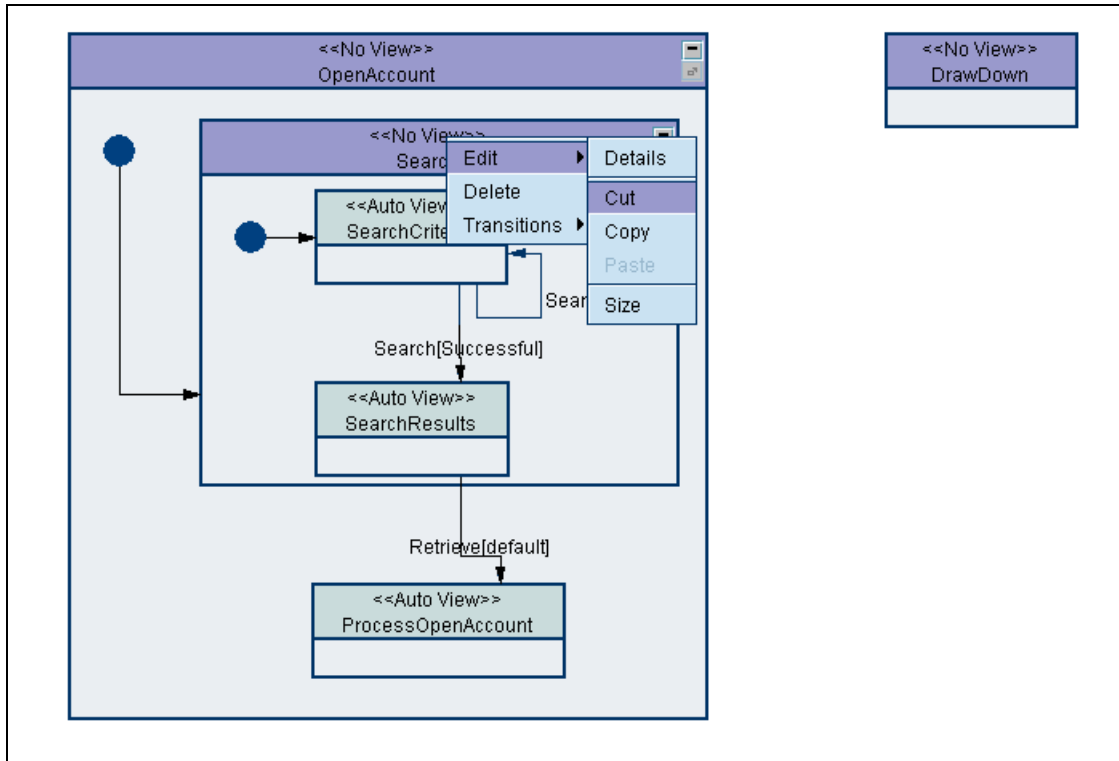
10.2.2 Cut & Paste Example

In this example we will cut and paste a search parent state and its child states to another parent state in the statechart. The next figure shows a small statechart for a sample application. We will cut and paste the search state in the OpenAccount parent state to the DrawDown parent state.

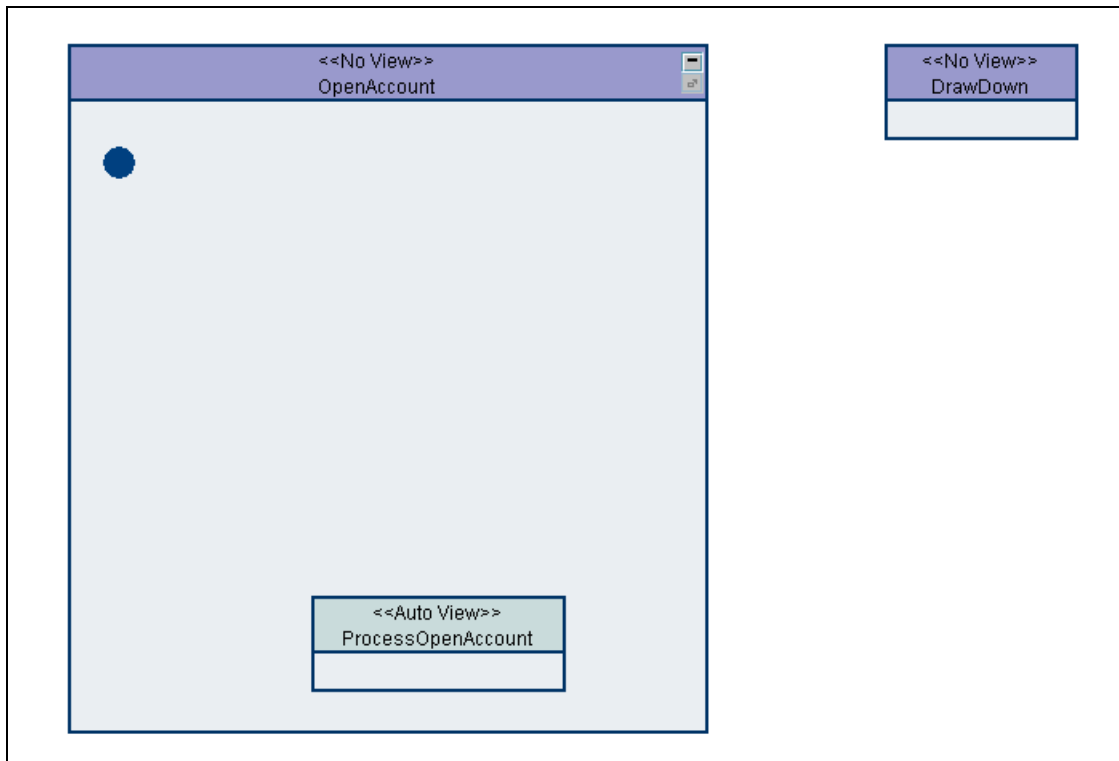


The OpenAccount Search state.

Right-click on the OpenAccount's Search state to bring up the state's popup menu. Select the Edit, Cut menu item as shown in the next figure.

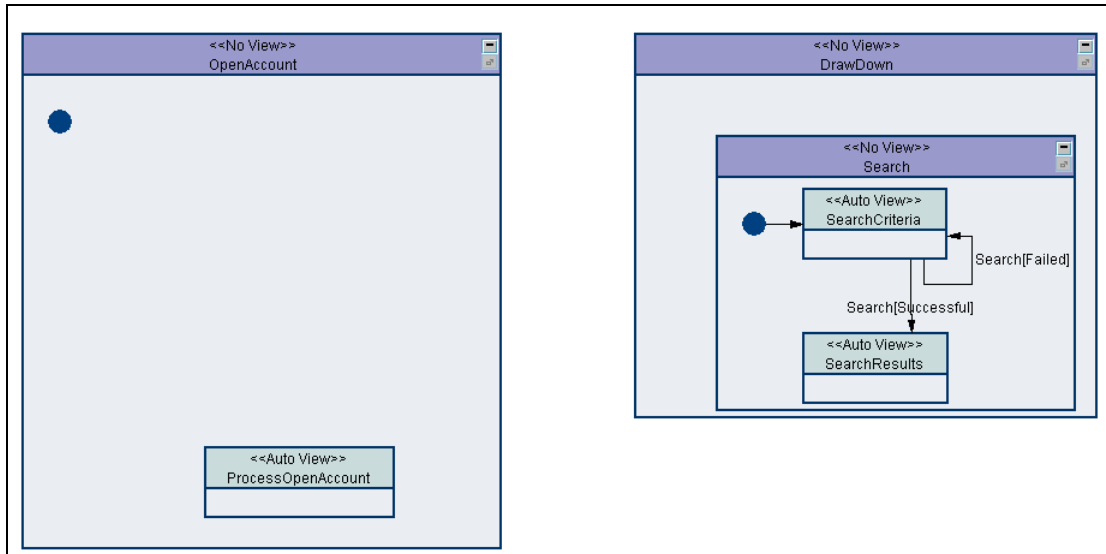


The Search state's Edit, Cut menu item selected.



The Search state cut from the OpenAccount state.

Move the mouse to an area in the DrawDown state where child states can be added. Right-click the mouse in this area to bring up the state's popup menu. Select the Edit, Paste menu item and the Search state will be pasted in this state as shown in the next figure.

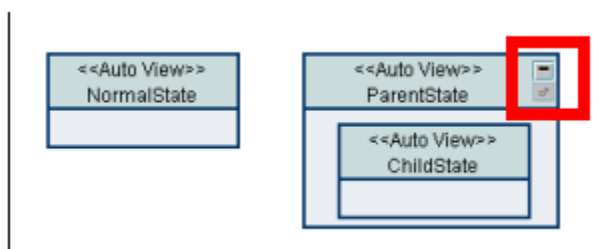


The Search state cut and pasted to the DrawDown state.

Please note that the Retrieve[default] transition was not pasted as this transition leaves the Search state and hence is ignored when the cut and paste is done. Also note that the initial state transition coming into the Search state is also ignored by the cut and paste. These transitions are also removed from the OpenAccount state.

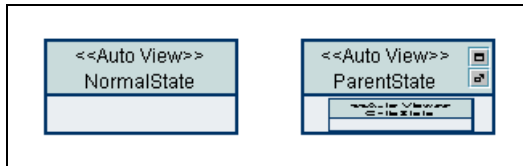
10.3 Parent States As Sub-charts

The orchestrator tool has the ability to minimize and maximize parent states. A parent state is any state that has any child states. When the state becomes a parent state two new icon buttons will appear in the right-hand side of the state's header. The following figure shows a normal state alongside a parent state.



A normal state along side a parent state.

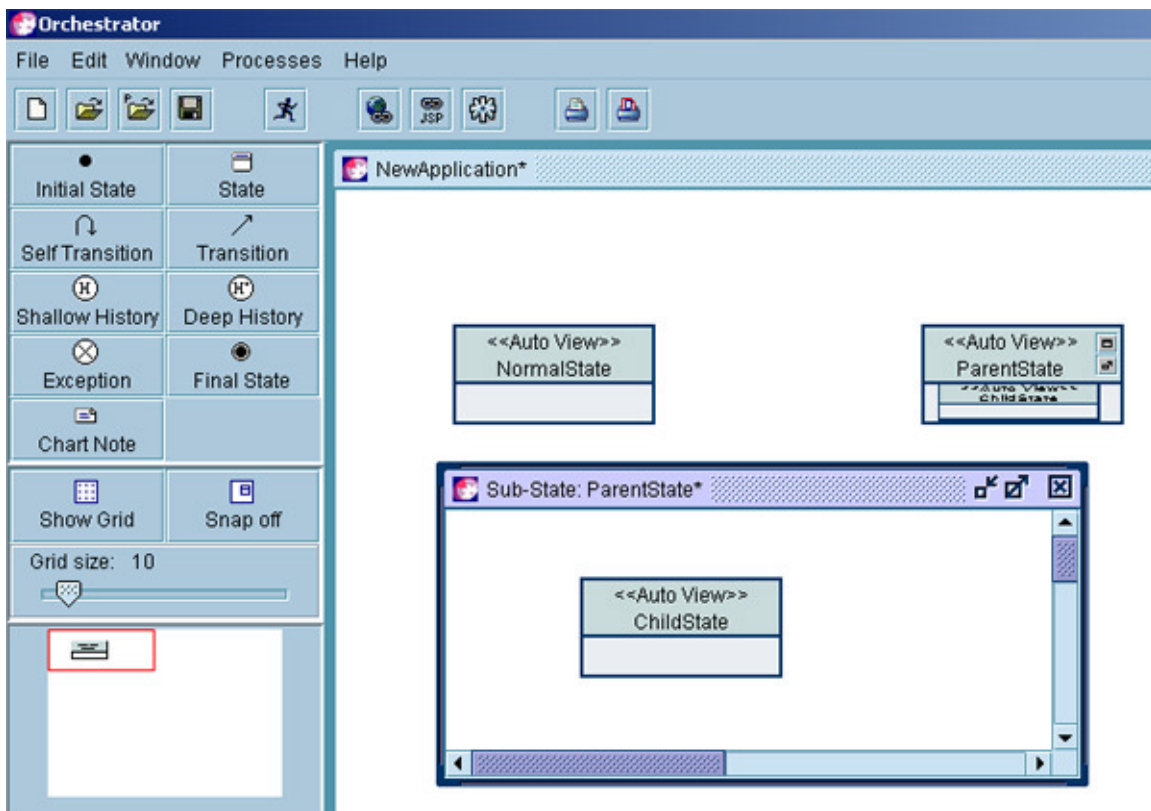
The highlighted area (the red box) in the previous figure shows the additional icon buttons added to the header area of the parent state. The top button, which is initially enabled, allows the user to minimize the parent state. Once minimized the bottom button will become enabled as shown in the following figure.



The parent state minimized.

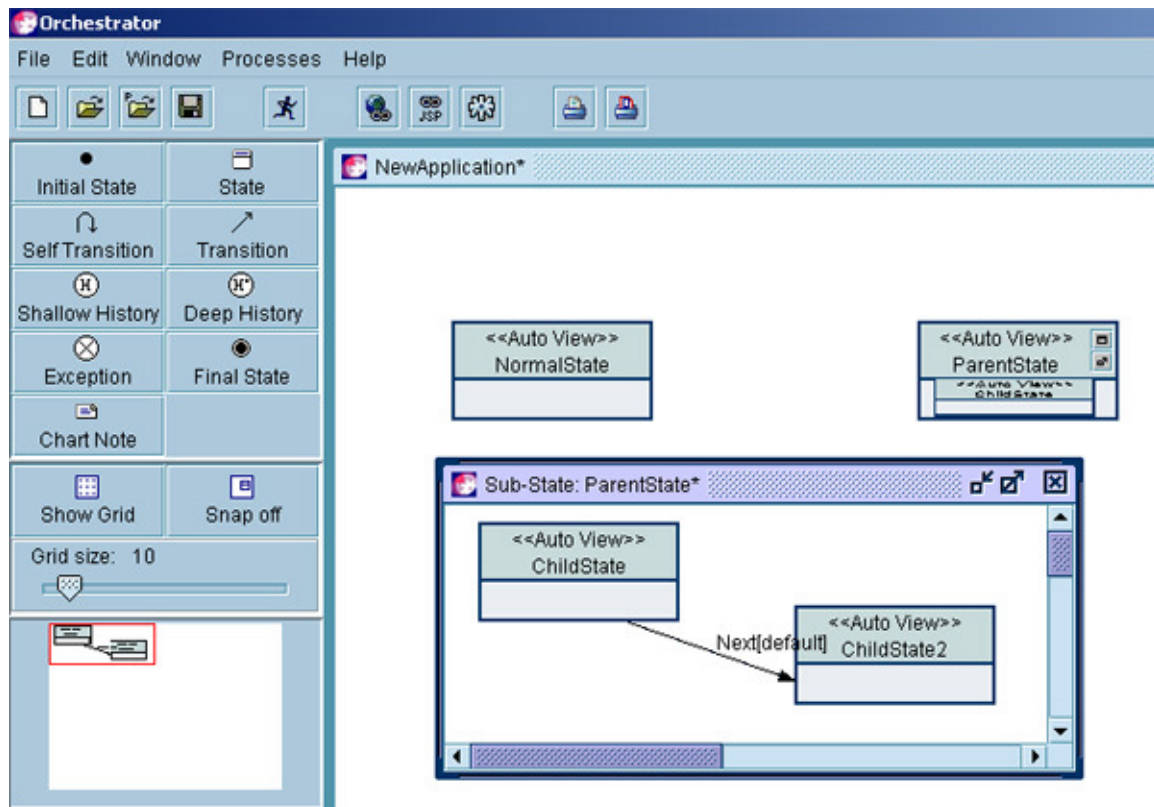
The top button will also change icons and when pressed again will maximize the parent state back to its original size. The top button is therefore interchangeable. It cycles between minimize and maximize icons and will only enable the bottom button when the parent state is minimized.

When the bottom button is enabled the user will be able to open that parent state in a new window. The next figure shows the parent state opened as a sub-chart window after the open button was pressed.



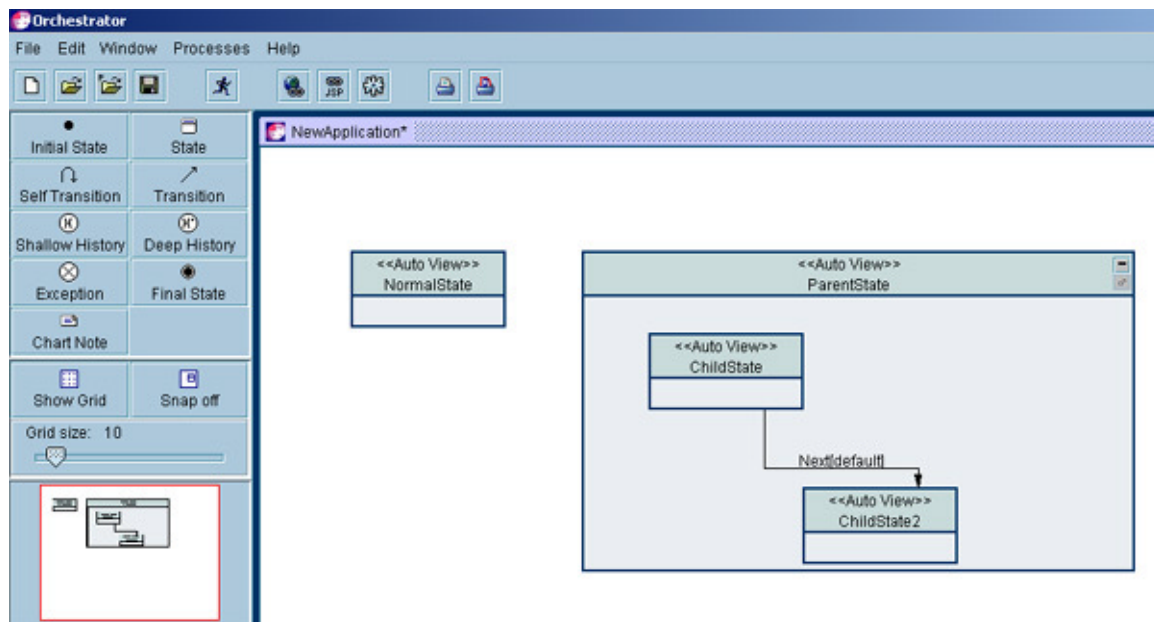
The parent state opened as a sub-chart.

Once opened as a sub-chart the user can draw on it as if it were a normal statechart. In the following figure a new child state has been added to the parent state sub-chart and a new transition.



Using the ParentState sub-chart to add new states and transitions.

When the user closes this window and maximizes the ParentState from the main statechart window the added states and transitions will have been added to the state.

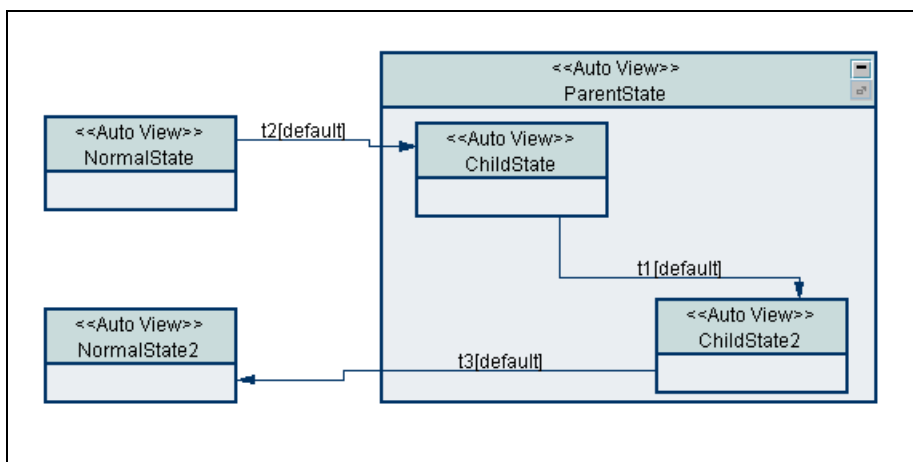


The ParentState maximized.

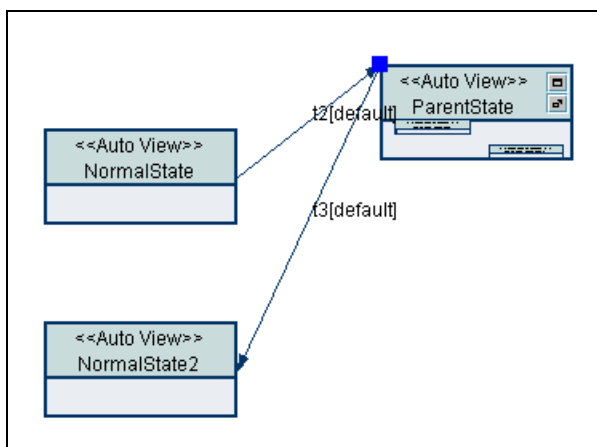
One thing to note is that when a parent state is minimized you will not be able to add states to it unless you maximize it.

10.3.1 Transitions Leaving And Entering Parent States

The Orchestrator tool handles transitions leaving and entering parent states when minimized or opened in a separate sub-chart in a special way. When parent states are minimized any transitions entering or leaving the parent state will be shown by drawing a blue box in the upper left-hand corner of the parent state and the transitions leaving and entering the parent state will be drawn to it. In the two figures that follow a transition enters the parent state to a child state and a transition leaves a child state to a state external to the parent state. The figures indicate what happens to the transitions when the parent state is minimized.

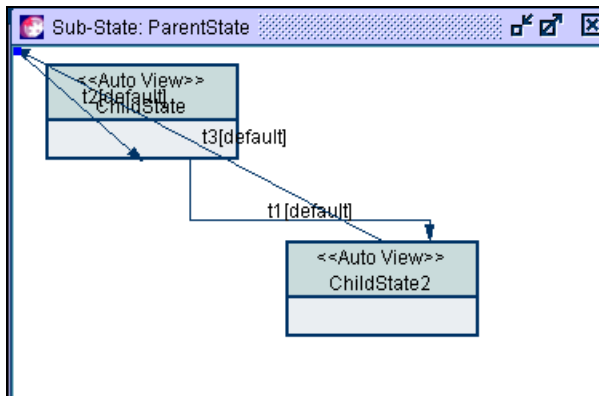


The ParentState maximized.



The ParentState minimized.

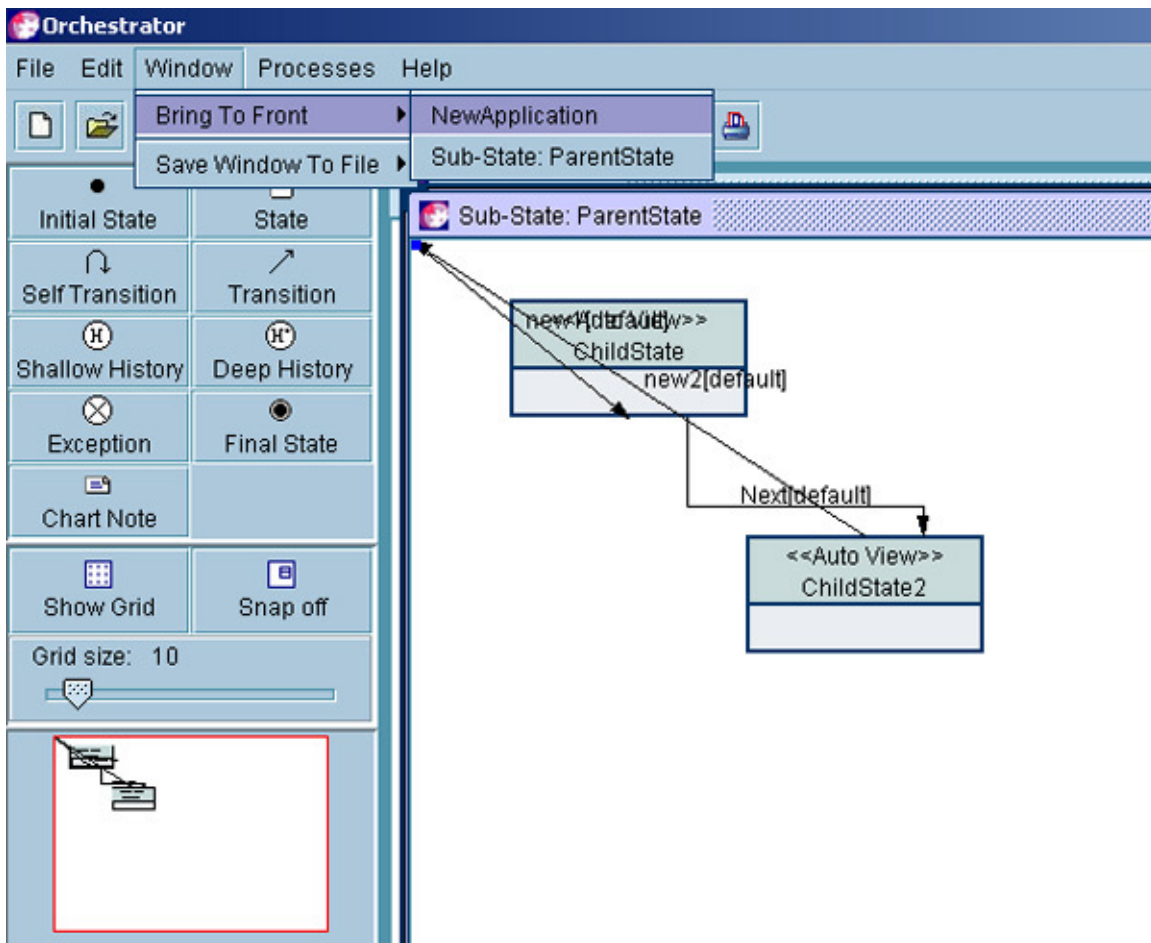
Similarly, when the parent state is opened in a sub-chart, the sub-chart window will also indicate which transitions leave or enter the parent state by drawing a small blue box in the upper left-hand corner of the sub-chart window as shown in the following window.



The ParentState opened in a sub-chart window.

10.3.2 Bringing Sub-charts To The Front

The orchestrator tool can allow any number of parent states to be opened as sub-charts in the desktop area. Where multiple internal windows are open, you can bring any opened window to the front by selecting the **Window -> Bring To Front** from the menu. This provides a menu item list for each window opened in the tool's desktop area. To bring a window to the front, select that window from the menu list. The following figure shows the NewApplication menu item highlighted so that it can be brought to the front of the desktop.



The NewApplication menu item selected so that it can be brought to the front.

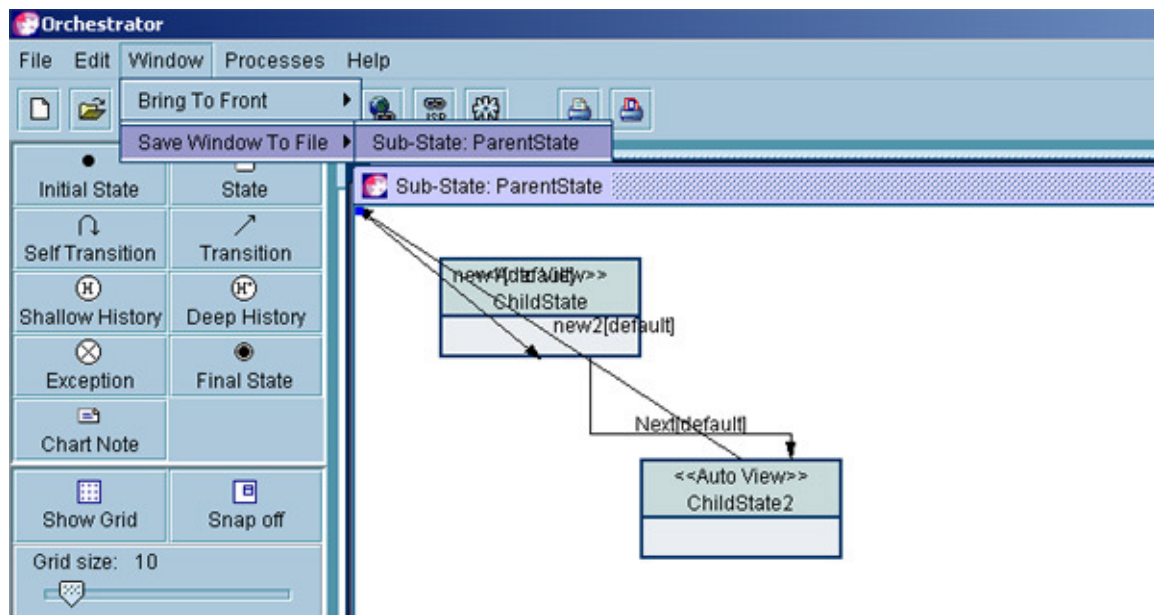
10.3.3 Why Use The Sub-chart Feature?

This feature is extremely useful when the statechart becomes very large. For an application such as a teller or call centre the statechart for the application will be huge, often with over 200 states and transitions. Building the statechart becomes more and more difficult as the application grows in size. This feature is extremely helpful when the statechart grows to such a size. You can more easily edit particular parent states in separate windows helping to reduce the clutter from the other states in the application. This mechanism is also the basis behind the multiple user support feature described in the next section.

10.4 Multiple User Support

The Orchestrator tool can be used for defining very large applications involving multiple users. Parent states opened in sub-chart windows can be saved to separate files linked to the main statechart XML file. Users can open the main statechart for the application and then open parent states as sub-charts. The parent state can then be saved to a separately linked file. The user can then edit this parent state independently of other users and save changes to this file.

The following example indicates how the ParentState can be saved to a separately linked file. First open the ParentState as a sub-chart. Next select the **Window -> Save Window To File** from the menu. This will display a list of menu items of states that can be saved to a separate linked file. Select the “Sub-State: ParentState” menu item. This will save the ParentState’s details to a separate file.



Saving the ParentState to a linked file.

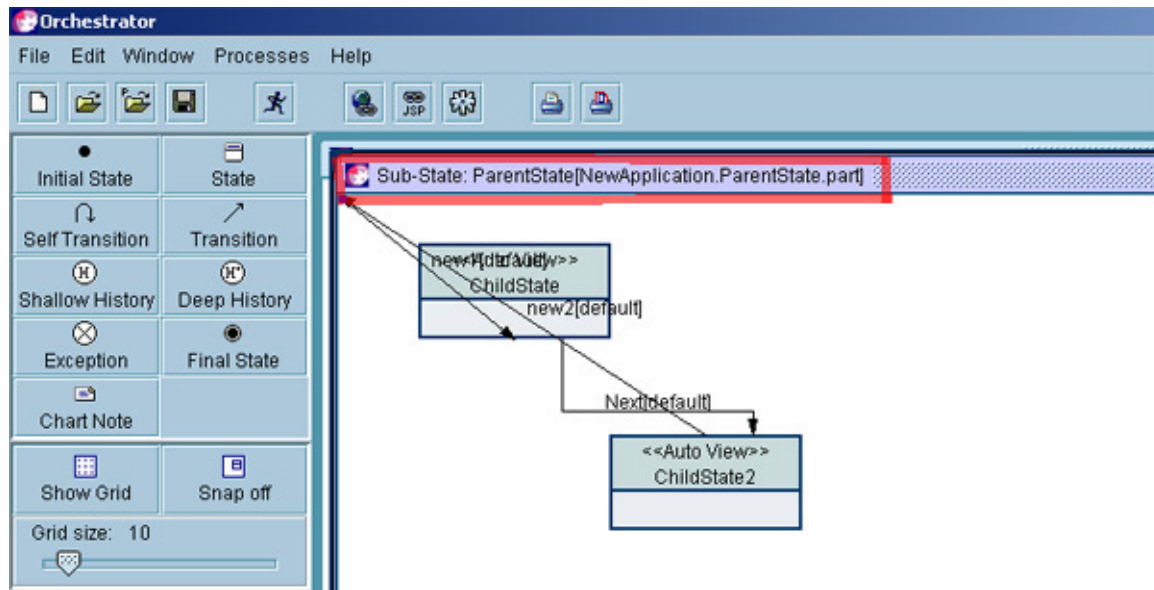
The format of the textual description of the menu items for the Save Window To File menu list is very specific. The format is:

Sub-State: << State Name>>

for example:

“Sub-State: ParentState” in the example.

When a parent state is saved to a file, the state’s window title will be updated to indicate this. The following figure shows the ParentState’s updated window title.




The ParentState window title updated to show the filename of the linked file.

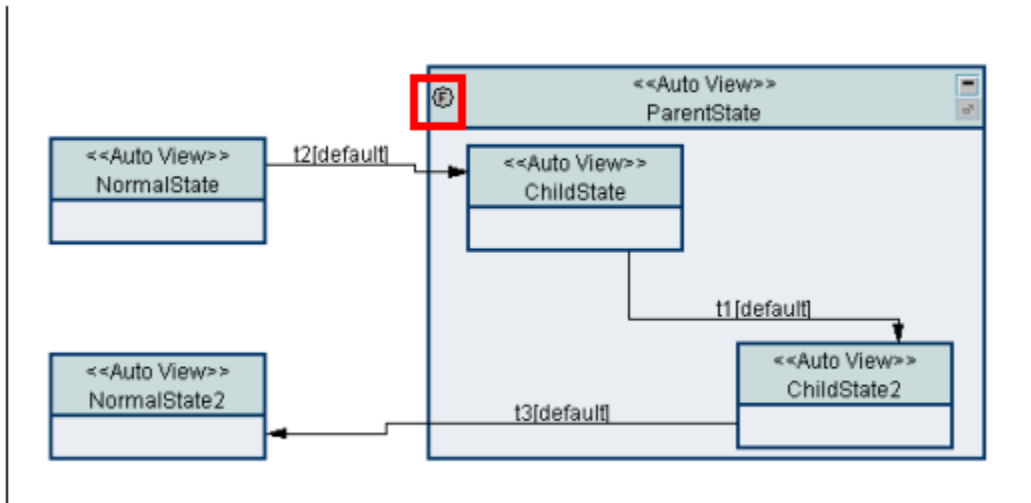
The format of the filename of the ParentState is also very specific. The format is:

<<Application Name>>.<< State Name>>.part

for example:

“NewApplication.ParentState.part”.

If the application is renamed then all its linked state files will also be renamed to ensure that the part files can be easily seen to link to the main application statechart file. When a parent state is saved to a linked file, its header will indicate that its contents are contained in a linked file by displaying a special icon. The  icon is used to indicate states whose contents are contained in a linked file. The following figure indicates this behaviour after the ParentState was saved to a linked file.



The ParentState displaying the  icon.

It must be remembered that to open a linked file you must open the application statechart first. Individual state linked files cannot be opened on their own. They must be opened through their parent application statecharts. Also note that when you edit the parent state in the sub-chart window the application statechart will only be updated when you either save the changes or close the window.

10.4.1 Note on users working on the same files

The Orchestrator tool is very much like any other tool in that it produces a number of flat files. Any user can edit these files and overwrite other users changes. It is therefore important that any files produced by the Orchestrator tool where multiple users are involved are version controlled. The only additional feature provided by the Orchestrator tool is that it will not allow statechart files to be saved if they are marked as read-only. In short it is the job of the users to ensure that their files are under version control and that they do not undo each other's work.

A statechart can be broken up into multiple part files allowing different users to work on separate part files. The main application file, which links all the part files together, should be given special attention. It effects all the part files and should only be maintained and modified by a single person.

11 Introduction To Writing A Swing Application

11.1 Overview

This section describes in brief what is required for a swing application to use the state machine.

The state machine framework supports applications deployed using the swing API as well as those deployed as web applications through the servlet API.

Designing swing applications is virtually identical to designing web applications. The same concern should be given to the behavior and flow control through the application in the state chart, and the same Controller and Process integration classes can be used.

Deploying a swing application using the state machine requires you to use the classes and interfaces in the `com.bankframe.fe.statemachine.ext.connectors.swing` package.

The structure of your application will be:

- An application main class acting as a Window or Applet for the application.
- The main class will contain a ViewContainer. The ViewContainer is a container within which all the application views will be displayed.
- The main class will also contain a StateMachineEventDispatcher, which will listen for StateMachineEvents fired from your views.
- The View classes will all implement IView and StateMachineEventSource. Whenever the user does anything that triggers an event on the state chart, the view must fire a StateMachineEvent.
- The sequence for processing an event is:
 - The view class will fire a StateMachineEvent.
 - The StateMachineEventDispatcher will receive the event and forward it to the RequestManager in the statemachine.
 - The RequestManager will return the new view class.
 - The StateMachineEventDispatcher will register itself as a StateMachineEventListener on the view, so that it will receive the next StateMachineEvent that is fired.
 - The StateMachineEventDispatcher will pass the view to the ViewContainer.
 - The ViewContainer will display the new view.

The important classes and interfaces are:

11.1.1 StateMachineEvent class

The StateMachineEvent class takes on the role of the Request. All user events that are to be processed by the state machine must be fired from the view as StateMachineEvents.

11.1.2 StateMachineEventSource interface

The StateMachineEventSource interface must be implemented by all View classes in addition to the IView interface. The interface contains methods for adding and removing StateMachineEventListeners to the view.

11.1.3 StateMachineEventDispatcher class

The StateMachineEventDispatcher manages the StateMachineEvents fired by the views, passing them on into the state machine. It also gives the resultant view to the ViewContainer for display.

11.1.4 ViewContainer interface

The ViewContainer interface marks the JContainer that will hold and display the views.

11.2 Writing the application main class

The application main class has a few tasks and responsibilities it must complete. Once these three steps are completed correctly it does not matter whether the main class is an applet, frame, or neither.

11.2.1 Create and display a ViewContainer

The application main class must create and display some class that implements ViewContainer. This will be where all the application views are displayed.

11.2.2 Create a StateMachineEventDispatcher

The application main class must create a StateMachineEventDispatcher. The dispatcher requires a ViewContainer and the state machine configuration Properties. The main class can also set a logger, user session manager and application manager if necessary. (In general, these can be loaded automatically based on the values in the configuration Properties.)

11.2.3 Fire the first Event

To start the application, the application main class must fire the first StateMachineEvent into the StateMachineEventDispatcher. Create a StateMachineEvent with **this** as the target and a null event name. The state machine will locate the start state for the application and give the appropriate view to the ViewContainer.

11.3 Writing the ViewContainer class

The ViewContainer class has one very simple responsibility. It must display the views that are given to it through the displayView(IView) method.

The ViewContainer is normally going to be a JPanel or other JContainer. When it receives a view it should check that the view is a JComponent, remove or hide the previous view and display the new one.

Note that the view might not be a JComponent. It could be any class. When writing the ViewContainer you

should be aware of the types of view that will be written for the application, and should have a way of displaying all of them.

For example: another type of view that might be supplied is a `JDialog`, in which case the `ViewContainer` should call the `show()` method to display the dialog. (Note that only modal dialogs should be used.)

11.4 Writing the View classes

As in the servlet environment, the view classes have two very simple and closely-related responsibilities. They must display information suitable to the current state, and they must present controls (e.g. buttons) to the user to allow them fire events.

The view classes must implement two interfaces:

11.4.1 View interface

The `IView` interface includes three methods that you must implement. There are two variants of the `build` method, one inherited from the `com.bankframe.fe.statemachine.base.apps.IView` interface, and one defined in the `com.bankframe.fe.statemachine.ext.apps.IView` interface.

You can define the first of these two methods with the following block of standard code:

```
public void build(RequestContext requestContext, IState currentState) {

    com.bankframe.fe.statemachine.ext.apps.View.
    build(requestContext, currentState, this);

}
```

The second version of the `build` method must be implemented in order to populate the view with the values that should be displayed to the user.

The third method is the `populateFromProperties` method, mentioned in the 'Managing ViewProperties' section below.

11.4.2 StateMachineEventSource interface

The two methods in the `StateMachineEventSource` interface can be implemented using the standard code below:

```
// listenerList is an instance of javax.swing.event.EventListenerList

public void addStateMachineEventListener(StateMachineEventListener
listener) {

    listenerList.add(StateMachineEventListener.class, listener);

}
```

```

public void removeStateMachineEventListener(StateMachineEventListener
listener) {

    listenerList.remove(StateMachineEventListener.class, listener);

}

```

When you need to fire a StateMachineEvent, use the code below as a guide:

```

StateMachineEvent event = new StateMachineEvent(this, eventName);

// set the parameters as required in the event.

StateMachineEventDispatcher.fireStateMachineEvent(event, listenerList);

```

11.5 Putting the View classes in the chart

Adding the swing view classes to your chart is as simple as putting the fully-qualified class name into the View Class text box on the StateDetails dialog.

If your view requires any viewProperties, you can add these in the StateDetails dialog also.

11.6 Managing ViewProperties

ViewProperties are the means through which you can include information in the state chart to be used by the View class. For example: viewProperties contain the jspName for the JSPView and the stylesheetURI for the XSLTAutoView.

You can use viewProperties by implementing the populateFromProperties method on your view to read values from the viewProperties and copy them to attributes that can later be used in the build method.

11.7 The tool view requirements

When a state is created in the Orchestrator tool the “Enter State Details” dialog is displayed as follows:

Enter State Details

Title:

State Type: com.bankframe.fe.statemachine.ext.connectors.servlet.AutoView ▼

State Details:

- (NONE)
- com.bankframe.fe.statemachine.ext.connectors.servlet.AutoView
- com.bankframe.fe.statemachine.ext.connectors.servlet.JSPView
- com.bankframe.fe.statemachine.ext.connectors.servlet.XSLTAutoView
- com.bankframe.fe.statemachine.ext.connectors.swing.AutoView
- com.bankframe.fe.statemachine.ext.connectors.swing.SwingView
- com.bankframe.fe.statemachine.ext.connectors.swing.XSLTSwingView

Processes:

Input Requirements:

name	description	defaultValue	requirement	validationRule
			REQUIRED	

The “Enter State Details” dialog.

Using this dialog the user must specify the type of view this state will represent. By default the tool provides six types, None, AutoView (JSP), XSLTAutoView, JSPView, AutoView (Swing) and SwingView. To specify new swing, JSP or XSLT views, the user must therefore provide a new view type. Writing your own swing view type class can do this. A view type class must implement the `com.bankframe.fe.statemachine.ext.apps.IView` interface. Additionally, a BeanInfo class can also be defined for the new view type. When the user selects the new type from the view type combo box the BeanInfo class for the view type can be loaded by the tool and displayed in the “View Details” area of the “Enter State Details” dialog. The BeanInfo class allows the user to customize the view and is written according to the JavaBeans standard.

An example of creating an IView class and a BeanInfo class for it can be seen in the following code samples. Here we show how the JSPView and the JSPViewBeanInfo classes were written.

11.7.1 The JSPView class

```

public class JSPView extends View {

    protected String jspName;

    protected String requestURL;

    /**
     * The JSP can expect an attribute in the request with the key
     * STATE_ATTRIBUTE_NAME that contains the IState implementor for the
     * current state.
     * <br>
     * The value of STATE_ATTRIBUTE_NAME is "State"
     */
    public static String STATE_ATTRIBUTE_NAME = "State";

    /**
     * The JSP can expect an attribute in the request with the key
     * VIEW_ATTRIBUTE_NAME that contains the instance of JSPView that was
     * used.
     * You might use this to build subclasses of JSPView that perform
    extra
    * processing of the data in the ResponseData, exposing the results of
    that
    * processing through methods on the view.
     * <br>
     * The value of VIEW_ATTRIBUTE_NAME is "View"
     */
    public static String VIEW_ATTRIBUTE_NAME = "View";

    /**
     * The JSP can expect an attribute in the request with the key

```

```

    * INPUTS_ATTRIBUTE_NAME that contains the instance of Inputs that was
    * used.
    * <br>
    * You can use this in the JSP to gain access to the data from the
    * incoming request, the user session, and the response data populated
    * by the controller.
    * <br>
    * The value of INPUTS_ATTRIBUTE_NAME is "Inputs"
    */
    public static String INPUTS_ATTRIBUTE_NAME = "Inputs";

    /**
     * The JSP can expect an attribute in the request with the key
     * REQUEST_CONTEXT_ATTRIBUTE_NAME that contains the current
    RequestContext.
     * <br>
     * The value of REQUEST_CONTEXT_ATTRIBUTE_NAME is "RequestContext"
     */
    public static String REQUEST_CONTEXT_ATTRIBUTE_NAME =
    "RequestContext";

    public static String RESPONSE_DATA_ATTRIBUTE_NAME = "ResponseData";

    /**
     * Constructor for JSPView.
     */
    public JSPView() {
        super();
    }

    /**

```

```

    * @see com.bankframe.fe.statemachine.ext.apps.View#build(IState,
Inputs, RequestContext)

    */

    public void build(

        IState state,

        Inputs inputs,

        RequestContext requestContext) throws StateMachineUserException
    {

        HttpServletRequest request =
((Request)inputs.getRequest()).getRequest();

        Response response = (Response)requestContext.getResponse();

        requestURL = request.getRequestURL().toString();

        request.setAttribute(STATE_ATTRIBUTE_NAME, state);

        request.setAttribute(VIEW_ATTRIBUTE_NAME, this);

        request.setAttribute(INPUTS_ATTRIBUTE_NAME, inputs);

        request.setAttribute(REQUEST_CONTEXT_ATTRIBUTE_NAME,
requestContext);

        request.setAttribute(RESPONSE_DATA_ATTRIBUTE_NAME,
response.getResponseData());

        RequestDispatcher dispatcher =
request.getRequestDispatcher(jspName);

        try {

            dispatcher.include(request, response.getResponse());

        } catch (ServletException ex) {

            throw new StateMachineUserException(ex);

        } catch (IOException ex) {

            throw new StateMachineUserException(ex);

        }

    }

    /**

    * Returns the jspName.

```

```

    * @return String
    */
    public String getJspName() {
        return jspName;
    }

    /**
     * Sets the jspName.
     * @param jspName The jspName to set
     */
    public void setJspName(String jspName) {
        this.jspName = jspName;
    }

    /**
     * Returns the jspName.
     * @return String
     * @deprecated
     */
    public String getJSPName() {
        return jspName;
    }

    /**
     * Sets the jspName.
     * @param jspName The jspName to set
     * @deprecated
     */
    public void setJSPName(String jspName) {
        this.jspName = jspName;
    }

```



```

    }

    /**
     * Returns the requestURL.
     * @return String
     */
    public String getRequestURL() {
        return requestURL;
    }

    /**
     * @see
     com.bankframe.fe.statemachine.ext.apps.IView#populateFromProperties(Properties)
     */
    public void populateFromProperties(Properties viewProperties) {
        if (viewProperties != null) {
            if (viewProperties.getProperty("jspName") != null) {
                setJspName(viewProperties.getProperty("jspName"));
            }
        }
    }
}

```

11.7.2 The JSPViewBeanInfo Class

```

public class JSPViewBeanInfo extends SimpleBeanInfo {

    protected PropertyDescriptor[] propertyDescriptors;

    protected BeanDescriptor beanDescriptor;

```

```

/**
 * Constructor for JSPViewBeanInfo.
 */
public JSPViewBeanInfo() throws IntrospectionException {
    super();

    PropertyDescriptor jspNameDescriptor = new
PropertyDescriptor("jspName", JSPView.class, "getJspName", "setJspName");

    PropertyDescriptor requestURLDescriptor = new
PropertyDescriptor("requestURL", JSPView.class, "getRequestURL", null);

    propertyDescriptors = new
PropertyDescriptor[]{jspNameDescriptor, requestURLDescriptor};

    beanDescriptor = new BeanDescriptor(JSPView.class,
GenericCustomizer.class);
}

/**
 * Returns the propertyDescriptors.
 * @return PropertyDescriptor[]
 */
public PropertyDescriptor[] getPropertyDescriptors() {
    return propertyDescriptors;
}

/**
 * Returns the beanDescriptor.
 * @return BeanDescriptor
 */
public BeanDescriptor getBeanDescriptor() {
    return beanDescriptor;
}
}

```

The JSPViewBeanInfo class allows the user to enter the JSP filename for that particular state. When the JSPView is selected in the enter state details dialog the following customizer is loaded in the “View Details” area as a result of providing the JSPViewBeanInfo class:

Enter State Details

Title:

State Type:

State Details:

jspName:

Processes:

Input Requirements:

name	description	defaultValue	requirement	validationRule
			REQUIRED	

Edit note OK Cancel

The JSPViewBeanInfo class loaded.

11.8 The swing application requirements

The statemachine comes complete with two different connector packages, designed to allow deployment of applications within swing or servlet environments.

This section describes how to write a swing application based on the statemachine.

Before you set up your application you should have already created your statechart for the application and identified your views and controllers. If your application requires any special controller classes, please read the section on creating controller classes. On start-up there are a few steps your application needs to take in order to use the statemachine. First, it needs to designate an object to contain and display the views as they

are produced. This object will probably be an instance of JPanel or another JContainer, and must implement the ViewContainer interface.

11.8.1 The ViewController interface

The `com.bankframe.fe.statemachine.ext.connectors.swing.ViewContainer` interface has one method that must be implemented and that is the `displayView(IView view)` method. When the state machine calls this method, the container class that implements the method must display the specified view to the user. Please see the API JavaDocs for further details on using this interface.

11.8.2 Setting the application properties

The swing application then needs to set up the properties required by the `UserSessionManagerFactory`, `ApplicationManagerFactory`, `ApplicationManager` and `RequestContext`. The default values for these classes when using running a swing application are:

```
com.bankframe.fe.statemachine.base.UserSessionManager=
com.bankframe.fe.statemachine.ext.sessionmanagers.
inmemory.UserSessionManager

com.bankframe.fe.statemachine.base.ApplicationManager=
com.bankframe.fe.statemachine.ext.apploders.sax.ApplicationManager
```

11.8.3 The state machine events

Next, your swing application must create an instance of the `StateMachineEvent Dispatcher` class passing in the `ViewContainer` and `Properties` as parameters to the constructor of the class. Finally, create a `StateMachineEvent` and pass it into the `StateMachineEventDispatcher`. The `StateMachineEventDispatcher` and `StateMachineEvent` classes can be found in the `com.bankframe.fe.statemachine.ext.connectors.swing` package. This initial event can have the `ViewContainer` as its target and a null event name. This will cause the statemachine to load the application, locate the start state, build the appropriate view, and pass it back into the `ViewContainer` via the `displayView` method.

To complete the circle and ensure all subsequent events are properly handled, there are two remaining details. The views for the application must be sources of `StateMachineEvents`, implementing the `StateMachineEventSource` interface. The view must be able to recognize those user actions that are events described on the statechart and fire `StateMachineEvents` appropriately. The `ViewContainer` must ensure that all views it displays have the `StateMachineEventDispatcher` registered as a `StateMachineEventListener` with the view. This ensures that when the view fires a `StateMachineEvent`, the dispatcher receives it, passes it to the `RequestManager`, and passes the result view back to the `ViewContainer`.

11.9 A swing application example

The Orchestrator tool provides a preview frame for loading the currently opened statechart and stepping through the statechart using a swing AutoView class. This is a simple example of a swing application using the state machine. We will use it here to provide a simple example of how a swing application can be created using the state machine. The following code is the PreviewFrame used by the Orchestrator tool:

```
/**
 * The PreviewFrame class.
 *
 * This class provides a swing frame for running a preview of a drawn
 * statechart.
 *
 * @author Brian O'Byrne
 */
public class PreviewFrame extends JFrame implements ViewContainer {

    private StateMachineEventDispatcher eventDispatcher;

    private JScrollPane scrollPane;

    private JPanel viewportComponent;

    /**
     * The PreviewFrame constructor.
     *
     * @param appDoc Document is the XML document representation of the
     * statechart to be previewed.
     */
    public PreviewFrame(Document appDoc) {
        this(appDoc, "State Chart Editor Preview");
    }

    /**
     * The PreviewFrame constructor.
     *
     * @param app Application is the statechart application to be
     * previewed.
     */
    public PreviewFrame(Application app) {
```

```

        this(app, "State Chart Editor Preview");
    }

    /**
     * The PreviewFrame constructor.
     * @param appDoc Document the statechart xml.
     * @param title String
     */
    public PreviewFrame(Document appDoc, String title) {
        super(title);

        initComponents();

        Properties applicationProperties = new
        Properties(System.getProperties());

        applicationProperties.setProperty("com.bankframe.fe.statemachine.base.Appl
        icationManager", "com.eontec.statechart.preview.ApplicationManager");

        applicationProperties.setProperty("com.bankframe.fe.statemachine.base.User
        SessionManager",
        "com.bankframe.fe.statemachine.ext.sessionmanagers.inmemory.UserSessionMan
        ager");

        applicationProperties.setProperty(RequestManager.VIEW_OVERRIDE_KEY,
        "com.bankframe.fe.statemachine.ext.connectors.swing.AutoView");

        try {
            eventDispatcher = new StateMachineEventDispatcher(this,
            applicationProperties);

            ApplicationManager appManager =
            (ApplicationManager)eventDispatcher.getApplicationManager();

            appManager.loadApplication(appDoc);

            appManager.getDefaultApplication();

            eventDispatcher.handleStateMachineEvent(new
            StateMachineEvent(this, null));

        } catch (StateMachineException ex) {

```

```

        ex.printStackTrace();
    }
}

/**
 * The PreviewFrame constructor.
 * @param app Application the statechart.
 * @param title String
 */
public PreviewFrame(Application app, String title) {
    super(title);

    initComponents();

    Properties applicationProperties = new
Properties(System.getProperties());

    applicationProperties.setProperty("com.bankframe.fe.statemachine.base.Appl
icationManager", "com.eontec.statechart.preview.ApplicationManager");

    applicationProperties.setProperty("com.bankframe.fe.statemachine.base.User
SessionManager",
    "com.bankframe.fe.statemachine.ext.sessionmanagers.inmemory.UserSessionMan
ager");

    applicationProperties.setProperty(RequestManager.VIEW_OVERRIDE_KEY,
    "com.bankframe.fe.statemachine.ext.connectors.swing.AutoView");

    try {
        eventDispatcher = new StateMachineEventDispatcher(this,
applicationProperties);

        ApplicationManager appManager =
(ApplicationManager) eventDispatcher.getApplicationManager();

        appManager.loadApplication(app);

        appManager.getDefaultApplication();

        eventDispatcher.handleStateMachineEvent(new
StateMachineEvent(this, null));
    }
}

```

```

        } catch (StateMachineException ex) {
            ex.printStackTrace();
        }
    }

/**
 * This method initializes the frame.
 */
private void initComponents() {
    addWindowListener(new java.awt.event.WindowAdapter() {
        public void windowClosing(java.awt.event.WindowEvent evt) {
            exitForm(evt);
        }
    });

    this.getContentPane().setLayout(new BorderLayout());
    scrollPane = new JScrollPane();

    JButton closeButton = new JButton();
    closeButton.setActionCommand("CLOSE_BUTTON_CMD");
    closeButton.setText("Close Preview");
    closeButton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent actionEvent) {
            exitForm(actionEvent);
        }
    });

    viewportComponent = new JPanel();
    viewportComponent.setLayout(new BorderLayout());
    JPanel viewportComponentFiller = new JPanel();
    viewportComponent.add(viewportComponentFiller,
        BorderLayout.CENTER, 0);

    viewportComponent.add(new JPanel(), BorderLayout.NORTH, 1);

```



```

        scrollPane.setViewportView(viewportComponent);

        scrollPane.setBackground(null);

        this.getContentPane().add(scrollPane, BorderLayout.CENTER);

        this.getContentPane().add(closeButton, BorderLayout.SOUTH);

        this.setIconImage(ImageLoader.getImageIcon("STATE_MACHINE_ICON").getImage());

        this.setSize(new Dimension(550,600));
    }

    /**
     * This method hides the preview frame.
     */
    private void exitForm(java.awt.event.WindowEvent evt) {
        this.hide();
    }

    /**
     * This method hides the preview frame.
     */
    private void exitForm(ActionEvent evt) {
        this.hide();
    }

    /**
     * This method will display the specified view in the preview frame.
     * @see
     * com.bankframe.fe.statemachine.ext.connectors.swing.ViewContainer#displayView(IView)
     */
    public void displayView(IView view) {

```

```

((StateMachineEventSource)view).addStateMachineEventListener(eventDispatcher);

viewportComponent.remove(1);

viewportComponent.add((Component)view, BorderLayout.NORTH, 1);

validate();

repaint();

}

/**
 * This method adds a StateMachineProcessingListener to the
 * statemachine event dispatcher.
 *
 * @param listener StateMachineProcessingListener
 */
public void
addStateMachineProcessingListener(StateMachineProcessingListener listener)
{

    this.eventDispatcher.addStateMachineProcessingListener(listener);

}

/**
 * This method removes a StateMachineProcessingListener to the
 * statemachine event dispatcher.
 *
 * @param listener StateMachineProcessingListener
 */
public void
removeStateMachineProcessingListener(StateMachineProcessingListener
listener) {

    this.eventDispatcher.removeStateMachineProcessingListener(listener);

}

```

```

/**
 * This method adds a collection of StateMachineProcessingListeners to
the statemachine event dispatcher.
 * @param listeners Collection
 */
public void addStateMachineProcessingListener(Collection listeners) {

    this.eventDispatcher.addStateMachineProcessingListener(listeners);

}

/**
 * This method removes a collection of StateMachineProcessingListener
to the statemachine event dispatcher.
 * @param listeners Collection
 */
public void removeStateMachineProcessingListener(Collection listeners)
{

    this.eventDispatcher.removeStateMachineProcessingListener(listeners);

}

}

```

The important things to look at in the code example are the constructors for the class. They create the application properties for the statemachine and set them specifically for this application. The preview frame provides its own ApplicationManager; this class is used to load the specified statechart application or xml document in this instance.

```

applicationProperties.setProperty("com.bankframe.fe.statemachine.base.Appl
icationManager", "com.eontec.statechart.preview.ApplicationManager");

applicationProperties.setProperty("com.bankframe.fe.statemachine.base.User
SessionManager",
"com.bankframe.fe.statemachine.ext.sessionmanagers.inmemory.UserSessionMan
ager");

```

```
applicationProperties.setProperty(RequestManager.VIEW_OVERRIDE_KEY,
    "com.bankframe.fe.statemachine.ext.connectors.swing.AutoView");
```

It also sets a view override, which informs the statemachine that all views specified in the statechart must be ignored and only the swing AutoView class must be used as views for the preview frame. This will not be required in your swing applications, as you will want your application to load the views that you specify.

Next the constructor creates a StateMachineEventDispatcher using the previously highlighted properties.

```
eventDispatcher = new StateMachineEventDispatcher(this,
    applicationProperties);
```

The PreviewFrame class implements the ViewController interface and can therefore be used in the constructor of the StateMachineEventDispatcher class. The displayView method, which is required to be implemented by the PreviewFrame, is highlighted next. It is a simple method whose responsibility is only to display the next view. It also registers the event dispatcher with the view.

```
public void displayView(IView view) {

    ((StateMachineEventSource)view).addStateMachineEventListener(eventDispatch
er);

    viewportComponent.remove(1);

    viewportComponent.add((Component)view, BorderLayout.NORTH, 1);

    validate();

    repaint();

}
```

The AutoView instances, which are IView interfaces, will then fire events to the statemachine using the registered event dispatcher.

12 Validating Input Requirements

An additional feature that has now been added to the orchestrator and state machine is the ability for the orchestrator tool to be used to define validation rules for various input requirements and for the state machine to execute these rules before any event is handled. This is a very useful feature for web-based applications where no validation can be done on the actual JSP (for example no dynamic scripting is allowed on the page) and the form submitted to the state machine must be validated before any processing is done.

12.1 Define the Validation Rules

A new checkbox and table column has been added to the event transition wizard. The screen shot below shows the “Validate event’s input requirements?” checkbox and the input requirement table’s new validationRule column.

Transition Wizard

Enter Transition's Event Details

*Event: E1

Controller: com.eontec.statemachine.helpers.ChannelClientController

Controller Properties:

Char: mitandcheck.SubmitAndCheck

Processes:

TESTDP

☒ Validate event's input requirements?

InputRequirements:

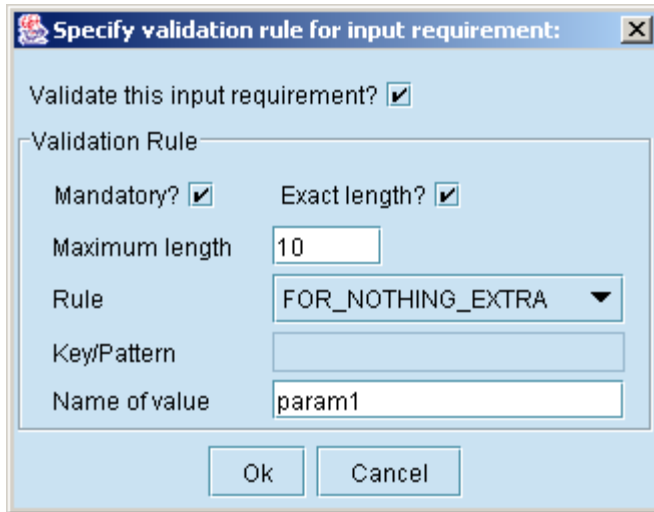
name	description	defaultValue	requirement	validationRule
DATA PACKET NAME	Auto InputDefinition b...	TEST	OPTIONAL	
PARAM1	Auto InputDefinition b...		REQUIRED	true,true,true;10;FOR_...
PARAM2	Auto InputDefinition b...		REQUIRED	true,true,true;10;FOR_...
REQUEST_ID	Auto InputDefinition b...	XXXXX	OPTIONAL	

(* Indicates mandatory fields)

Edit event note Edit transition note Next Cancel

Event's input requirement table with highlighted validationRule column.

The “Validate event’s input requirements?” checkbox must be selected if the event’s input requirements are to be validated. To specify a validation rule for an input requirement double-click in the validationRule cell for that input requirement. This will open the “Specify validation rule for input requirement.” dialog box, which allows the user to specify the rule for that input requirement. The following screen shot shows the dialog box that is displayed.



Specify validation rule for input requirement:

Validate this input requirement? ☒

Validation Rule

Mandatory? ☒ Exact length? ☒

Maximum length

Rule

Key/Pattern

Name of value

Ok Cancel

The validationRule dialog box.

12.2 How the state machine handles the validation check

Once the orchestrator has been used to define the validation rules, the state machine can be used to run the actual application. When an event is submitted to the state machine its first task is to determine if validation of the event's inputs is required before the event is processed and its transition followed. If validation is required then the state machine reads the rules for each input requirement and then validates each input based on the specified rule. Each input is tested in turn and a record is built up of all the inputs that fail validation. If no input fails validation then the state machine proceeds as normal. However if any of the input's fail validation then the record of failed inputs and their validation exceptions are added to the request as a collection of "FAILED_VALIDATION_ERRORS". The state machine then returns the user to the last displayed state. The view for that state can then display the failed validation rules to the user. The following screen shots show the orchestrator's preview frame running a test application and failing input validations.

The image shows a software window titled "State Chart Editor Preview". It contains two main sections for configuring state transitions. The first section is for "State: A", showing an event "E1 (from state A)" with four parameters: "PARAM2 (REQUIRED)" (empty), "DATA PACKET NAME" (filled with "TEST"), "REQUEST_ID" (filled with "XXXXX"), and "PARAM1 (REQUIRED)" (filled with "232"). Below these is a "Transitions from this event:" section with a text box containing "default to state B (DEFAULT)" and a button labeled "E1". The second section is for "Event: Home (from state TEST)", showing a "Transitions from this event:" section with a text box containing "default to state TEST (DEFAULT)" and a button labeled "Home". At the bottom of the window is a "Close Preview" button.

State: A	
Event: E1 (from state A)	
PARAM2 (REQUIRED)	
DATA PACKET NAME	TEST
REQUEST_ID	XXXXX
PARAM1 (REQUIRED)	232
Transitions from this event:	
default to state B (DEFAULT)	
E1	

Event: Home (from state TEST)	
Transitions from this event:	
default to state TEST (DEFAULT)	
Home	

Close Preview

The orchestrator preview frame for testing an application.

State Chart Editor Preview

State: A

Failed validation exceptions

Validation Exception, the "param2" is mandatory.
Validation Exception, the "param1" is required to be of exact length 10.

Event: E1 (from state A)

PARAM2 (REQUIRED)

DATA PACKET NAME

REQUEST_ID

PARAM1 (REQUIRED)

Transitions from this event:

default to state B (DEFAULT)

E1

Event: Home (from state TEST)

Transitions from this event:

default to state TEST (DEFAULT)

Home


Close Preview

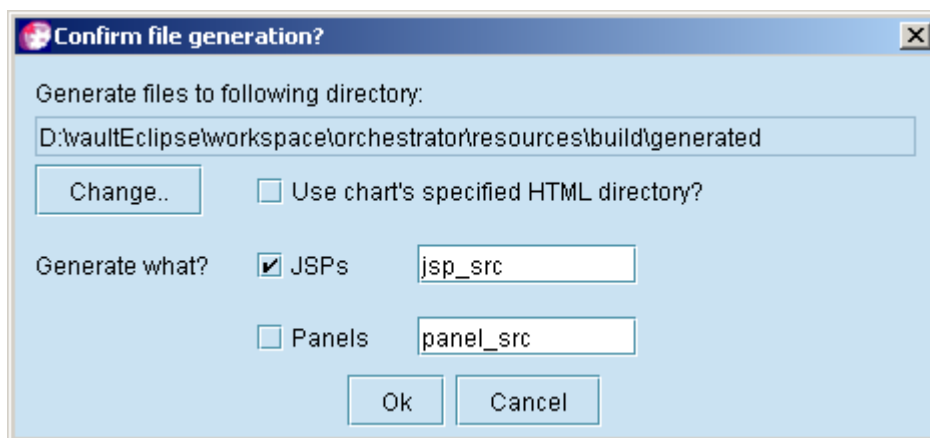
The "E1" event fired and the resultant validation failure results displayed.

13 Generating JSP and Swing Panels

The state machine has a concept of autoviews. If no view exists for a particular state then the state machine can supply an autoview for that state during runtime. This autoview was only created on the fly during runtime and was not a permanent file that could be used. A new feature has now being added to the orchestrator that allows the user to generate out the actually jsp or swing panels that the autoview would create. The files are very useful for providing initial starting points for view states for developing the application. The user can draw their statechart and generate a starting set of JSPs and/or swing panels from which the initial application can be tested and developed. The user can take these files and change or edit them as required. The JSPs and swing panels are created using a set of style sheets. The user has access to these and can modify them to change the look and feel of the files that get generated. The jsp style sheet is the <<orchestrator-install-dir>>\resources\jspTemplate.xml file, while the swing panel style sheet is the <<orchestrator-install-dir>>\resources\panelTemplate.xml. These files can be edited directly by the user to change what is produced when the orchestrator generator is used.

13.1 Running the Generator

Pressing the generator button  on the orchestrator tool bar will access the orchestrator view generator feature. When pressed the “Confirm file generation?” dialog is displayed as shown in the following screen shot.



The confirm file generation dialog.

The confirm file generation dialog will allow you to generate all the JSPs and/or Panels for the currently opened statechart. It generates the files to a special directory. This can be changed to point to any directory you wish or it can be pointed to the chart's specified HTML directory. When the “Ok” button is pressed the files are generated. If any file already exists the user will be asked to confirm if they wish to overwrite that file and all files in the folder.

When generating JSPs the filename for each state will be the jspName specified in the JSPView state. If it is

an autoview the JSP filename will be the state name post fixed with a “.jsp” extension. When generating swing panels, the filename will be based on the fully qualified classname specified for each SwingView's state panelName. If it is an autoview, then the panel name will be based on the state name and the package name will be defaulted to “temp”.

14 MCA Services Timing Points

Timing points have been added to various points in the state machine for performance testing purposes. To record the overall time for a statemachine request timing points have been added as follows:

- In the `com.bankframe.fe.statemachine.ext.connectors.servlet.EntryServlet` for JSP/HTML applications the `public void delegateToRequestManager(HttpServletRequest req, HttpServletResponse res) throws ServletException, IOException;` method records the overall time to handle a html request to the statemachine.
- In the `com.bankframe.fe.statemachine.ext.connectors.swing.StateMachineEventDispatcher` for swing applications the `public void handleStateMachineEvent(StateMachineEvent evt)` method records the overall time to handle a swing request to the statemachine.

Additional timing points have also been added to the

`com.bankframe.fe.statemachine.ext.apps.Controller` class and to two public methods:

- `public com.bankframe.fe.statemachine.base.apploders.IStateTransition getResult(RequestContext requestContext, com.bankframe.fe.statemachine.base.apploders.IEvent event) throws StateMachineUserException;`
- `public void doSideEffects(RequestContext requestContext, com.bankframe.fe.statemachine.base.apploders.IStateTransition transition) throws StateMachineUserException`

These timings determine how long it takes for the statemachine to get the correct transition to follow and to do the side effects for that chosen transition.

If any of these methods are overwritten, the timing code should also be put into the overwritten methods.

Timings have been added as follows:

```
TimingPoint tp = new TimingPoint("NAMEOFCOMPONENT",
BankFrameLogConstants.STATEMACHINE_SUBSYSTEM, "", this);
```

....code here.....

```
tp.exit(this);
```

The descriptions used in the state machine classes were:

```
"StateMachine EntryServlet request round trip time"
```

```
"StateMachineEventDispatcher request round trip time"  
"Controller getResult round trip time"  
"Controller doSideEffects round trip time"
```

The MCA 1.2 and later versions already have timing code for measuring the time for executing actual MCA requests. This time should be removed from the state machine times if requests are being sent to the EJB server by the statemachine controller classes to determine the correct state machine round trip time.