

Stateflow Best Practices

By Michael Burke

Topics

- Background
- Overview of terms
- Readability
 - Stateflow hierarchy
- Modeling tips
- Basic rules: MAAB style guide

Background

- Objective
 - Introduce concepts that will result in Stateflow models that are
 - Readable / Maintainable
 - Testable
 - Efficient
 - This presentation is not targeting power users!***
- Stateflow programming structures support multiple implementation methods
 - Because of this for even simple problems there are multiple ways of implementing a solution
- Understanding the Stateflow semantics and using a consistent set of approaches results in higher quality code that is easier to debug, test and maintain

Terms

The following set of slides define commonly used terms; experienced users may consider skipping these slides

Terms

- **Chart:** A Stateflow chart that contains either
 - State diagrams: A chart that contains State(s)
 - Flow charts: A chart that does not use State(s), only transitions and conditional logic.

NOTES: most Stateflow charts use a mixture of State diagrams and Flow Charts

- **Stateflow Semantics:** rules that define how the charts are evaluated
 - Mealy, Moore, “Classic” Stateflow
- **Stateflow Functions:** discrete functions that can be called from within the Stateflow diagram
 - Graphical functions
 - Simulink functions
 - MATLAB functions

-
- ```

stateDiagram-v2
 [*] --> attc_initialize : (in_attach_ctl_state == ATTO_INITIALIZE)
 attc_initialize --> attc_neutral_wait : (in_attach_ctl_state == ATTO_NEUTRAL_WAIT)
 attc_neutral_wait --> attc_test_mode : (in_attach_ctl_state == ATTO_TEST_MODE)
 attc_test_mode --> attc_test_mode_done : (in_attach_ctl_state == ATTO_TEST_MODE)
 attc_test_mode_done --> attc_neutral_wait : (in_attach_ctl_state == ATTO_NEUTRAL_WAIT)
 attc_neutral_wait --> attc_stopped : (in_attach_ctl_state == ATTO_STOPPED)
 attc_neutral_wait --> attc_go_rev : (in_attach_ctl_state == ATTO_NEUTRAL_WAIT)
 attc_neutral_wait --> attc_go_fwd : (in_attach_ctl_state == ATTO_NEUTRAL_WAIT)
 attc_stopped --> attc_neutral_wait : (in_attach_ctl_state == ATTO_STOPPED)
 attc_stopped --> attc_go_rev : (in_attach_ctl_state == ATTO_STOPPED)
 attc_go_rev --> attc_neutral_wait : (in_attach_ctl_state == ATTO_NEUTRAL_WAIT)
 attc_go_rev --> attc_stopped : (in_attach_ctl_state == ATTO_STOPPED)
 attc_go_fwd --> attc_neutral_wait : (in_attach_ctl_state == ATTO_NEUTRAL_WAIT)
 attc_go_fwd --> attc_stopped : (in_attach_ctl_state == ATTO_STOPPED)
 attc_go_rev --> [*] : (in_attach_ctl_state == ATTO_STOPPED)

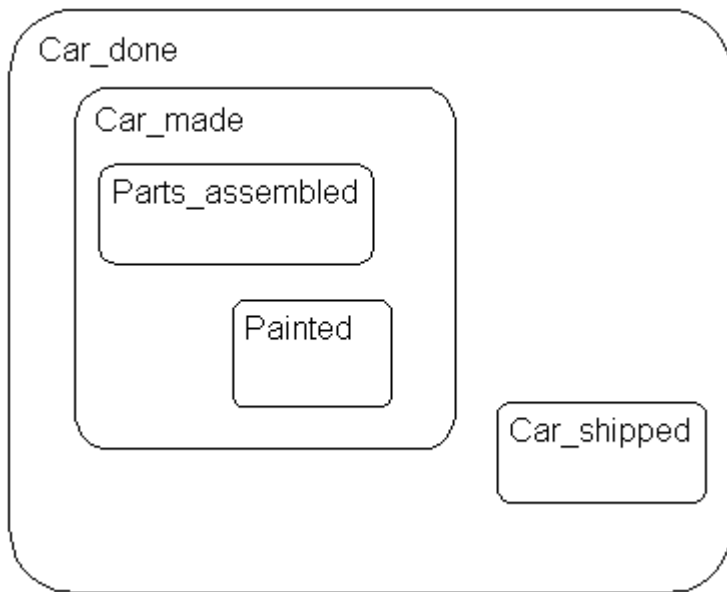
```
- attach\_ctl\_service!  
 ou: checkVoltage(attach\_ctl\_voltage, volt\_10v)
- function test\_mode\_done  
 (attach\_ctl\_test\_mode == ATT\_TEST\_MODE)
- function attc\_set\_rev(flag)  
 (Something = flag)
- function checkVoltage(volt, volt)  
 (volt == volt)(in\_attach\_ctl\_state == ATTO\_INITIALIZE)



## Terms

# States

In the following example, drawing one state within the boundaries of another state indicates that the inner state is a substate (or child) of the outer state (or superstate). The outer state is the parent of the inner state:



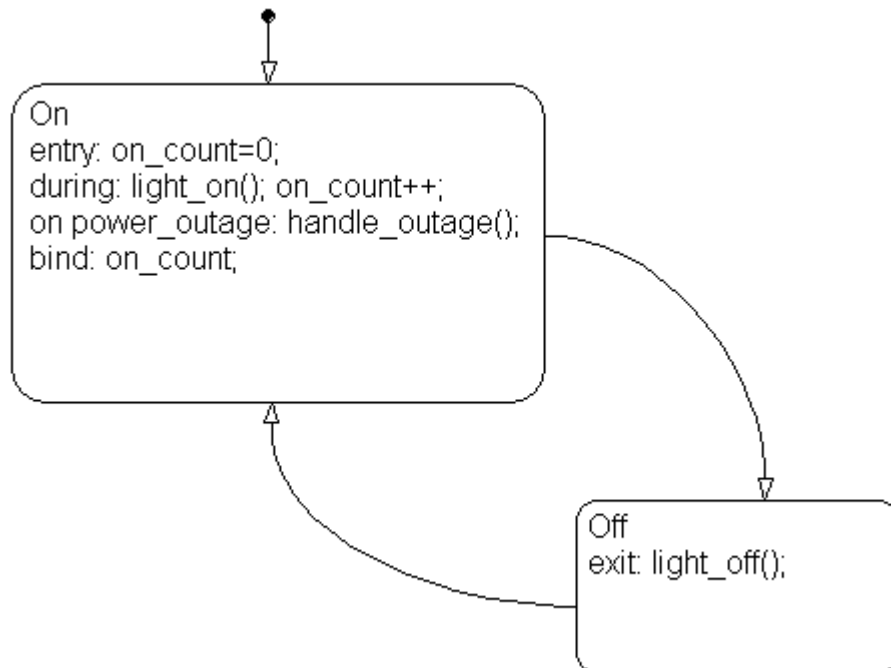
Stateflow hierarchy can also be represented textually, in which the Stateflow chart is represented by the slash (/) character and each level in the hierarchy of states is separated by the period (.) character

- /Car\_done
- /Car\_done.Car\_made
- /Car\_done.Car\_shipped
- /Car\_done.Car\_made.Parts\_assembled
- /Car\_done.Car\_made.Painted

# Terms

## States

**State Actions:** After the name, you enter optional action statements for the state with a keyword label that identifies the type of action. You can specify none, some, or all of them. The colon after each keyword is required. The slash following the state name is optional as long as it is followed by a carriage return.



**Entry Action.** Preceded by the prefix entry or en for short. In the preceding example, state On has entry action on\_count=0. This means that the value of on\_count is reset to 0 whenever state On becomes active (entered).

**During Action.** Preceded by the prefix during or du for short. In the preceding label example, state On has two during actions, light\_on() and on\_count++. These actions are executed whenever state On is already active and any event occurs.

**Exit Action.** Preceded by the prefix exit or ex for short. In the preceding label example, state Off has the exit action light\_off(). If the state Off is active, but becomes inactive (exited), this action is executed.

**On Event\_Name Action.** Preceded by the prefix on event\_name, where event\_name is a unique event. In the preceding label example, state On has an on power\_outage action. If state On is active and the event power\_outage occurs, the action handle\_outage() is executed.

**Bind Action.** Preceded by the prefix bind. In the preceding label example, the data on\_count is bound to the state On. This means that only the state On or a child of On can change the value of on\_count. Other states, such as the state Off, can use on\_count in its actions, but it cannot change its value in doing so.



# Terms

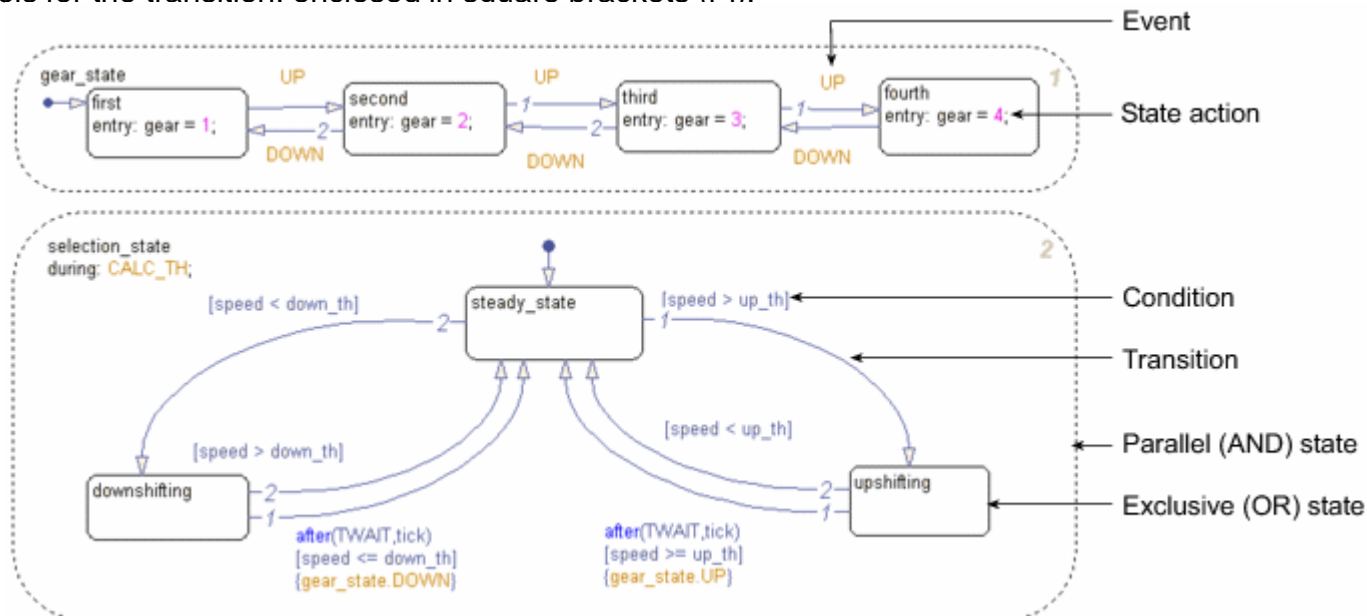
**Exclusive (OR) states.** States that represent mutually exclusive modes of operation. No two exclusive (OR) states can ever be active or execute at the same time. Exclusive (OR) states are represented graphically by a solid rectangle:

**Parallel (AND) states.** States that represent independent modes of operation. Two or more parallel (AND) states at the same hierarchical level can be active concurrently, although they execute in a serial fashion. Parallel (AND) states are represented graphically by a dashed rectangle with a number indicating execution order

**Transitions.** Graphical objects that link one state to another and specify a direction of flow. Transitions are represented by unidirectional arrows:

**State actions.** Actions executed based on the status of a state.

**Conditions.** Boolean expressions that allow a transition to occur when the expression is true. Conditions appear as labels for the transition, enclosed in square brackets (if 1).

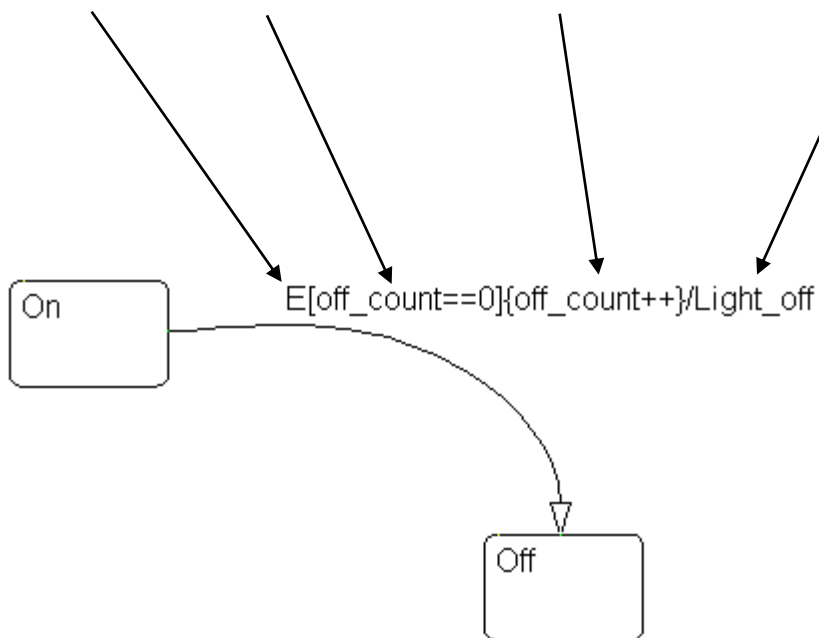


## Terms:

# Transition Notation

A transition is characterized by its label. The label can consist of an event, a condition, a condition action, and/or a transition action. The ? character is the default transition label. Transition labels have the following general format:

event[condition]{condition\_action}/transition\_action




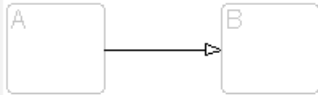

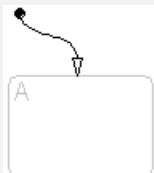

**Event Trigger.** Specifies an event that causes the transition to be taken, provided the condition, if specified, is true. Specifying an event is optional.

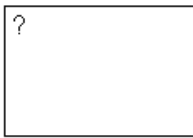
**Condition:** Specifies a Boolean expression that, when true, validates a transition to be taken for the specified event trigger. Enclose the condition in square brackets ([])

**Condition Action.** Follows the condition for a transition and is enclosed in curly braces ({}). It is executed as soon as the condition is evaluated as true and before the transition destination has been determined to be valid.

**Transition Action.** Executes after the transition destination has been determined to be valid provided the condition, if specified, is true. If the transition consists of multiple segments, the transition action is only executed when the entire transition path to the final destination is determined to be valid.

# Terms

| Name                 | Notation                                                                            |
|----------------------|-------------------------------------------------------------------------------------|
| State                |    |
| Transition           |    |
| History Junction     |    |
| Default Transition   |   |
| Connective Junction  |  |
| Truth Table Function | <b><i>truth</i>table</b><br><b>y = func(x)</b>                                      |

| Name                      | Notation                                                                             |
|---------------------------|--------------------------------------------------------------------------------------|
| Graphical Function        | <b><i>function</i></b> y = func(x)                                                   |
| Embedded MATLAB™ Function | <b>eM</b><br>y = func(x)                                                             |
| Box                       |  |

# Terms Data

Stateflow allows you to control both the data type and scope of all data in the chart

| Scope Value       | Description                                                                                                                                                                                                                                                                                                                                                            |
|-------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Local             | Data defined in the current Stateflow chart only.                                                                                                                                                                                                                                                                                                                      |
| Constant          | Read-only constant value that is visible to the parent Stateflow object and its children.                                                                                                                                                                                                                                                                              |
| Parameter         | Constant whose value is defined in the MATLAB <sup>®</sup> workspace, or derived from a Simulink <sup>®</sup> block parameter that you define and initialize in the parent masked subsystem. The Stateflow data object must have the same name as the parameter.                                                                                                       |
| Input             | Input argument to a function if the parent is a graphical, truth table, or Embedded MATLAB <sup>™</sup> function. Otherwise, the Simulink model provides the data to the Stateflow chart via an input port on the Stateflow block.                                                                                                                                     |
| Output            | Return value of a function if the parent is a graphical, truth table, or Embedded MATLAB function. Otherwise, the Stateflow chart provides the data to the Simulink model via an output port on the Stateflow block.                                                                                                                                                   |
| Data Store Memory | Data object that binds to a Simulink data store, which is a signal that functions like a global variable because all blocks in a model can access that signal. This binding allows the Stateflow chart to read and write the Simulink data store, thereby sharing global data with the model. The Stateflow object must have the same name as the Simulink data store. |
| Temporary         | Data that persists only during the execution of a function. You can define temporary data only for a graphical, truth table, or Embedded MATLAB function                                                                                                                                                                                                               |
| Exported          | Data from the Simulink model that is made available to external code defined in the Stateflow hierarchy, as described in <a href="#">Sharing Stateflow<sup>®</sup> Data with External Modules</a> . You can define exported data only for a Stateflow machine.                                                                                                         |
| Imported          | Data parented by the Simulink model that is defined by external code embedded in the Stateflow machine, as described in <a href="#">Sharing Stateflow<sup>®</sup> Data with External Modules</a> . You can define imported data only for a Stateflow machine.                                                                                                          |

# Terms: Transition Types

## What Is a Default Transition?

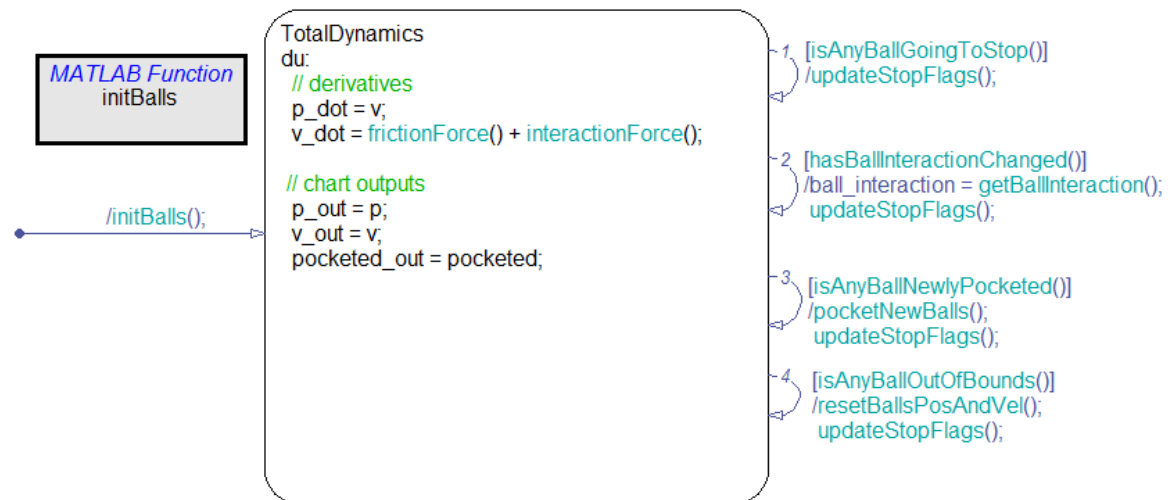
A *default transition* specifies which exclusive (OR) state to enter when there is ambiguity among two or more neighboring exclusive (OR) states. A default transition has a destination but no source object. For example, a default transition specifies which substate of a superstate with exclusive (OR) decomposition the system enters by default, in the absence of any other information, such as a history junction. A default transition can also specify that a junction should be entered by default.

## Inner Transitions

An *inner transition* is a transition that does not exit the source state.

## Self-Loop Transitions

A transition that originates from and terminates on the same state is a self-loop transition. The following chart contains four self-loop transitions:



## What Is a Supertransition?

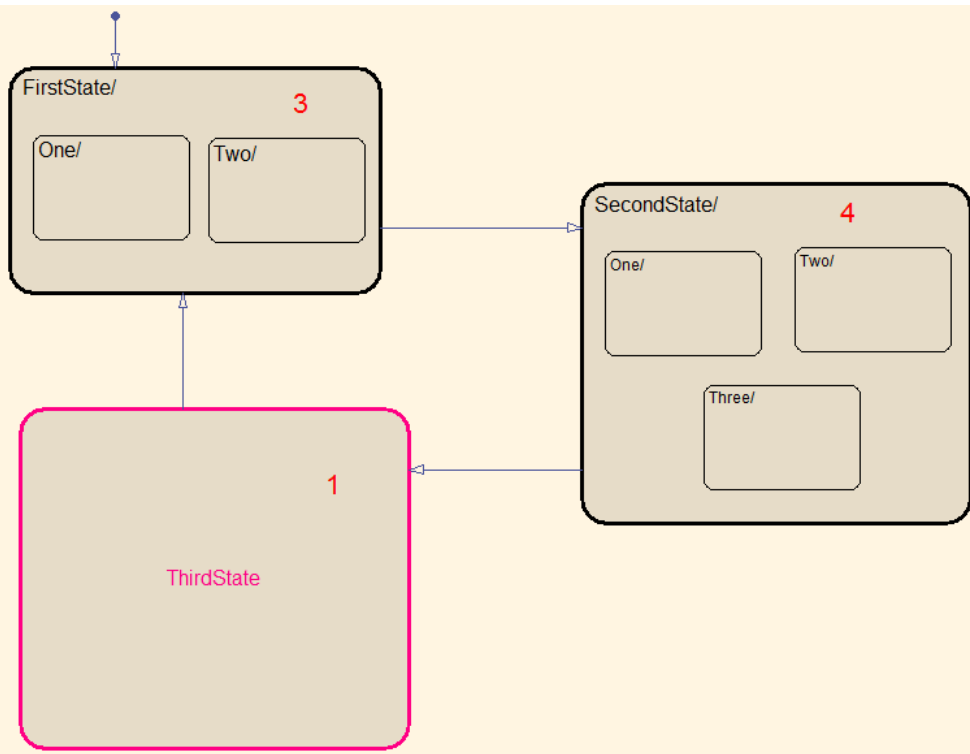
A *supertransition* is a transition between different levels in a chart, for example, between a state in a top-level chart and a state in one of its subcharts, or between states residing in different subcharts at the same or different levels in a chart. You can create supertransitions that span any number of levels in your chart, for example, from a state at the top level to a state that resides in a subchart several layers deep in the chart.

The point where a supertransition enters or exits a subchart is called a *slit*. Slits divide a supertransition into graphical segments.

## Readability

### Stateflow Hierarchy: States per level

- **Limit 6 ~ 10 “states” per level of the Stateflow chart**
  - Subcharted and Atomic Subcharted States count as a single ‘chart’
  - For nested States count the States inside the top level state



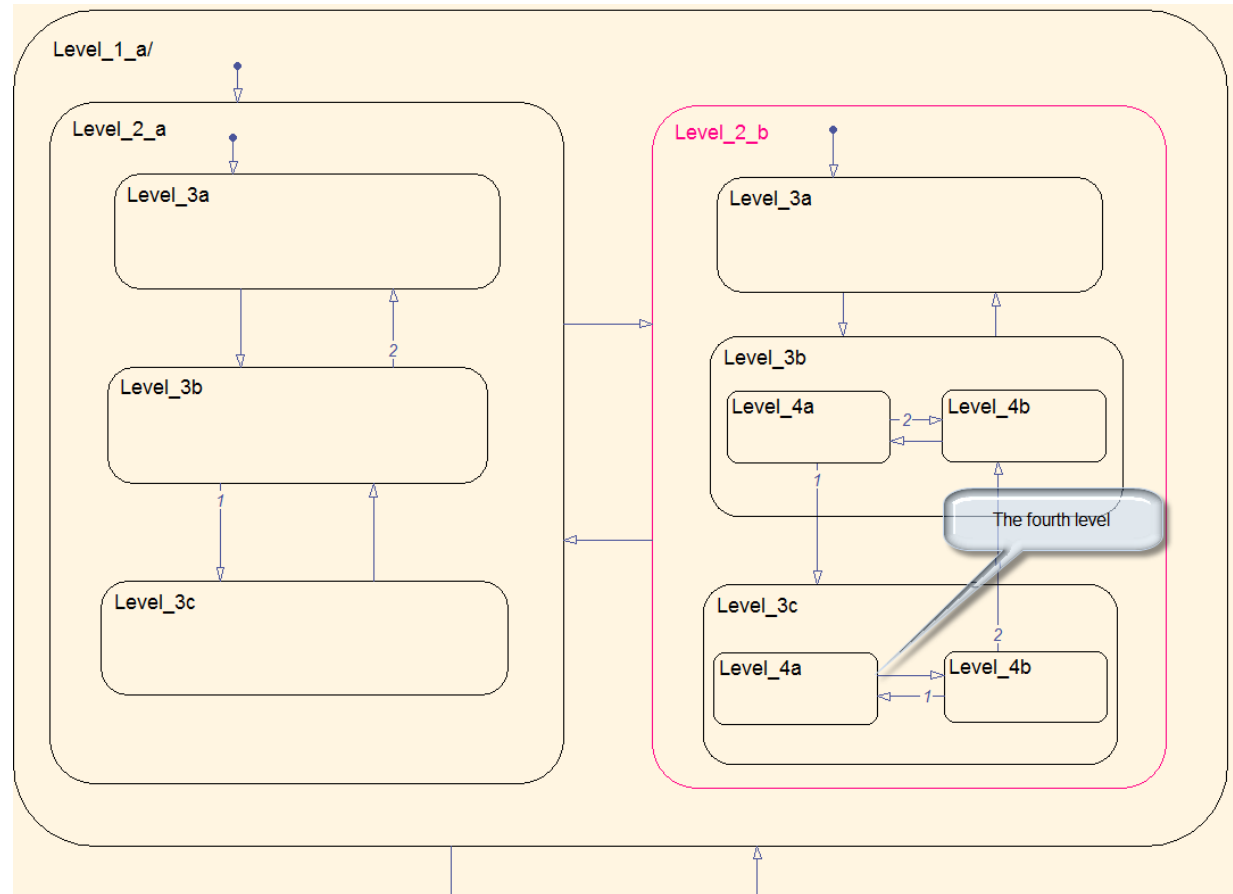
This example has a count of 8 States

- FirstState: 3 states (self + 2)
- SecondState: 4 states (self + 3)
- ThirdState: 1 state (self)

## Readability

### Stateflow Hierarchy: Nesting of states

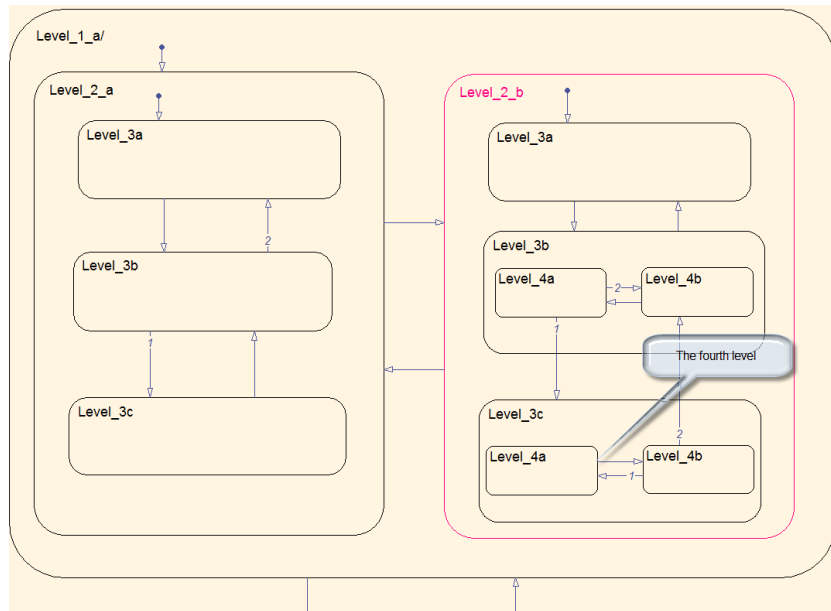
- Limit the number of **nested States to 3 per level**
  - Consider sub-charting the states when they are more than three deep



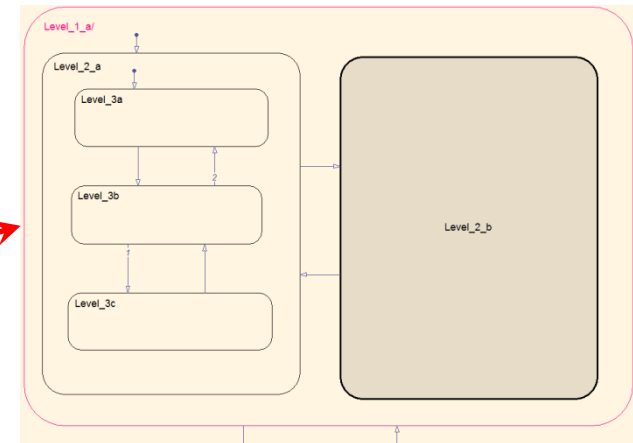
# Readability

## Stateflow Hierarchy: Grouping with sub-charts

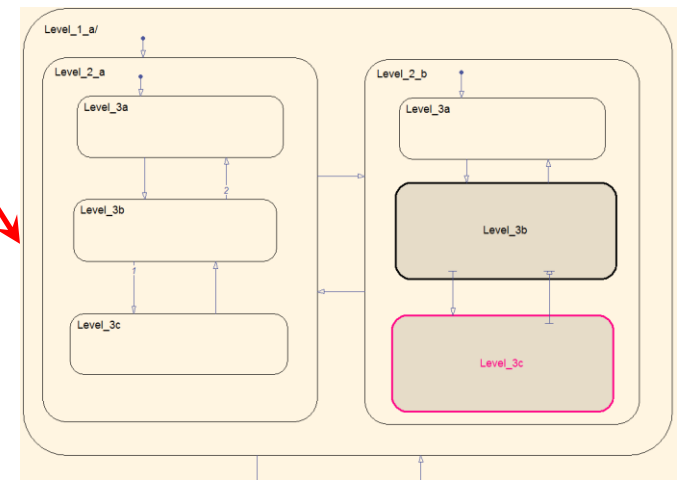
Original



Good



Ok...



In this example the preferred regrouping encapsulated several layers through sub-charting. Following this approach all the transitions are source states are visible.



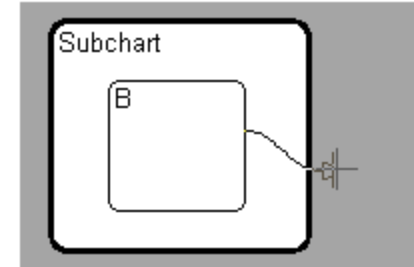
## Readability

# Super Transitions

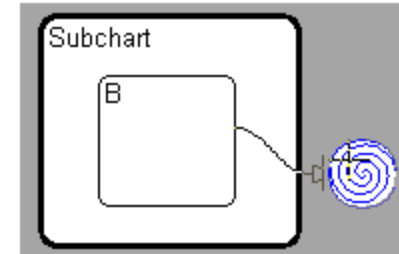
To improve readability when grouping states together into sub charts minimize transitions into and out of the chart (Super Transitions)

Note: In 12b and later the wormhole is not used

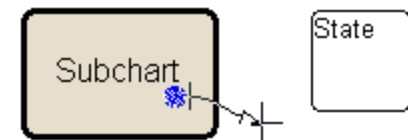
1. Draw an inner transition segment from the source object anywhere just outside the border of the subchart. A slit appears as shown.



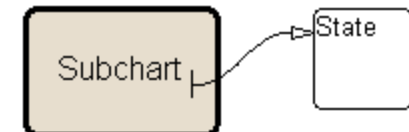
2. Keep dragging the transition away from the border of the subchart. A wormhole appears.



3. Drag the transition down the wormhole. The parent of the subchart appears.



4. Complete the connection.

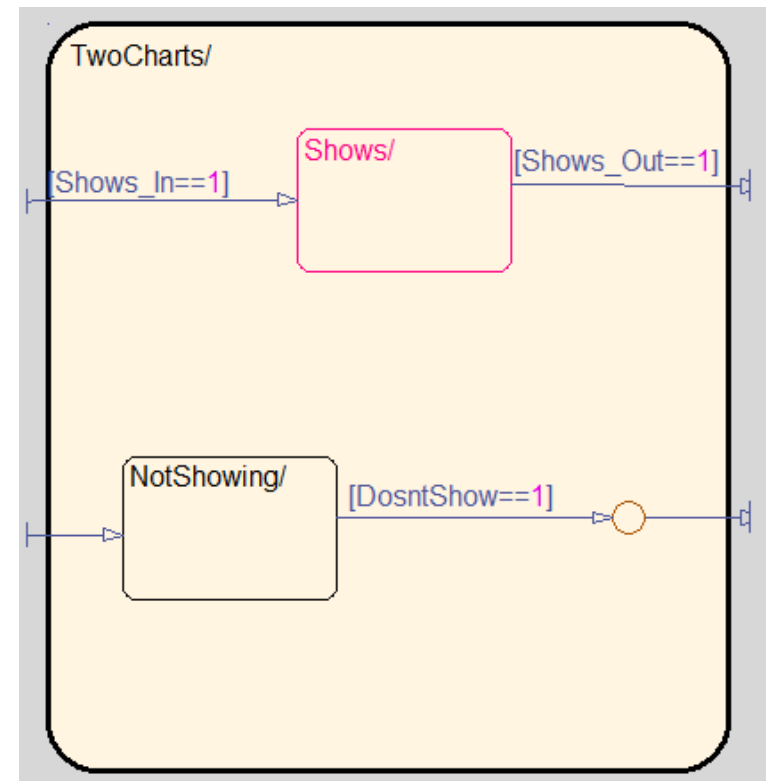
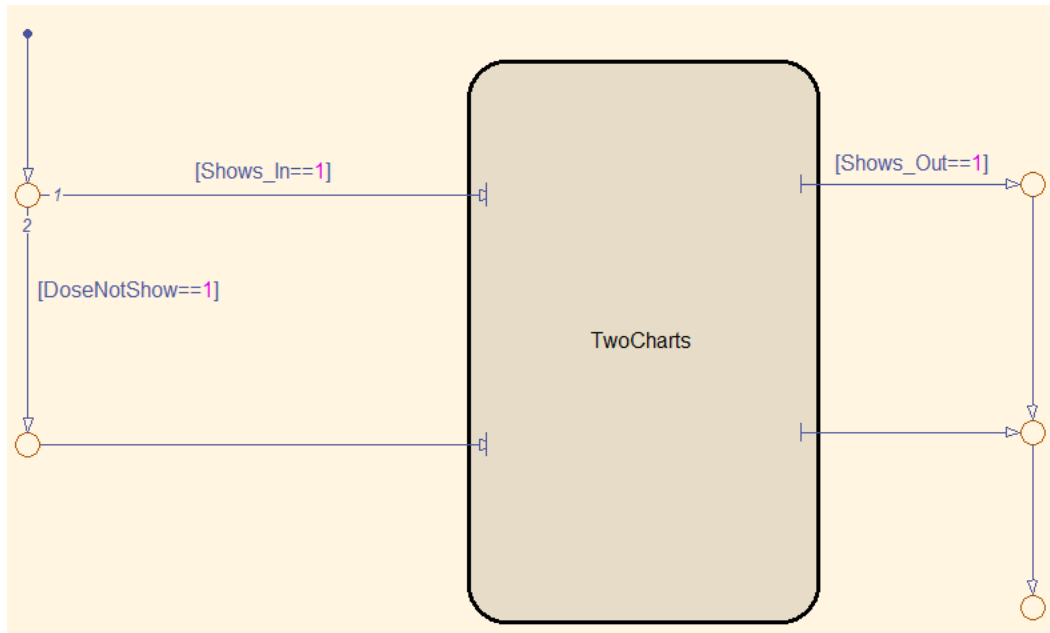


## Readability

### Super Transitions (cont)

Text in the 'source' state will be displayed in the destination state if it is on the line segment connected to the slit.

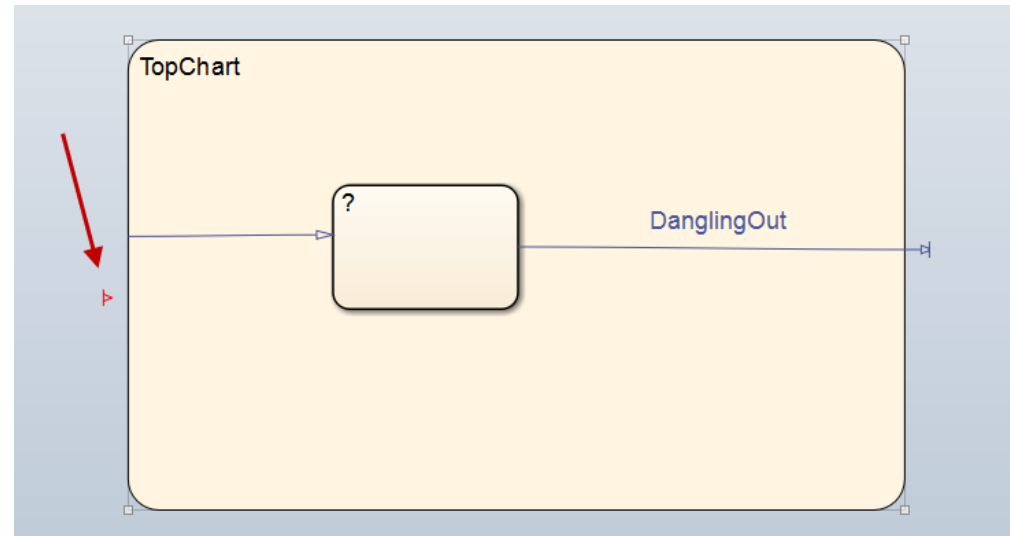
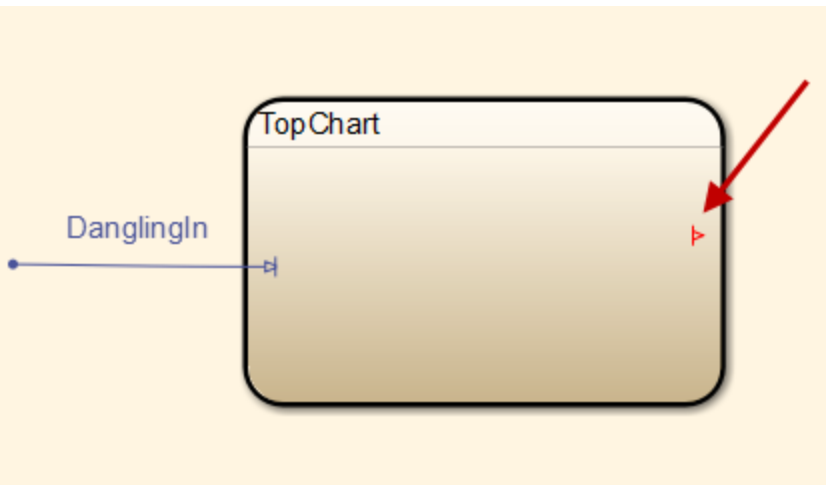
If there are multiple super transitions place the condition logic on the line segment connected to the slit.



## Readability

# Super Transitions

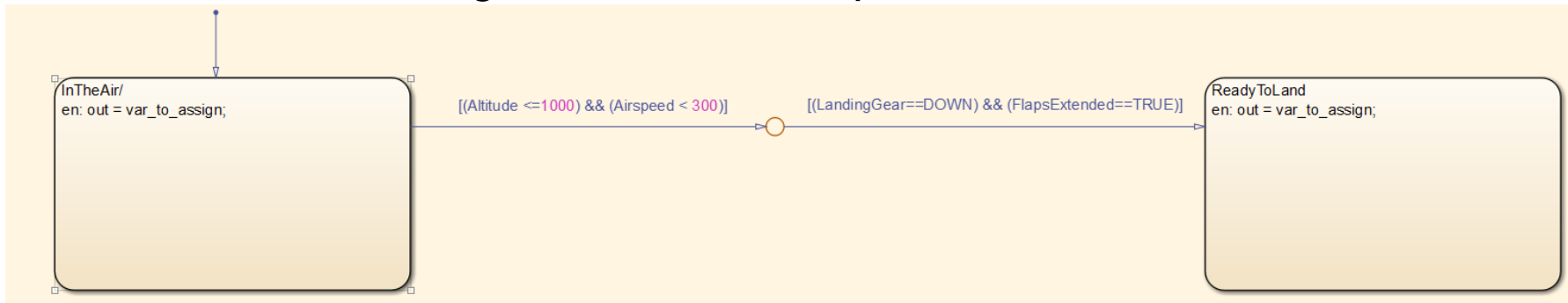
- **Do not leave “Dangling” Super Transitions**
  - Dangling super transitions can occur when an inner condition is created by mistake



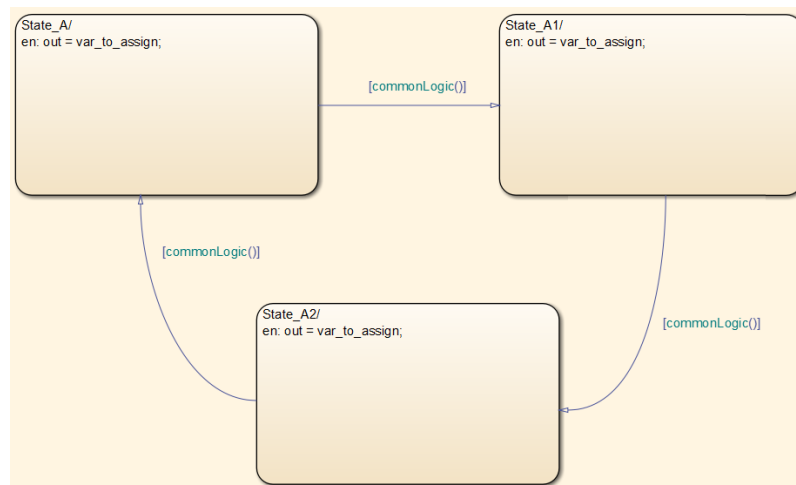
# Readability Transitions

For longer transition logic consider

- **Splitting the logic into multiple line segments**
- Consider creating a graphical function when the same conditional logic is used in multiple locations



In addition to splitting by length the logic was split thematically...

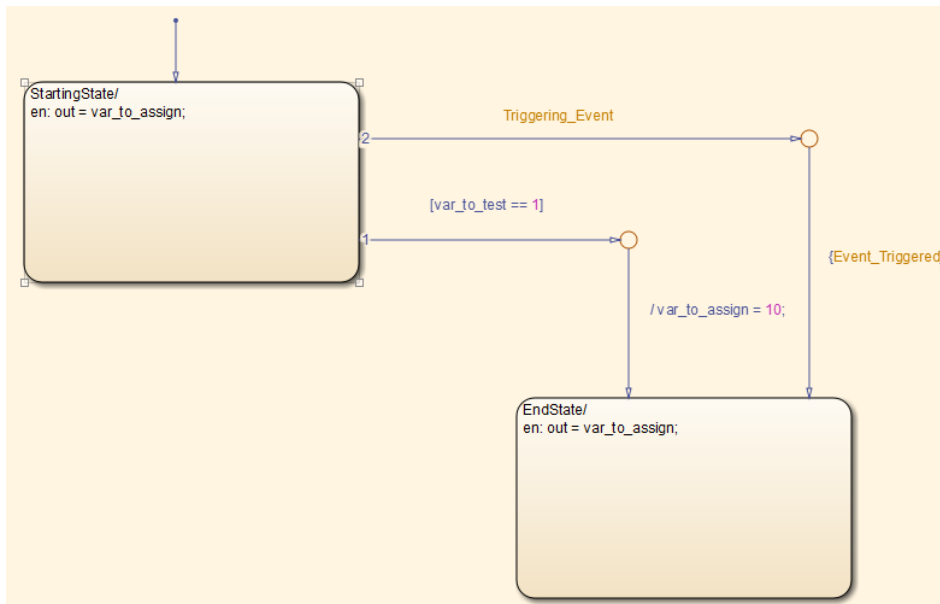


To aid readability use meaningful function names

## Readability

# Conditions and Actions

- Adopt a consistent method for condition and transition actions
  - For maximum clarity consider placing the actions on a separate segment of the transition

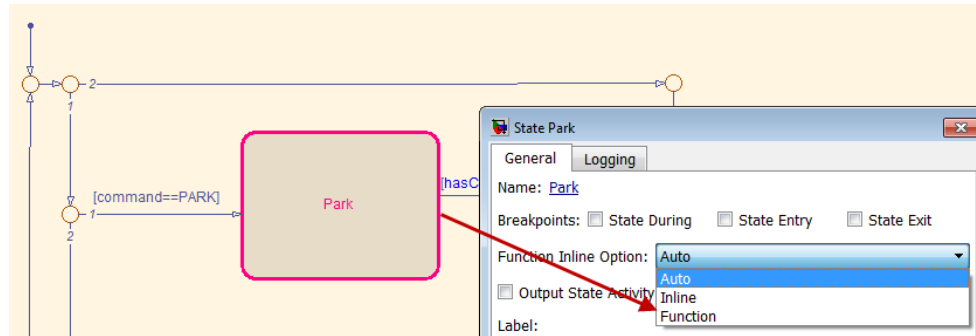


In this example the conditions are drawn horizontally and the assignments are on the vertical

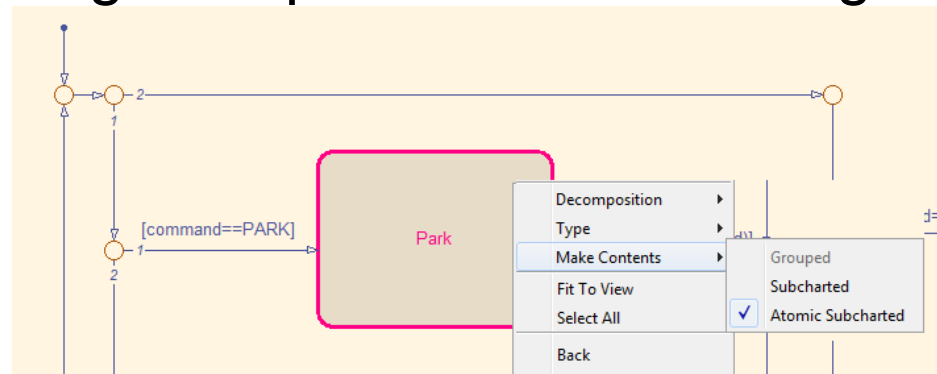
# Stateflow architecture

## Use of functions

- Use the explicit “Function” setting sparingly



- If function partitioning is required consider using an Atomic Subchart



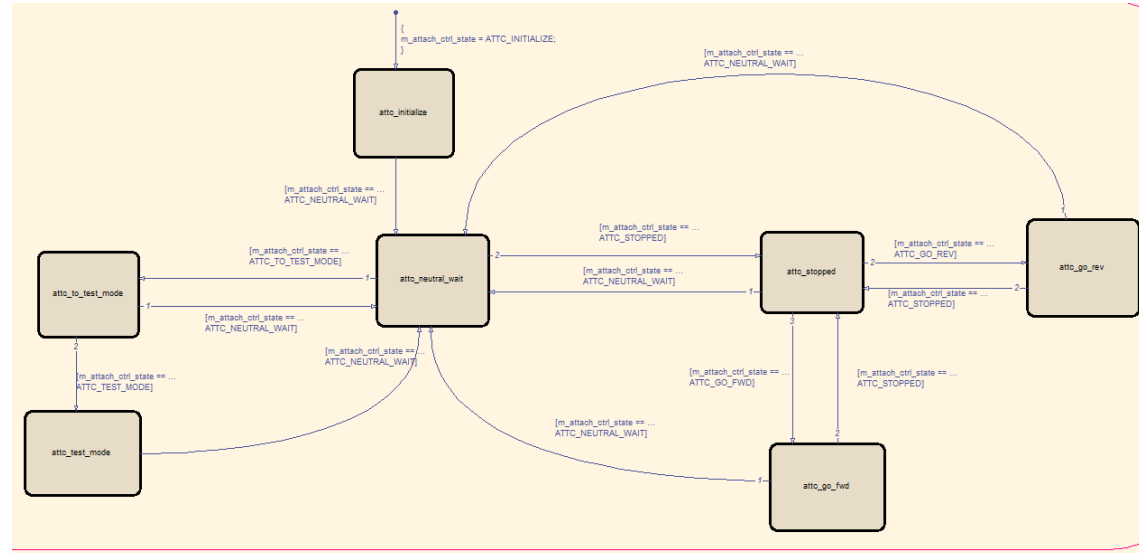
Function introduce memory and processing over head; if the State needs to be partitioned into a function using an Atomic Subchart allows you to independently test the resulting State chart / function.

# Stateflow architecture

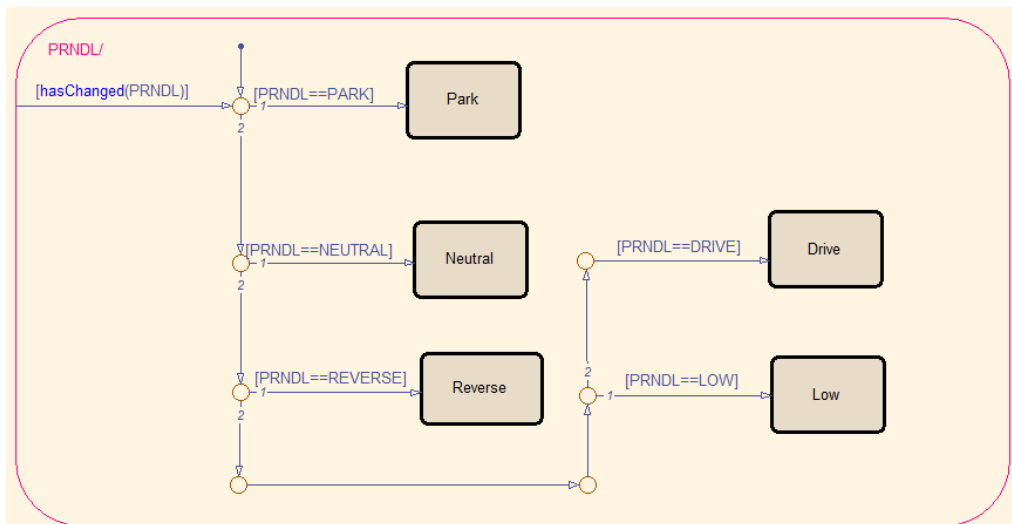
## Selecting transition style

One to many verses One to some...

In some instances transitions can go between any state in the chart, for example the states in a transmission shift column. In these instance ladder logic as shown below is appropriate.



However the in most cases a transition centric view (as above) of the States is more **efficient** and provides a more intuitive way of understanding the relationship between states.



## Architecture

# Use a State Diagrams, Flow chart or Truth Table?

Most Stateflow charts are mixtures of the three basic types however each has a primary use...

### **Truth Tables:** (combinatorial logic)

- logical comparisons with a limited set of output modes
- The output mode is dependent on all the input conditions

### **Flow charts: (decision tree)**

- Combination of logical comparisons and basic math and loop constructs
- Does not depend on state information

### **State diagram**

- Mode based modeling where outputs are dependent on state variables

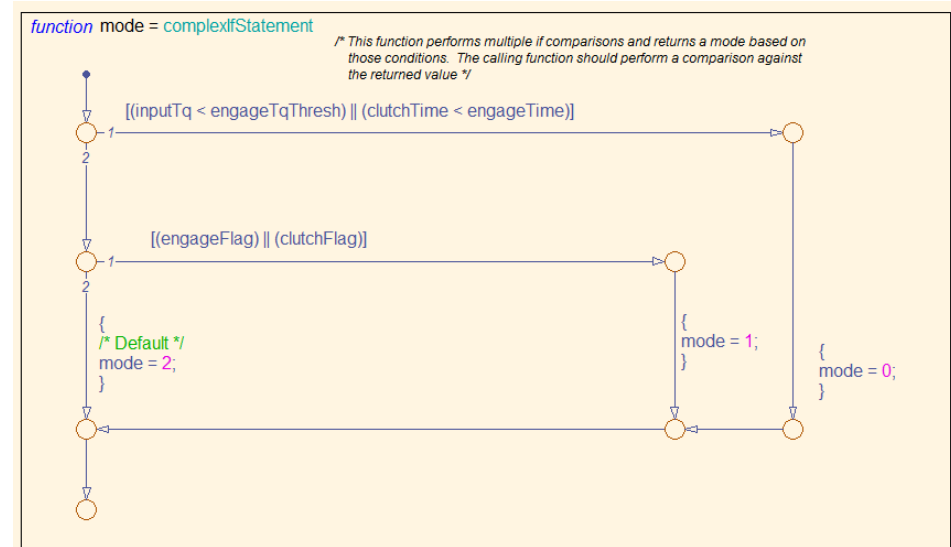
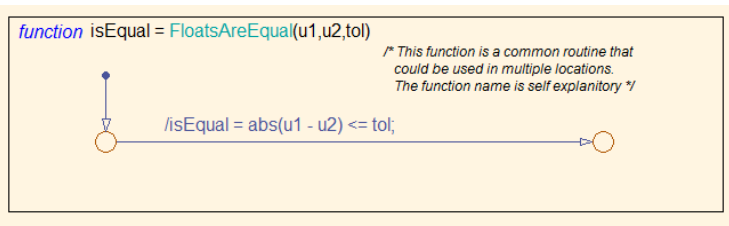
### **State transition table (structured interface for State diagrams)**



# Tips

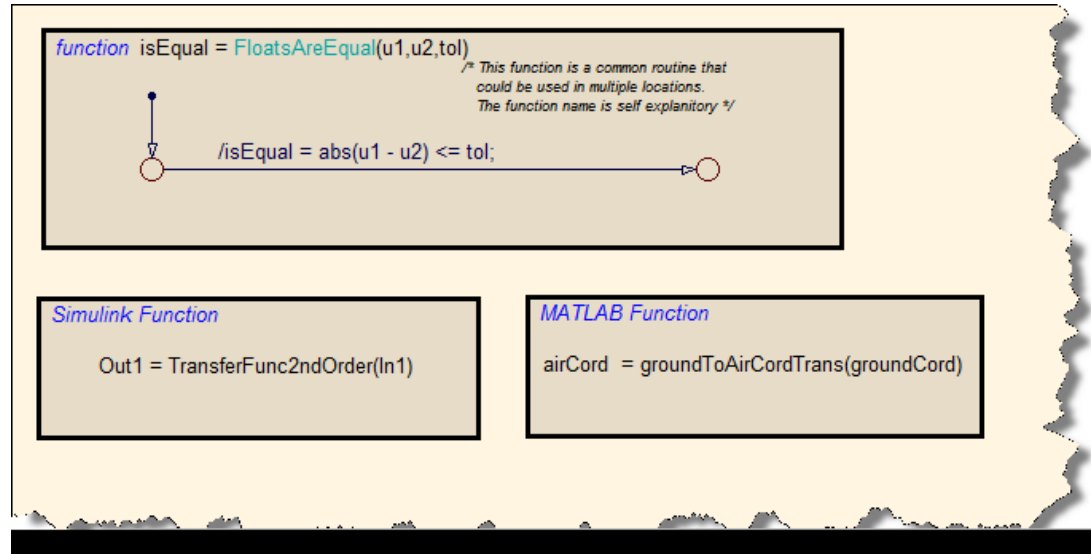
## Use of Graphical functions

- Use **Graphical functions** to
  - Improve chart readability
  - Ensure consistent implementation of common operations
- Despite the name graphical functions can either generate functions or inlined code.



# Tips

## Reuse of graphical functions



### How to Reuse Functions with an Atomic Box

To reuse functions across multiple models:

1. Create a library model with an atomic box that contains the function you want to reuse.
2. Create a separate model with multiple charts.
  - a. In each chart that calls the function, add a linked atomic box.
  - b. Write each call to the function using the full path:

`linked_box_name.function_name`

Using the full path for the function call has the following advantages:

- Makes clear the dependency on the function in the linked atomic box
- Avoids pollution of the global namespace
- Does not affect efficiency of the generated code

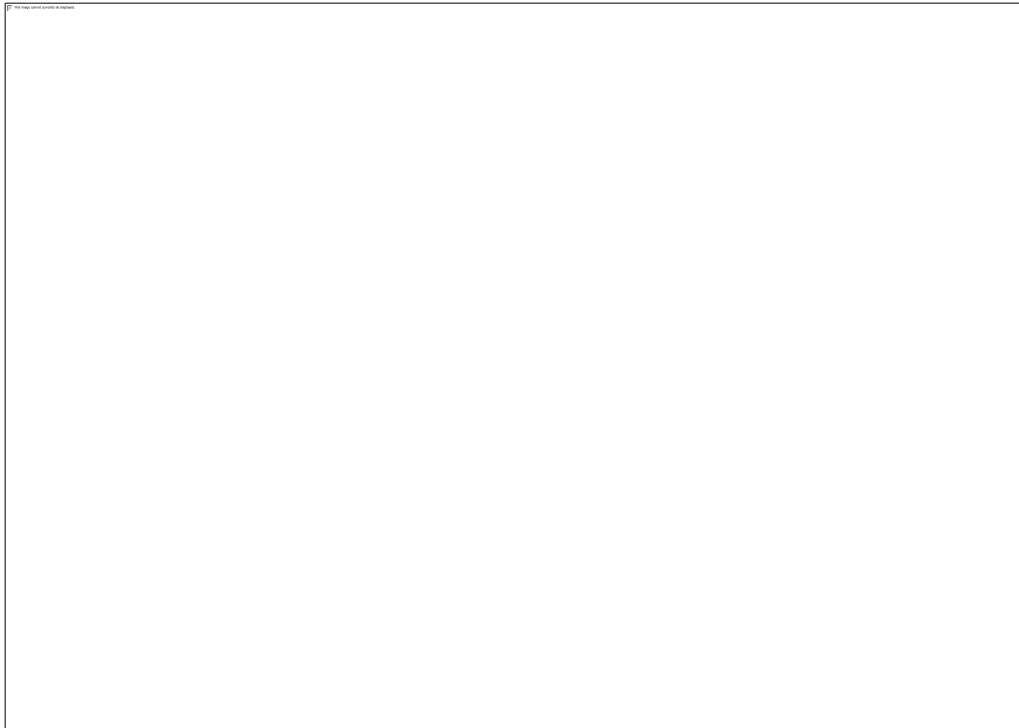
3. Reuse that model multiple times as referenced blocks in a top model.

Because there are no exported functions in the charts, you can use more than one instance of that referenced block in the top model.

## Tips

# Understanding calls to graphical functions

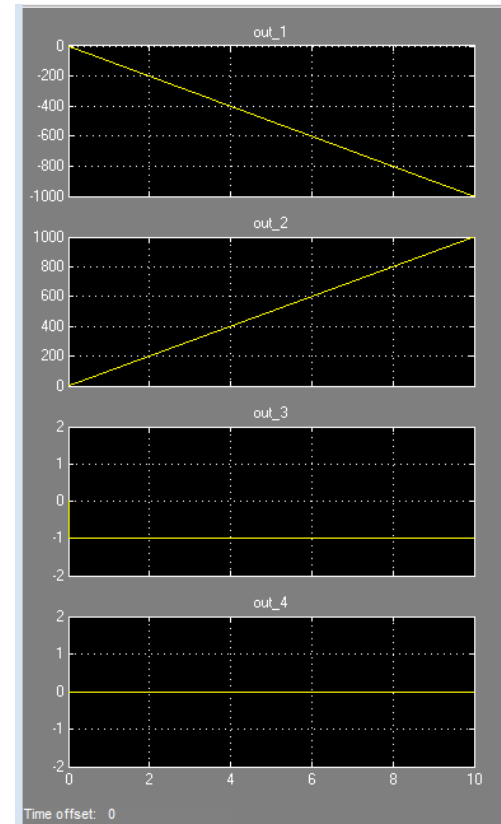
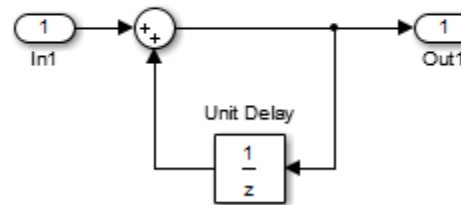
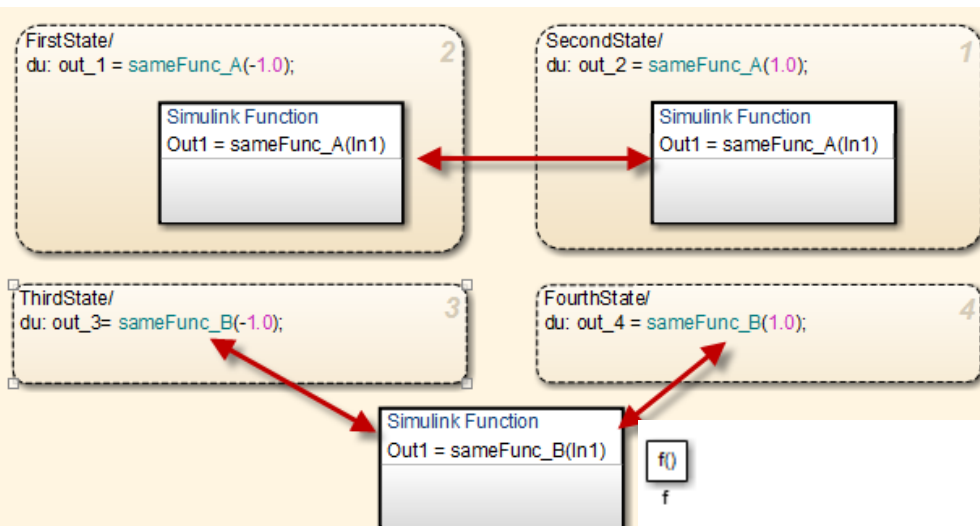
- Graphical functions can be called multiple times during the same time step from a single chart
- If the graphical function contains state information then the state information **may** be updated multiple times
  - If the block uses a  $\Delta t$  only the first call to the function updates the state (Integrators, transfer functions)
  - Else every call to the function updates the state (Unit Delay)



## Tips

# Graphical functions: **Scope**

- Graphical functions use the scope of the parent state
  - If the same function is **in** two or more States then it will be generated as a reusable function with **unique states**
  - The same function **used** by multiple functions it will be a single function with a **shared state**.



## Tips

# Selecting a MATLAB, Simulink, or Graphical function

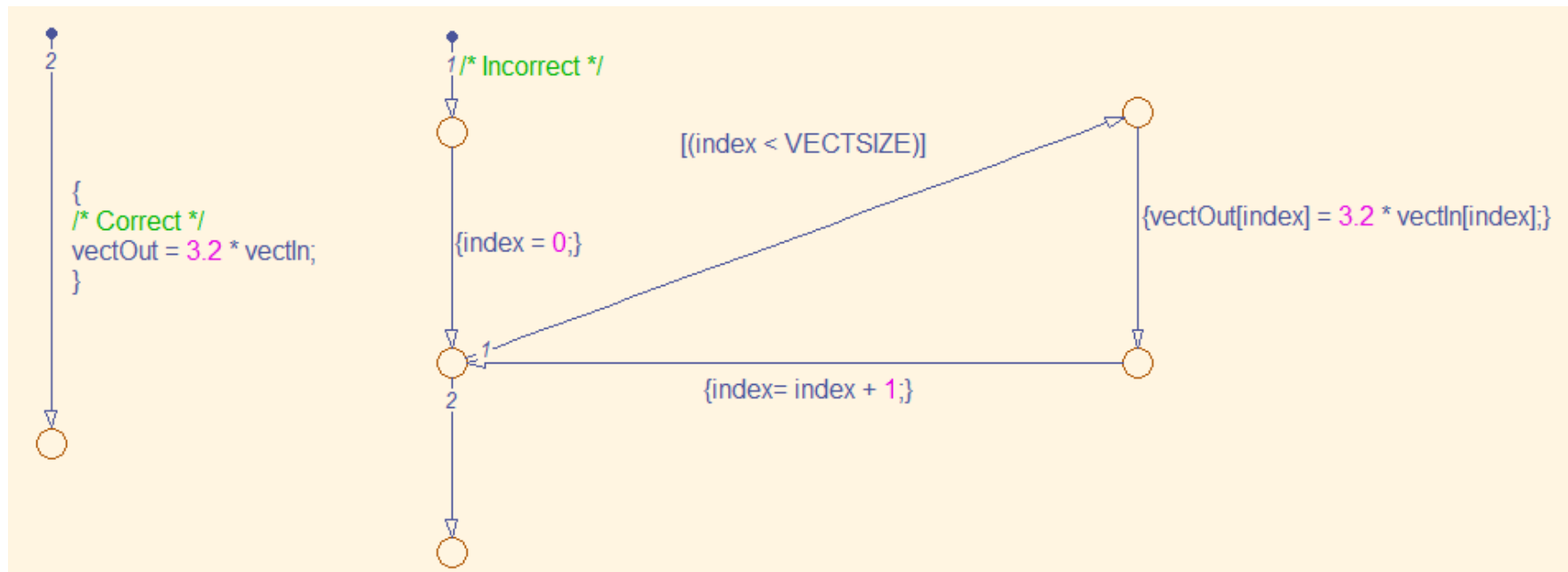
Use...

- MATLAB for complex math
- Simulink for traditional control problems
  - Transfer functions, integrators, filters...
- Graphical functions for
  - if / then / else
  - Loop control

## Tips

# Matrix Math

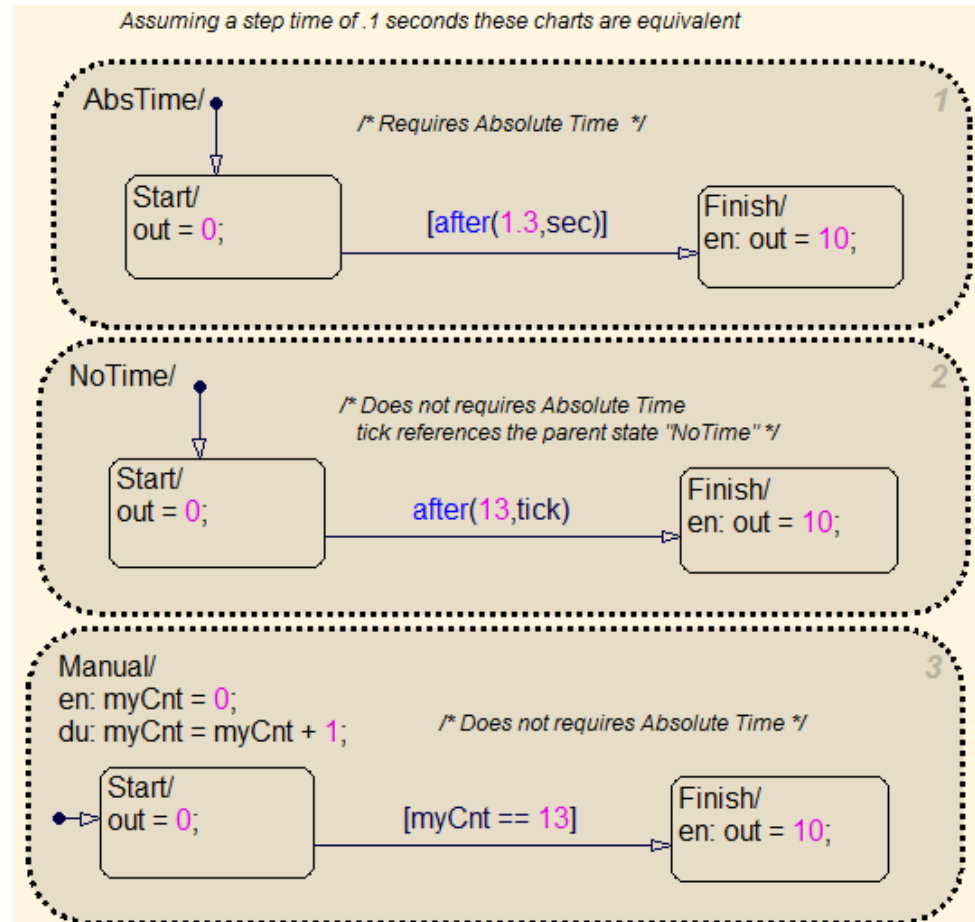
- Simulink and Stateflow natively handle matrix operations
  - Do not use For loops to execute matrix math



## Tips

# Temporal logic verses Counters

- Stateflow's temporal logic functions reduces 'book keeping' operations
  - e.g. the State Manual initializes and increments a counter
- Use of the 'sec' key word use of absolute time which may not be compatible with embedded controller applications



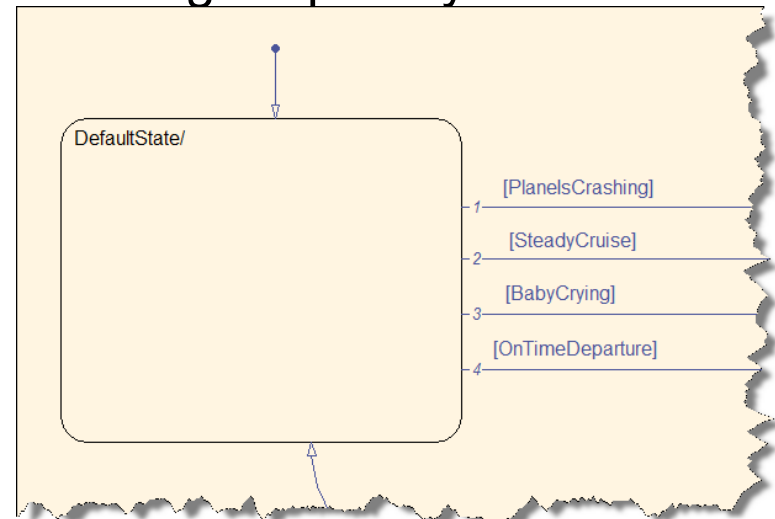
## Tips

### Prioritization of transition order

- When multiple transition paths exist set the order of evaluation from most common to least common
  - The exception to this rule is when a transition is for a high integrity check that should always be evaluated

In this example the most common condition is “SteadyCruise” however the condition “PlanelCrashing” has a higher priority due to the emergency nature of the condition

Note: making PlanelCrashing an event may be the correct implementation in this example



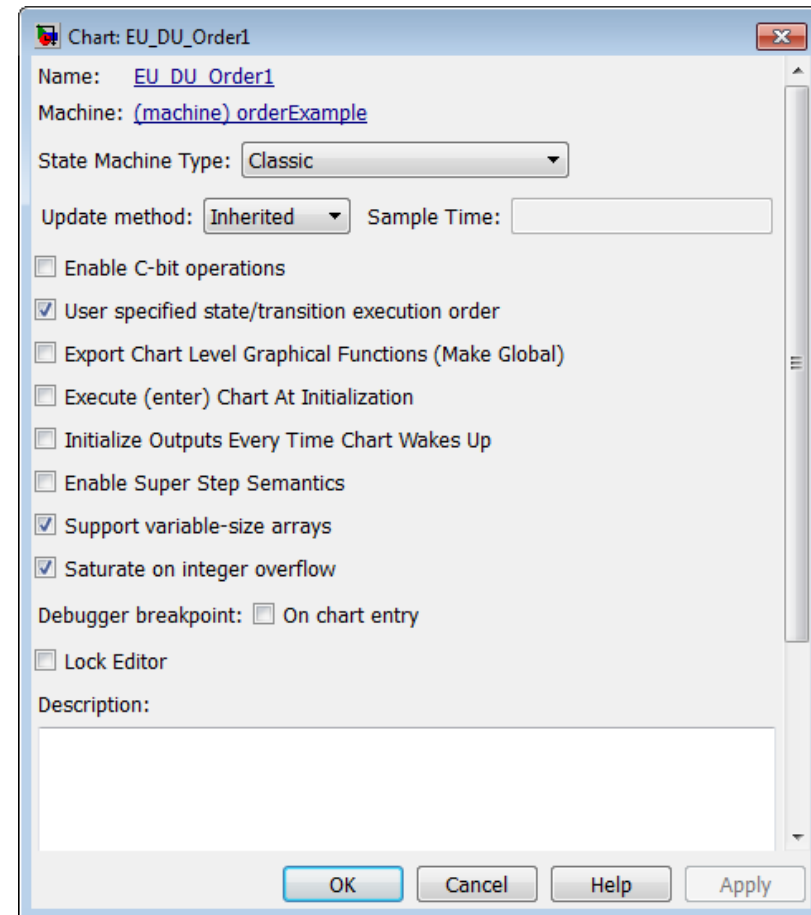


## Tips

# Stateflow Options....

The Stateflow parameters allow you to customize chart behavior

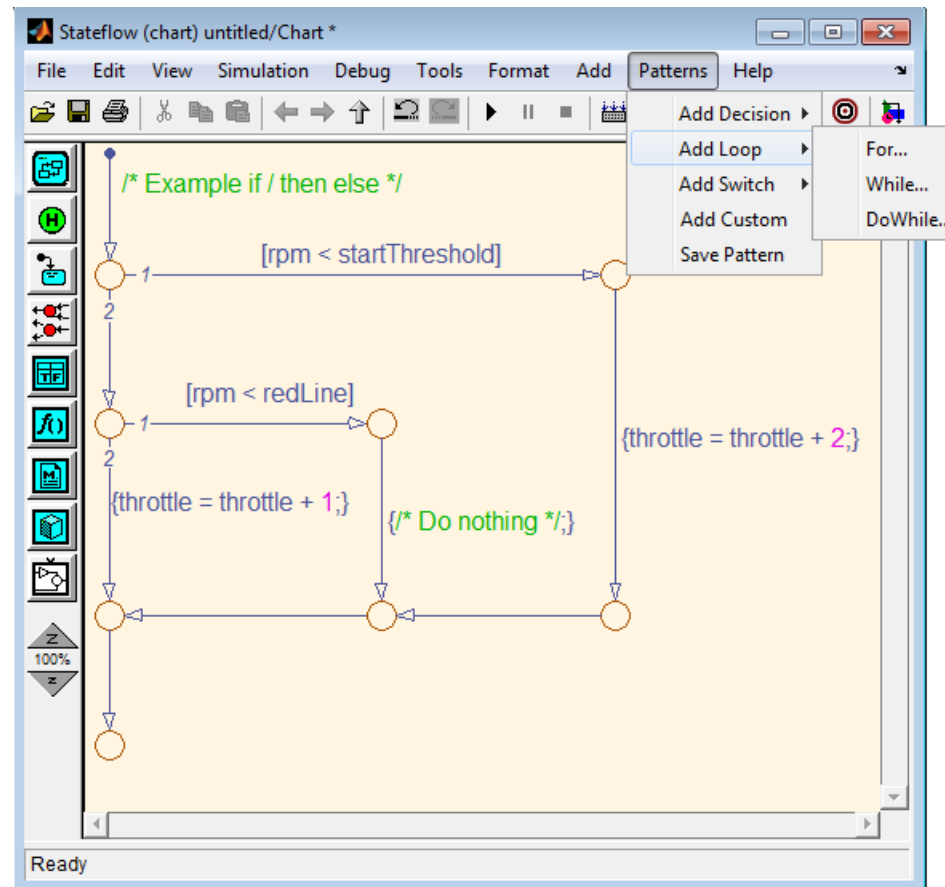
- Use consistent settings across a project



## Tips

# Stateflow Patterns

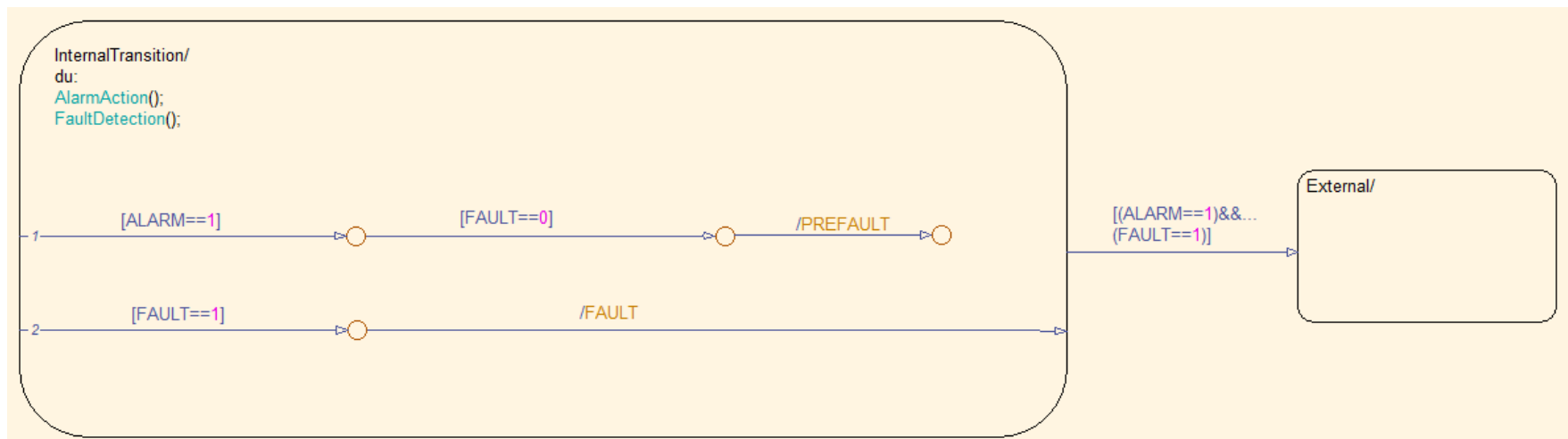
- When possible use standard patterns for creation of flow charts



## Tip

# Use of Internal transitions

- Internal transitions can be used to conditionally set the value of variables. They are...
  - Executed after DU
  - Executed before external transitions
- They can be used instead of external transitions that loop back into the state
  - Internal transitions that terminate on the state will trigger entry actions

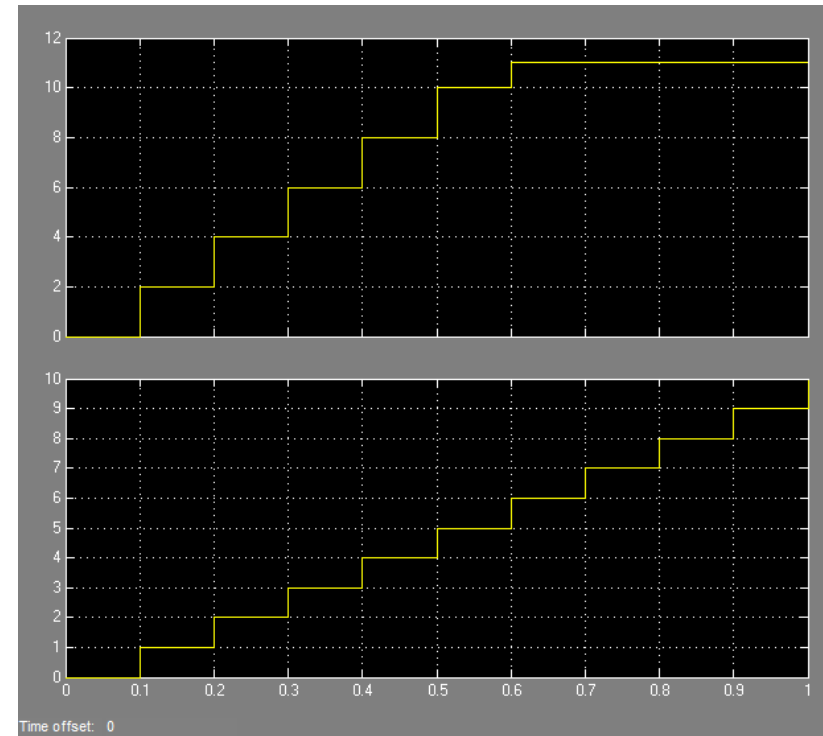
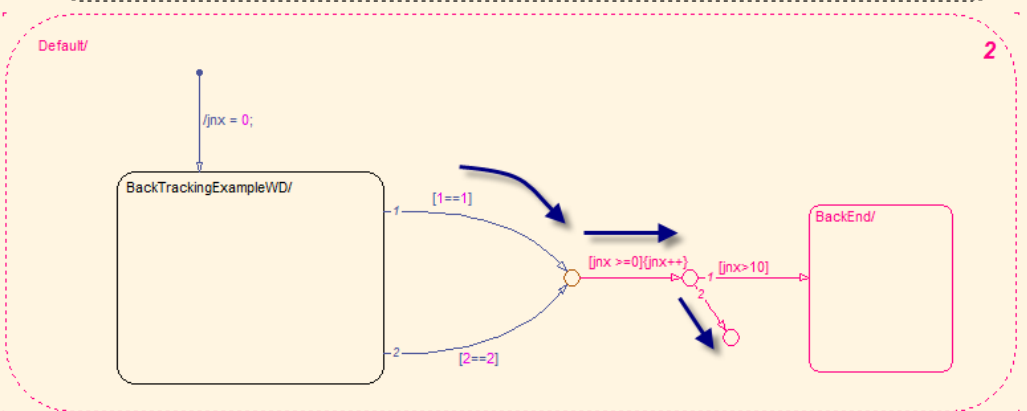
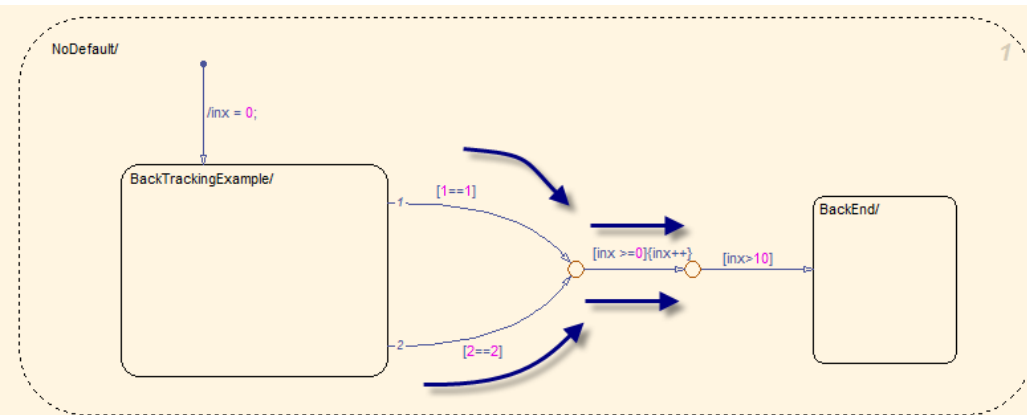


## Tip

# Backtracking and Condition Actions

- Condition Actions execute every time the condition evaluates true
  - Backtracking can result in the same evaluation being performed multiple times

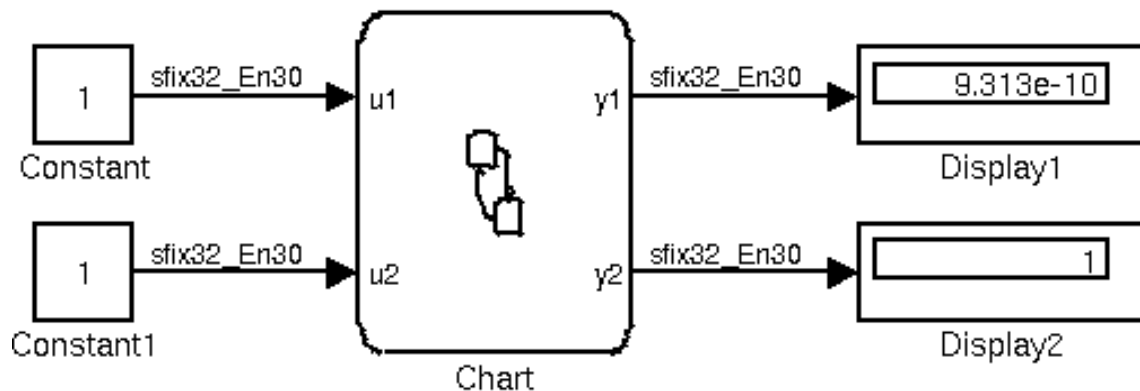
Due to back tracking the incrimination operation takes place twice in the top state; the default transition in the bottom state keep prevents this



## Stateflow fixed point

# Use the colon equals (:=) operator

- Allows you to specify accumulator type (like Sum block of Simulink)



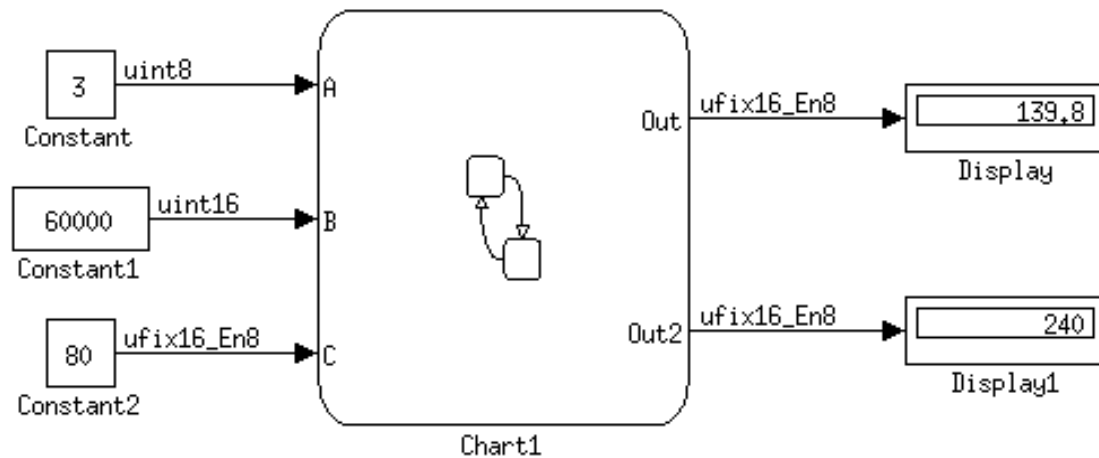
```
{
 //No colon equals
 y1 = u1 * u2;

 // Colon equals
 y2 := u1 * u2;
}
```

For “`y2 := u1 * u2`” the accumulator type (for `*`) is the same as the type of `y2`

## Stateflow fixed point

# Only use a single operation on a line



```

{
 Out = (A * B * C) / 60000;

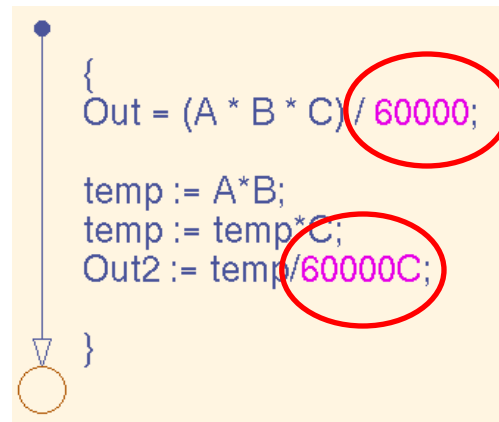
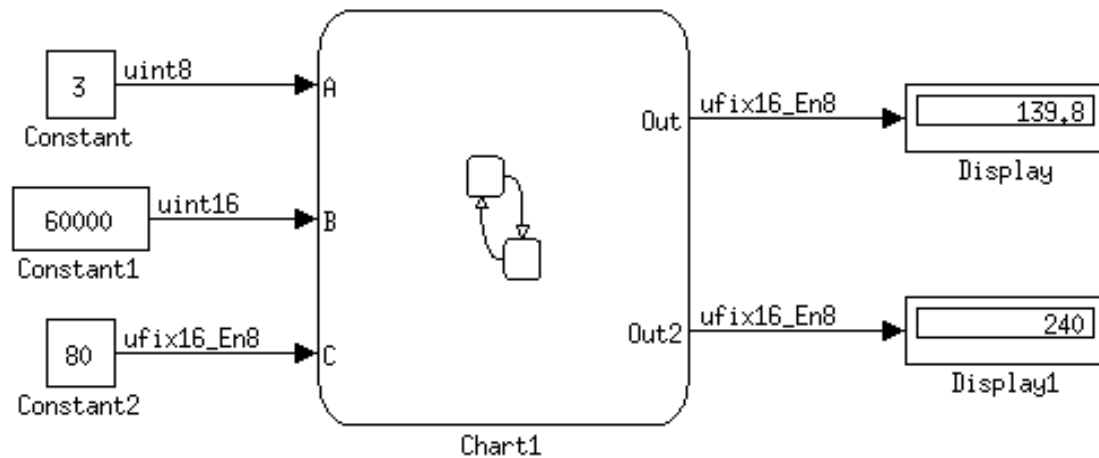
 temp := A*B;
 temp := temp*C;
 Out2 := temp/60000C;
}

```

This is because the colon equals operator only applies to a single operator

# Stateflow fixed point

## Use Context sensitive constants



With 60000C, the type used for the constant is a fixed point type instead of double.

## Stateflow Basics:

# MAAB Rules: Readability

- **db\_0129: Stateflow transition appearance**
- **db\_0133: Use of patterns for Flowcharts**
- **db\_0132: Transitions in Flowcharts**
- **jc\_0501: Format of entries in a State block**
- **jc\_0501: Format of entries in a State block**
- **db\_0150: State machine patterns for conditions**
- **db\_0151: State machine patterns for transition actions**
- **db\_0148: Flowchart patterns for conditions**
- **db\_0149: Flowchart patterns for condition actions**
- **db\_0134: Flowchart patterns for If constructs**
- **db\_0159: Flowchart patterns for case constructs**
- **db\_0135: Flowchart patterns for loop constructs**



## Stateflow Basics:

# MAAB Rules: Interface

- **db\_0123: Stateflow port names**
- **jc\_0511: Setting the return value from a graphical function**
- **jc\_0521: Use of the return value from graphical functions**
- **db\_0122: Stateflow and Simulink interface signals and parameters**
- **db\_0125: Scope of internal signals and local auxiliary variables**
- **jc\_0491: Reuse of variables within a single Stateflow scope**
- **db\_0127: MATLAB commands in Stateflow**
- **jm\_0011: Pointers in Stateflow**
- **db\_0126: Scope of events**
- **jm\_0012: Event broadcasts**

## Stateflow Basics:

# MAAB Rules: Math

- **na\_0001: Bitwise Stateflow operators**
- **jc\_0451: Use of unary minus on unsigned integers in Stateflow**
- **na\_0013: Comparison operation in Stateflow**
- **jc\_0481: Use of hard equality comparisons for floating point numbers in Stateflow**
- **jc\_0541: Use of tunable parameters in Stateflow**