

REVISIONS			
REV.	DESCRIPTION	DATE	APPROVED

## Matlab modelling guidline

Designed by: Gergely Kálmán

**HYUNDAI**

Technologies Center Hungary Ltd.  
H-1146 Budapest, Hermina Road 22.

Controlled by:

**Matlab modelling guidline**

Approved by: László Bojtor

Drawing Number:

**PE10006042**

Date:2014.02.23

Number of pages:

32Hiba!

**A könyvjelző nem létezik.**

## 2.1. Motivation

The MAAB guidelines are an important basis for project success and teamwork - both in-house and when cooperating with partners or subcontractors. Observing the guidelines is one key prerequisite to achieving

- ☐ System integration without problems
- ☐ Well-defined interfaces.
- ☐ Uniform appearance of models, code and documentation
- ☐ Reusable models
- ☐ Readable models
- ☐ Problem-free exchange of models
- ☐ A simple, effective process
- ☐ Professional documentation
- ☐ Understandable presentations
- ☐ Fast software changes
- ☐ Cooperation with subcontractors
- ☐ Handing over of research or predevelopment projects to product development

### 3.1.1. na\_0026: Consistent software environment

During software development, it is recommended that a consistent software environment is used across the project. Software includes, but is not limited, to:

- ☐ MATLAB
- ☐ Simulink
- ☐ C Compiler (for simulation)
- ☐ C Compiler (for target hardware)

Consistent software environment implies that the same version of the software is used across the full project. The version number applies to any patches or extensions to the software used by a group.

### 4.1.1. ar\_0001: Filenames

A filename conforms to the following constraints:	
FORM	filename = name.extension <b>name:</b> no leading digits, no blanks <b>extension:</b> no blanks
UNIQUENESS	<ul style="list-style-type: none"><li><input type="checkbox"/> all filenames within the parent project directory</li><li><input type="checkbox"/> cannot conflict with C / C++ or MATLAB keywords</li></ul>
ALLOWED CHARACTERS	<b>name</b> a b c d e f g h i j k l m n o p q r s t u v w x y z A B C D E F G H I J K L M N O P Q R S T U V W X Y Z 0 1 2 3 4 5 6 7 8 9 _ <b>extension:</b> a b c d e f g h i j k l m n o p q r s t u v w x y z A B C D E F G H I J K L M N O P Q R S T U V W X Y Z 0 1 2 3 4 5 6 7 8 9
UNDERSCORES	<b>name:</b> <ul style="list-style-type: none"><li>• can use underscores to separate parts</li><li>• cannot have more than one consecutive underscore</li><li>• cannot start with an underscore</li><li>• cannot end with an underscore</li></ul> <b>extension:</b> <ul style="list-style-type: none"><li>• should not use underscores</li></ul>

#### 4.1.2. ar\_0002: Directory names

A directory name conforms to the following constraints:

FORM	directory name = name name: no leading digits, no blanks
UNIQUENESS	all directory names within the parent project directory
ALLOWED CHARACTERS	name: a b c d e f g h i j k l m n o p q r s t u v w x y z A B C D E F G H I J K L M N O P Q R S T U V W X Y Z 0 1 2 3 4 5 6 7 8 9 _
UNDERSCORES	name: <ul style="list-style-type: none"><li>• underscores can be used to separate parts</li><li>• cannot have more than one consecutive underscore</li><li>• cannot start with an underscore</li><li>• cannot end with an underscore</li></ul>

#### 4.1.3. na\_0035: Adoption of naming conventions

Adoption of a naming convention is recommended. A naming convention provides guidance for naming blocks, signals, parameters and data types. Naming conventions frequently cover issues such as:

☐ Compliance with the programming language and downstream tools

o Length

o Use of symbols

☐ Readability

o Use of underscores

o Use of capitalization

☐ Encoding information

o Use of "meaningful" names

o Standard abbreviations and acronyms

o Data type

o Engineering units

o Data ownership

o Memory type

#### 4.2.1. jc\_0201: Usable characters for Subsystem name

#### 4.2.2. jc\_0211: Usable characters for Inport block and Outport block

#### 4.2.3. jc\_0221: Usable characters for signal line name

#### 4.2.4. na\_0030: Usable characters for Simulink Bus names

#### 4.2.5. jc\_0231: Usable characters for block names

The names of all Subsystem blocks should conform to the following constraints:

FORM	<b>name:</b> <ul style="list-style-type: none"> <li>• should not start with a number</li> <li>• should not have blank spaces</li> <li>• should not have carriage returns</li> </ul>
ALLOWED CHARACTERS	<b>name:</b> a b c d e f g h i j k l m n o p q r s t u v w x y z A B C D E F G H I J K L M N O P Q R S T U V W X Y Z 0 1 2 3 4 5 6 7 8 9 _
UNDERSCORES	<b>name:</b> <ul style="list-style-type: none"> <li>• underscores can be used to separate parts</li> <li>• cannot have more than one consecutive underscore</li> <li>• cannot start with an underscore</li> <li>• cannot end with an underscore</li> </ul>

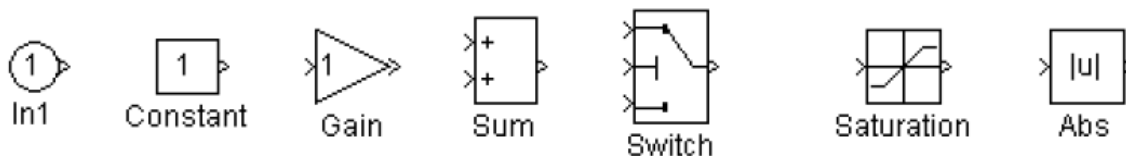
#### 4.2.6. na\_0014: Use of local language in Simulink and Stateflow

The local language should be used only in descriptive fields. Descriptive fields are text entry points that do not affect code generation or simulation.

#### Basic Blocks

This document uses the term “Basic Blocks” to refer to blocks from the base Simulink library.

Examples of basic blocks:



#### 5.1.1. na\_0006: Guidelines for mixed use of Simulink and Stateflow

The choice of whether to use Simulink or Stateflow to model a given portion of the control algorithm functionality should be driven by the nature of the behavior being modeled.

- ☐ If the function primarily involves complicated logical operations, use Stateflow diagrams.
- ☐ Stateflow should be used to implement modal logic – where the control function to be performed at the current time depends on a combination of *past and present logical conditions*.
- ☐ If the function primarily involves numerical operations, use Simulink features.

#### Specifics:

- ☐ If the primary nature of the function is logical, but some simple numerical calculations are done to support the logic, implement the simple numerical functions using the Stateflow action language.

- If the primary nature of the function is numeric, but some simple logical operations are done to support the arithmetic, implement the simple logical functions with Simulink blocks.

If the primary nature of the function is logical, and some complicated numerical calculations must be done to support the logic, use a Simulink subsystem to implement the numerical calculations. The Stateflow software should invoke the execution of this subsystem, using a function-call.

Use the Stateflow product to implement modal logic, where the control function to be performed at the current time depends on a combination of *past and present logical conditions*. (If there is a need to store the result of a logical

condition test in Simulink, for example, by storing a flag, this is one indicator of the presence of modal logic, which should be modeled with Stateflow software.)

### 5.1.2. na\_0007: Guidelines for use of Flow Charts, Truth Tables and State Machines

Within Stateflow, the choice of whether to use a flow chart or a state chart to model a given portion of the control algorithm functionality should be driven by the nature of the behavior being modeled.

- ☐ If the primary nature of the function segment is to calculate modes of operation or discrete-valued states, use state charts. Some examples are:
  - Diagnostic model with pass, fail, abort, and conflict states
  - Model that calculates different modes of operation for a control algorithm
- ☐ If the primary nature of the function segment involves if-then-else statements, use flowcharts or truth tables.

Specifics:

- ☐ If the primary nature of the function segment is to calculate modes or states, but if-then-else statements are required, add a flow chart to a state within the state chart. (See 7.5 Flowchart Patterns)
















### 5.2.1. db\_0143: Similar block types on the model levels

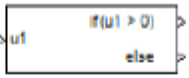





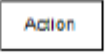
To allow partitioning of the model into discreet units, every level of a model must be designed with building blocks of the same type (i.e. only Subsystem or only basic blocks). The blocks listed in this rule are used for signal routing. You can place them at any level of the model.

**Blocks which can be placed on every model level:**

Inport



Output	
Mux	
Demux	
Bus Selector	
Bus Creator	
Selector	
Ground	
Terminator	
From	
Goto	
Merge	
Unit Delay	
Rate Transition	
Data Type Conversion	
Data Store Memory	

If	
Case	
Function-Call Generator	
Function-Call Split	
Trigger <sup>(1)</sup>	
Enable <sup>(2)</sup>	
Action port <sup>(3)</sup>	

### 5.2.2. db\_0144: Use of Subsystems

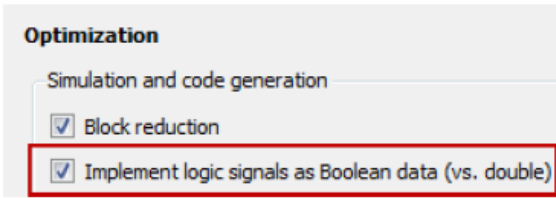
Blocks in a Simulink diagram should be grouped together into subsystems based on functional decomposition of the algorithm, or portion thereof, represented in the diagram.  
 Avoid grouping blocks into subsystems primarily for the purpose of saving space in the diagram. Each subsystem in the diagram should represent a unit of functionality required to accomplish the purpose of the model or submodel. Blocks can also be grouped together based on behavioral variants or timing.  
 If creation of a subsystem is required for readability issues, then a virtual subsystem should be used.

### 5.2.3. db\_0040: Model hierarchy

The model hierarchy should correspond to the functional structure of the control system.

### 6.1.1. jc\_0011: Optimization parameters for Boolean data types

The optimization option for Boolean data types must be enabled (on).

Path	Parameter	Image
Configuration Parameters > Optimization > Simulation and code generation > Implement logic signals as Boolean data (vs. double)	BooleanDataType	

### 6.1.2. jc\_0021: Model diagnostic settings

The following diagnostics must be enabled. An enabled diagnostic is set to either “warning” or “error”. Setting the diagnostic option to “none” is not permitted. Diagnostics that are not listed can be set to any value (none, warning, or error).

- ☐ Solver Diagnostics
- ☐ Algebraic loop
- ☐ Minimize algebraic loop
- ☐ Sample Time Diagnostics
- ☐ Multitask rate transition
- ☐ Data Validity Diagnostics
- ☐ Inf or NaN block output
- ☐ Duplicate data store names
- ☐ Connectivity
- ☐ Unconnected block input ports
- ☐ Unconnected block output ports
- ☐ Unconnected line
- ☐ Unspecified bus object at root Outport block
- ☐ Mux blocks used to create bus signals
- ☐ Invalid function-call connection
- ☐ Element name mismatch

#### 7.1.1.1. na\_0004: Simulink model appearance

View Options	Setting
Model Browser	unchecked
Screen color	white
Status Bar	checked
Toolbar	checked
Zoom factor	Normal (100%)
<b>Block Display Options</b>	<b>Setting</b>
Background Color	white
Foreground Color	black
Execution Context Indicator	unchecked
Library Link Display	none
Linearization Indicators	checked
Model/Block I/O Mismatch	unchecked
Model Block Version	unchecked
Sample Time Colors	unchecked
Sorted Order	unchecked
<b>Signal Display Options</b>	<b>Setting</b>
Port Data Types	unchecked
Signal Dimensions	unchecked
Storage Class	unchecked
Test point Indicators	checked
Viewer Indicators	checked
Wide Non-scalar Lines	checked



### 7.1.2. db\_0043: Simulink font and font size

All text elements (block names, block annotations and signal labels) except free text annotations within a model must have the same font style and font size. Fonts and font size should be selected for legibility.

Note: The selected font should be directly portable (e.g. Simulink/Stateflow default font) or convertible between platforms (e.g. Arial/Helvetica 12pt).

### 7.1.3. db\_0042: Port block in Simulink models

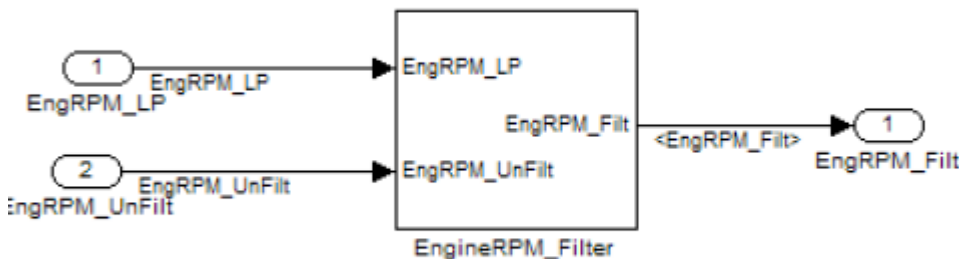
In a Simulink model, the ports comply with the following rules:

- ☐ Inports should be placed on the left side of the diagram, but they can be moved in to prevent signal crossings.
- ☐ Outports should be placed on the right side, but they can be moved in to prevent signal crossings.
- ☐ Duplicate Inports can be used at the subsystem level if required, but should be avoided, if possible.
- ☐ Do not use duplicate Inports at the root level.

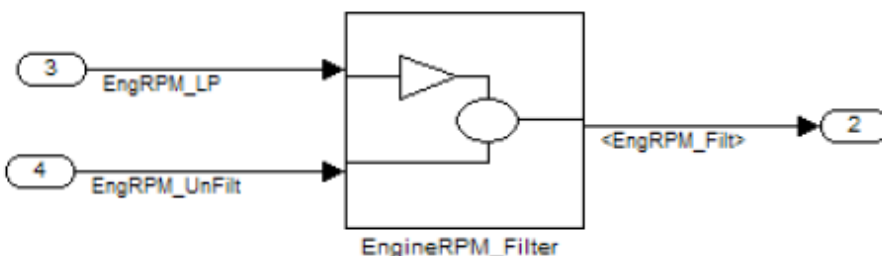
### 7.1.4. na\_0005: Port block name visibility in Simulink models

First option shall be used.

1. The name of an Inport or Outport is not hidden. ("Format / Hide Name" is not allowed.)



2. The name of an Inport or Outport must be hidden. ("Format / Hide Name" is used.)  
*Exception: inside library subsystem blocks, the names may not be hidden.*



**Correct: Use of signal label**



#### 7.1.5. jc\_0081: Icon display for Port block

The Icon display setting should be set to Port number for Inport and Outport blocks.

#### 7.1.6. jm\_0002: Block resizing

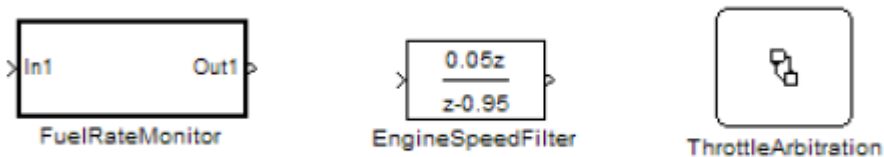
All blocks in a model must be sized such that their icon is completely visible and recognizable. In particular, any text displayed (for example, tunable parameters, filenames, or equations) in the icon must be readable. This guideline requires resizing of blocks with variable icons or blocks with a variable number of inputs and outputs. In some cases, it may not be practical or desirable to resize the block icon of a subsystem block so that all of the input and output names within it are readable. In such cases, you may hide the names in the icon by using a mask or by hiding the names in the subsystem associated with the icon. If you do this, the signal lines coming into and out of the subsystem block should be clearly labeled in close proximity to the block.

#### 7.1.7. db\_0142: Position of block names

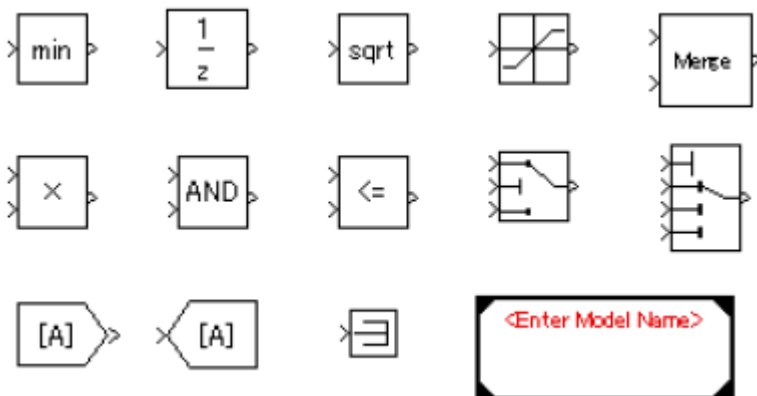
If shown the name of each block should be placed below the block.

#### 7.1.8. jc\_0061: Display of block names

- Display a block name when it provides descriptive information.



- The block name should not be displayed if the block function is known and understood from the block appearance.



#### 7.1.9. db\_0146: Triggered, enabled, conditional Subsystems

The blocks that define subsystems as either conditional or iterative should be located at a consistent location at the top of the subsystem diagram. These blocks are:

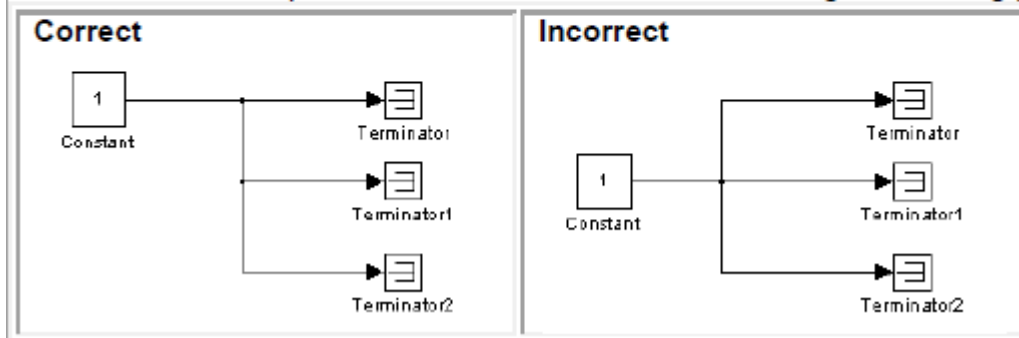
- ☐ Enable
- ☐ For Iterator
- ☐ Action Port
- ☐ Switch Case Action
- ☐ Trigger

□ While Iterator

7.1.11. db\_0032: Simulink signal appearance

Signal lines

- Should not cross each other, if possible.
- Are drawn with right angles.
- Are not drawn one upon the other.
- Do not cross any blocks.
- Should not split into more than two sub lines at a single branching point.

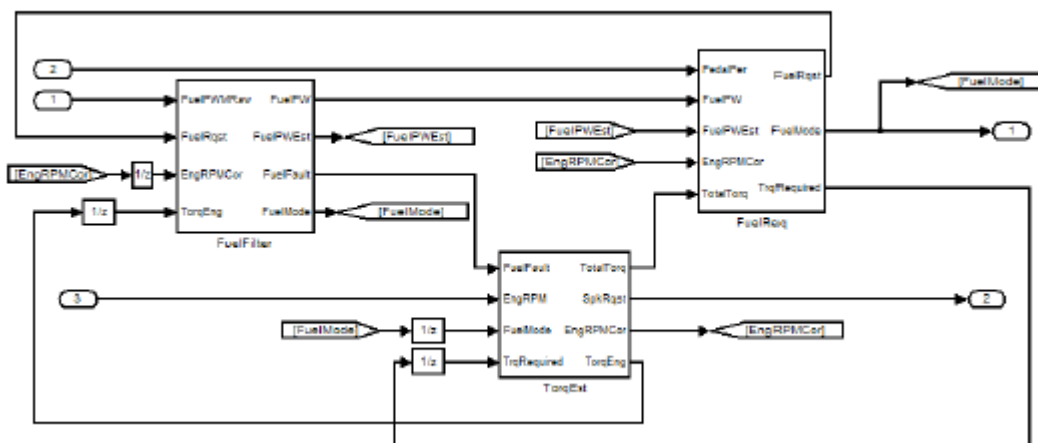


7.1.12. db\_0141: Signal flow in Simulink models

- Signal flow in a model is from left to right.
  - Exception: Feedback loops
- Sequential blocks or subsystems are arranged from left to right.
  - Exception: Feedback loops
- Parallel blocks or subsystems are arranged from top to bottom.

7.1.13. jc\_0171: Maintaining signal flow when using Goto and From blocks

Correct



7.1.14. na\_0032: Use of Merge Blocks

When using merge blocks:

- Signals entering a merge block must not branch off to any other block.
- With buses:
  - All buses must be identical. This includes:
    - Number of elements
    - Element names
    - Element order
    - Element data type
    - Element size
  - Buses must be either all virtual or all non-virtual.
  - All bus lines entering a merge block must not branch off to any other block.

#### 7.1.15. jm\_0010: Port block names in Simulink models

For some items, it is not possible to define a single approach applicable to all organizations' internal processes. However, within a given organization, it is important to follow a **single** consistent approach. An organization applying the guidelines must select **one** of these alternatives.

1. Names of Inport blocks and Outport blocks must match the corresponding signal or bus names. **Exceptions:**
  - When any combination of an Inport block, an Outport block, and any other block have the same block name, a suffix or prefix should be used on the Inport and Outport blocks.
  - One common suffix / prefix is “\_in” for Inports and “\_out” for Outports.
  - Any suffix or prefix can be used on the ports, however the selected prefix should be consistent.
  - Library blocks and reusable subsystems that encapsulate generic functionality.

2. When the names of Inport and Outport blocks are hidden, apply a consistent naming practice for the blocks. Suggested practices include leaving the names as their default names (for example, Out1), giving them the same name as the associated signal or giving them a shortened or mangled version of the name of the associated signal.

#### 7.2.1. na\_0008: Display of labels on signals

A label must be displayed on a signal originating from the following blocks:

- ☐ Inport block
- ☐ From block (block icon exception applies – see Note below)
- ☐ Subsystem block or Stateflow chart block (block icon exception applies)
- ☐ Bus Selector block (the tool forces this to happen)
- ☐ Demux block
- ☐ Selector block
  
- ☐ Data Store Read block (block icon exception applies)
- ☐ Constant block (block icon exception applies)

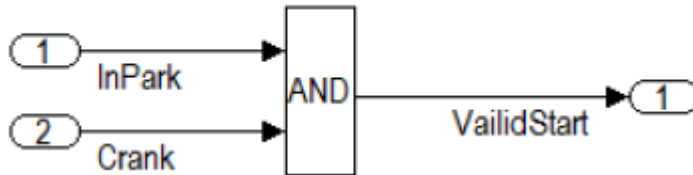
A label must be displayed on any signal connected to the following destination blocks (directly or by way of a basic block that performs a non transformative operation):

- ☐ Outport block
- ☐ Goto block
- ☐ Data Store Write block
- ☐ Bus Creator block
- ☐ Mux block
- ☐ Subsystem block

- ☐ Chart block

Note: Block icon exception (applicable only where called out above): If the signal label is visible in the originating block icon display, the connected signal does not need to have the label displayed, *unless* the signal label is needed elsewhere due to a destination-based rule.

### Correct



### 7.2.2. na\_0009: Entry versus propagation of signal labels

If a label is present on a signal, the following rules define whether that label shall be created there (entered directly on the signal) or propagated from its true source (inherited from elsewhere in the model by using the „<“ character).

- Any displayed signal label must be *entered* for signals that:
  - Originate from an Inport at the Root (top) Level of a model
  - Originate from a basic block that performs a transformative operation (For the purpose of interpreting this rule only, the Bus Creator block, Mux block, and Selector block shall be considered to be included among the blocks that perform transformative operations.)
- Any displayed signal label must be *propagated* for signals that:
  - Originate from an Inport block in a nested subsystem **Exception:** If the nested subsystem is a library subsystem, a label may be *entered* on the signal coming from the Inport to accommodate reuse of the library block.
  - Originate from a basic block that performs a non-transformative operation
  - Originate from a Subsystem or Stateflow chart block **Exception:** If the connection originates from the output of a library subsystem block instance, a new label may be *entered* on the signal to accommodate reuse of the library block.

### 7.2.4. db\_0081: Unconnected signals, block inputs and block outputs

A system must not have any:

- ☐ Unconnected subsystem or basic block input.
- ☐ Unconnected subsystem or basic block outputs
- ☐ Unconnected signal lines

In addition:

- ☐ An otherwise unconnected input should be connected to a ground block
- ☐ An otherwise unconnected output should be connected to a terminator block

### 7.3.1. na\_0003: Simple logical expressions in If Condition block

A logical expression may be implemented within an If Condition block instead of building it up with logical operation blocks, if the expression contains two or fewer primary expressions. A primary expression is defined here to be one of the following:

- ☐ An input
- ☐ A constant
- ☐ A constant parameter
- ☐ A parenthesized expression containing no operators except zero or one instances of the following operators: <, <=, >, >=, ~, ==, ~. (See for the following examples.)

**Exception:**

A logical expression may contain more than two primary expressions if both of the following are true:

- ☐ The primary expressions are all inputs
- ☐ Only one type of logical operator is present

Examples of Acceptable Exceptions:

- ☐ `u1 || u2 || u3 || u4 || u5`
- ☐ `u1 && u2 && u3 && u4`

Examples of Primary Expressions:

- ☐ `u1`
- ☐ `5`
- ☐ `K`
- ☐ `(u1 > 0)`
- ☐ `(u1 <= G)`
- ☐ `(u1 > U2)`
- ☐ `(~u1)`
- ☐ `(EngineState.ENGINE_RUNNING)`

Examples of Acceptable Logical Expressions:

- ☐ `u1 || u2`
- ☐ `(u1 > 0) && (u1 < 20)`
- ☐ `(u1 > 0) && (u2 < u3)`
- ☐ `(u1 > 0) && (~u2)`
- ☐ `(EngineState.ENGINE_RUNNING) && (PRNDLState.PRNDL_PARK)`

Note: In this example `EngineState.ENGINE_RUNNING` and `PRNDLState.PRNDL_PARK` are enumeration literals

Examples of unacceptable logical expressions include:

- ☐ `u1 && u2 || u3` (too many primary expressions)
- `u1 && (u2 || u3)` (unacceptable operator within primary expression)
- ☐ `(u1 > 0) && (u1 < 20) && (u2 > 5)` (too many primary expressions that are not inputs)
- ☐ `(u1 > 0) && ((2*u2) > 6)` (unacceptable operator within primary expression)

### 7.3.2. na\_0002: Appropriate implementation of fundamental logical and numerical operations

Blocks that are intended to perform numerical operations must not be used to perform logical operations.

### 7.3.3. jm\_0001: Prohibited Simulink standard blocks inside controllers

- Χοντρολ αλγοριτημ μονελσ μυστ βε δεσιγνεδ φρομ δισχυρετε βλοχκς.
- The MathWorks “Simulink Block Data Type Support” table provides a list of blocks that support production code generation.
  - Use blocks that are listed as “Code Generation Support”.
  - Do not use blocks that are listed as “Not recommended for production code” – see footnote 4 in the table.
- ☐ In addition to the blocks defined by the above rule, do not use the following blocks

**Sources are not allowed:**

Sine Wave



Pulse Generator



Random Number



Uniform Random Number



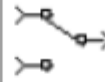
Band-Limited White Noise



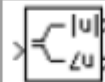
Slider Gain



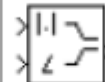
Manual Switch



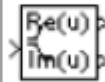
Complex to Magnitude-Angle



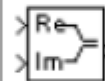
Magnitude-Angle to Complex



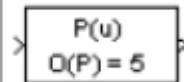
Complex to Real-Imag



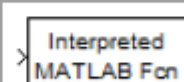
Real-Imag to Complex



Polynomial



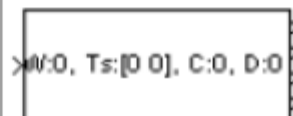
MATLAB Fcn<sup>(1)</sup>



Goto Tag Visibility



Probe



### 7.3.5. na\_0011: Scope of Goto and From blocks

For signal flows, the following rules apply:

- ☐ From and Goto blocks must use local scope.

Note: Control flow signals may use global scope.

Control flow signals are output from:

- ☐ Function-call generators
- ☐ If and Case blocks

### 7.3.6. jc\_0141: Use of the Switch block

- The switch condition, input 2, must be a Boolean value.
- ☐ The block parameter “Criteria for passing first input” should be set to  $u2 \neq 0$ .

### 7.3.7. jc\_0121: Use of the Sum block

- Use the “rectangular” shape.
  - Be sized so that the input signals do not overlap.
  - You may use the round shape in feedback loops.
    - There should be no more than 3 inputs.
    - The inputs may be positioned at 90,180,270 degrees.
    - The output should be positioned at 0 degrees.
- 

### 7.3.9. jc\_0161: Use of Data Store Read/Write/Memory blocks

- Prohibited in a data flow layer.
- Allowed between subsystems running at different rates.

### 7.4.1. db\_0112: Indexing

Use a consistent vector indexing method for all blocks.

When possible, use zero-based indexing to improve code efficiency. However, since MATLAB blocks do not support zero-based indexing, one-based indexing can be used for models containing MATLAB blocks.

### 7.4.2. na\_0010: Grouping data flows into signals

#### Vectors

The individual scalar signals composing a vector must have common functionality, data types, dimensions and units. The most common example of a vector signal is sensor or actuator data that is grouped into an array indexed by location. The output of a Mux block must always be a vector. The inputs to a Mux block must always be scalars.

#### Busses

Signals that do not meet criteria for us as a vector, as described above, must only be grouped into bus signals. Use Bus selector blocks may only be used with a bus signal input; they must not be used to extract scalar signals from vector signals.

### 7.4.3. db\_0110: Tunable parameters in basic blocks

To insure that a parameter is tunable, it must be entered in a block dialog field:

- ☐ Without any expression.
- ☐ Without a data type conversion.



- ☐ Without selection of rows or columns.

#### 7.5.1. na\_0012: Use of Switch vs. If-Then-Else Action Subsystem

The **Switch** block:

- ☐ Should be used for modeling simple *if-then-else* structures, if the associated *then* and *else* actions involve only the assignment of constant values.



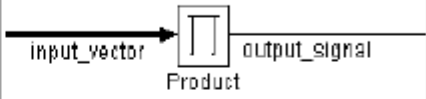
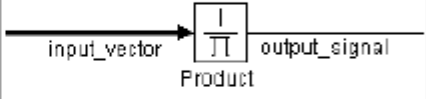
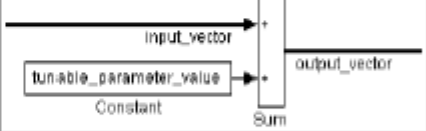
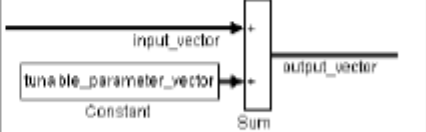
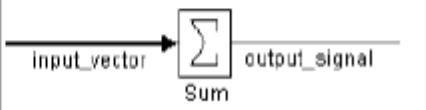
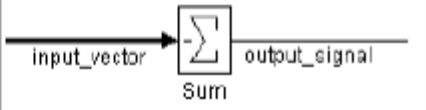
The **if-then-else action subsystem** construct:

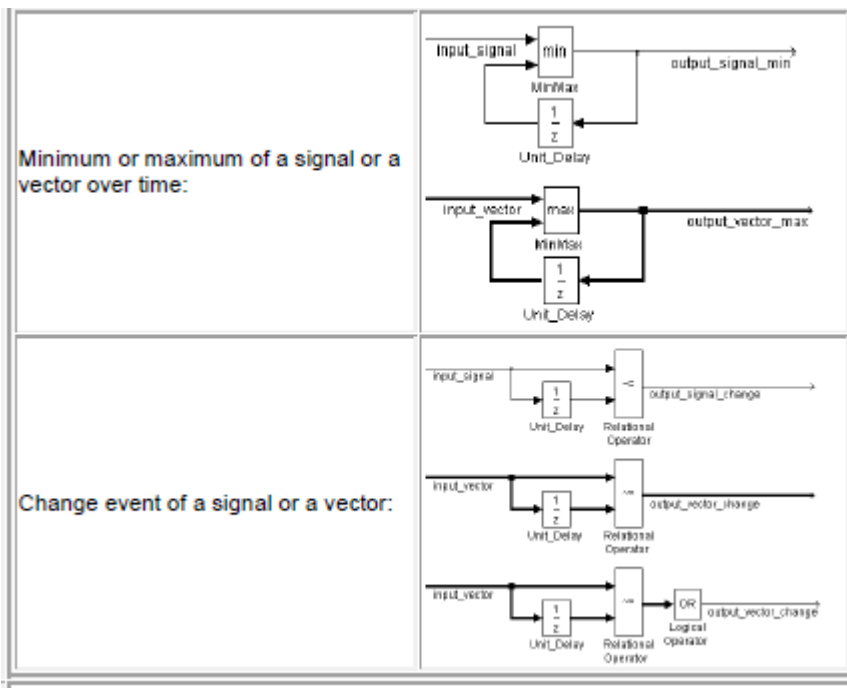
- ☐ Should be used for modeling *if-then-else structures*, if the associated *then* and/or *else* actions require complicated computations. This will maximize simulation efficiency and the efficiency of generated code (Note that even a basic block, for example a table look-up, may require fairly complicated computations.)

#### 7.5.4. na\_0028: Use of If-Then-Else Action Subsystem to Replace Multiple Switches

The use of switch constructs should be limited, typically to 3 levels. Replace switch constructs that have more than 3 levels with an If-Then-Else action subsystem construct.

#### 7.5.6. db\_0117: Simulink patterns for vector signals

Vector loop: for (i=0; i<input_vector_size; i++) { output_vector(i) = input_vector(i) * tunable_parameter_value; } 	
Vector loop: for (i=0; i<input_vector_size; i++) { output_vector(i) = input_vector(i) * tunable_parameter_vector(i); } 	
Vector loop: output_signal = 1; for (i=0; i<input_vector_size; i++) { output_signal = output_signal * input_vector(i); } 	
Vector loop: output_signal = 1; for (i=0; i<input_vector_size; i++) { output_signal = output_signal / input_vector(i); } 	
Vector loop: for (i=0; i<input_vector_size; i++) { output_vector(i) = input_vector(i) + tunable_parameter_value; } 	
Vector loop: for (i=0; i<input_vector_size; i++) { output_vector(i) = input_vector(i) + tunable_parameter_vector(i); } 	
Vector loop: output_signal = 0; for (i=0; i<input_vector_size; i++) { output_signal = output_signal + input_vector(i); } 	
Vector loop: output_signal = 0; for (i=0; i<input_vector_size; i++) { output_signal = output_signal - input_vector(i); } 	



#### 7.5.8. jc\_0111: Direction of Subsystem

Subsystem must not be reversed.

#### 8.1.1. db\_0123: Stateflow port names

The name of a Stateflow input/output should be the same as the corresponding signal.  
Exception: Reusable Stateflow blocks may have different port names.

#### 8.1.2. db\_0129: Stateflow transition appearance

Transitions in Stateflow:

- ☐ Do not cross each other, if possible.
- ☐ Are not drawn one upon the other.
- ☐ Do not cross any states, junctions or text fields.
- ☐ Allowed, if transitioning to an internal state.

Transition labels can be visually associated to the corresponding transition.

#### 8.1.3. db\_0137: States in state machines

For all levels in a state machine, including the root level, for states with exclusive decomposition, the following rules apply:

- ☐ At least two exclusive states must exist.
- ☐ A state cannot have only one substate.
- ☐ The initial state of every hierarchical level with exclusive states is clearly defined by a default transition. In the case of multiple default transitions, there must always be an unconditional default transition.

#### 8.1.5. db\_0132: Transitions in Flowcharts

The following rules apply to transitions in Flowcharts:

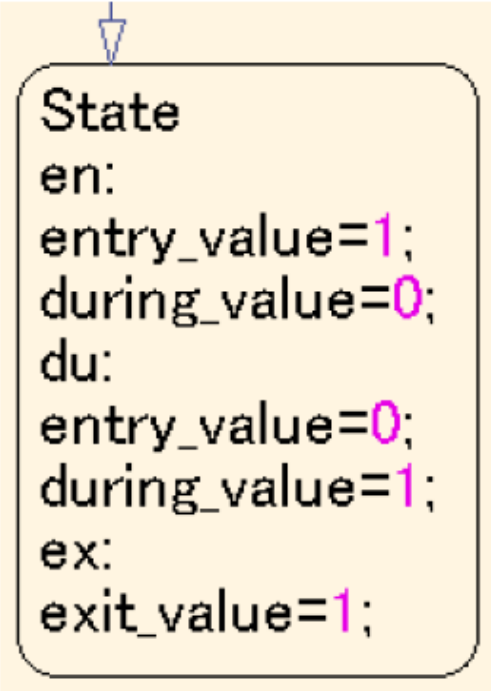
- ☐ Conditions are drawn on the horizontal.
- ☐ Actions are drawn on the vertical.
- ☐ Loop constructs are intentional exceptions to this rule.
- ☐ Transitions have a condition, a condition action, or an empty transition.

#### 8.1.6. jc\_0501: Format of entries in a State block

A new line should:

- Start after the entry (en) during (du), and exit (ex) statements.
- Start after the completion of an assignment statement “;”.

Correct



```
State
en:
entry_value=1;
during_value=0;
du:
entry_value=0;
during_value=1;
ex:
exit_value=1;
```

#### 8.1.7. jc\_0511: Setting the return value from a graphical function

The return value from a graphical function must be set in only one place.

#### 8.1.9. jc\_0521: Use of the return value from graphical functions

The return value from a graphical function should not be used directly in a comparison operation.

#### 8.2.1. na\_0001: Bitwise Stateflow operators

The bitwise Stateflow operators (&, |, and ^) should not be used in Stateflow charts unless you want bitwise operations.

To enable bitwise operations:

1. Select File > Chart Properties
2. Select “Enable C-bit Operations”.

### 8.2.2. jc\_0451: Use of unary minus on unsigned integers in Stateflow

Do not perform unary minus on unsigned integers.

### 8.2.3. na\_0013: Comparison operation in Stateflow

- Comparisons should be made only between variables of the same data type.
- If comparisons are made between variables of different data types, the variables need to be explicitly type cast to matching data types.

Do not make comparisons between unsigned integers and negative numbers.

### 8.2.4. db\_0122: Stateflow and Simulink interface signals and parameters

All charts should use strong data typing with Simulink (The option "Use Strong Data Typing with Simulink I/O" must be selected).

### 8.2.5. db\_0125: Scope of internal signals and local auxiliary variables

Internal signals and local auxiliary variables are "Local data" in Stateflow:

- ☐ All local data of a Stateflow block must be defined on the chart level or below the Object Hierarchy.
- ☐ No local variables exist on the machine level (that is, there is no interaction between local data in different charts).
- ☐ Parameters and constants are allowed at the machine level.

### 8.2.6. jc\_0481: Use of hard equality comparisons for floating point numbers in Stateflow

Do not use hard equality comparisons ( $\text{Var1} == \text{Var2}$ ) with two floating point numbers.

- ☐ If a hard comparison is required, a margin of error should be defined and used in the comparison (LIMIT in the example).
- ☐ Hard equality comparisons may be done between two integer data types.

### 8.2.7. jc\_0491: Reuse of variables within a single Stateflow scope

The same variable should not have multiple meanings (usages) within a single Stateflow state.

### 8.2.8. jc\_0541: Use of tunable parameters in Stateflow

Create tunable parameters in Stateflow charts in one of the following ways:

- 1.) Define the parameters in the Stateflow chart and corresponding parameters in the base workspace
- 2.) Include the tunable parameters as an input into the Stateflow chart. The parameters must be defined in the base workspace.

### 8.2.9. db\_0127: MATLAB commands in Stateflow

Individual companies should decide on the use of MATLAB functions. If they are permitted, then MATLAB functions should only be accessed through the MATLAB function block.

### 8.2.10. jm\_0011: Pointers in Stateflow

In a Stateflow diagram, pointers to custom code variables are not allowed.

#### **8.3.1. db\_0126: Scope of events**

The following rules apply to events in Stateflow:

- ☐ All events of a Chart must be defined on the chart level or lower.
- ☐ There is no event on the machine level (that is, there is no interaction with local events between different charts).


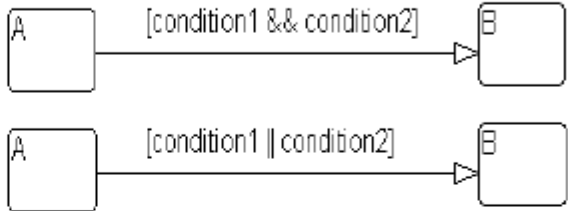
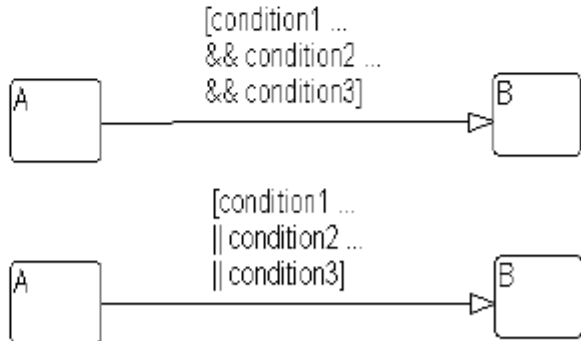
#### **8.3.2. jm\_0012: Event broadcasts**

The following rules apply to event broadcasts in Stateflow:

- ☐ Directed event broadcasts are the only type of event broadcasts allowed.
- ☐ The send syntax or qualified event names are used to direct the event to a particular state.
- ☐ Multiple send statements should be used to direct an event to more than one state.

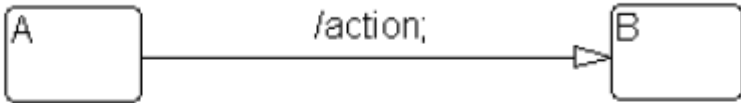
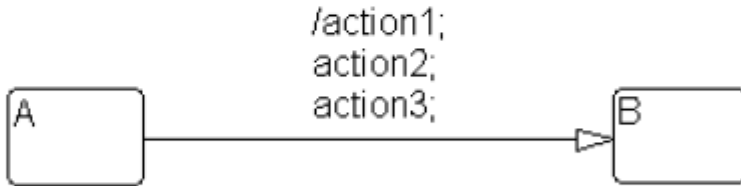
#### **8.4.1. db\_0150: State machine patterns for conditions**

The following patterns are used for conditions within Stateflow state machines:

Equivalent Functionality	State Machine Pattern
<p>ONE CONDITION:</p> <p><i>(condition)</i></p>	 <pre> graph LR     A[A] -- "[condition]" --&gt; B[B] </pre>
<p>UP TO THREE CONDITIONS, SHORT FORM:</p> <p>(The use of different logical operators in this form is not allowed, use sub conditions instead)</p> <p><i>(condition1 &amp;&amp; condition2)</i></p> <p><i>(condition1    condition2)</i></p>	 <pre> graph LR     A1[A] -- "[condition1 &amp;&amp; condition2]" --&gt; B1[B]     A2[A] -- "[condition1    condition2]" --&gt; B2[B] </pre>
<p>TWO OR MORE CONDITIONS, MULTILINE FORM:</p> <p>A sub condition is a set of logical operations, all of the same type, enclosed in parentheses.</p> <p>(The use of different operators in this form is not allowed, use sub conditions instead.)</p> <p><i>(condition1 ... &amp;&amp; condition2 ... &amp;&amp; condition3)</i></p> <p><i>(condition1 ...    condition2 ...    condition3)</i></p>	 <pre> graph LR     A3[A] -- "[condition1 ...&amp;&amp; condition2 ...&amp;&amp; condition3]" --&gt; B3[B]     A4[A] -- "[condition1 ...   condition2 ...   condition3]" --&gt; B4[B] </pre>

#### 8.4.2. db\_0151: State machine patterns for transition actions

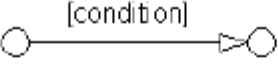

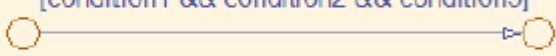
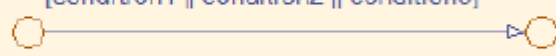
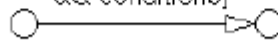
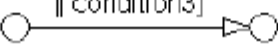
The following patterns are used for transition actions within Stateflow state machines:

Equivalent Functionality	State Machine Pattern
<p>ONE TRANSITION ACTION:</p> <p><i>action;</i></p>	 <pre> graph LR     A[A] -- "/action;" --&gt; B[B] </pre>
<p>TWO OR MORE TRANSITION ACTIONS, MULTILINE FORM: (Two or more transition actions in one line are not allowed.)</p> <p><i>action1;</i> <i>action2;</i> <i>action3;</i></p>	 <pre> graph LR     A[A] -- "/action1; action2; action3;" --&gt; B[B] </pre>
<div> <input checked="" type="checkbox"/> Readability             <input checked="" type="checkbox"/> Verification and Validation         </div> <div> <input checked="" type="checkbox"/> Workflow             <input checked="" type="checkbox"/> Code Generation         </div> <div> <input checked="" type="checkbox"/> Simulation         </div>	

#### 8.5.1. db\_0148: Flowchart patterns for conditions

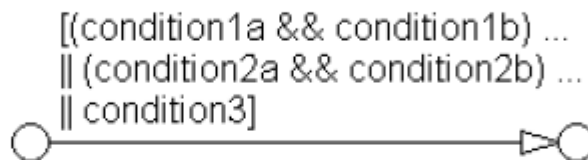
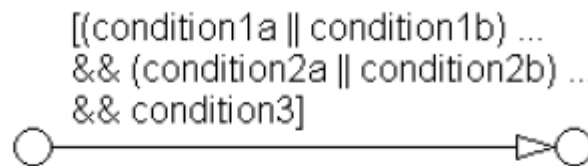


The following patterns are used for conditions within Stateflow Flowcharts:

Equivalent Functionality	Flowchart Pattern
<p>ONE CONDITION:</p> <p><i>[condition]</i></p>	 <pre>graph LR; S(( )) -- "[condition]" --&gt; T(( ))</pre> <p><i>/* comment */</i></p>  <pre>graph LR; S(( )) -- "/* comment */ [condition]" --&gt; T(( ))</pre>
<p>UP TO THREE CONDITIONS, SHORT FORM: (The use of different logical operators in this form is not allowed. Use sub conditions instead.)</p> <p><i>[condition1 &amp;&amp; condition2 &amp;&amp; condition3]</i></p> <p><i>[condition1 &amp;&amp; condition2 &amp;&amp; condition3]</i></p> <p><i>[condition1    condition2    condition3]</i></p> <p><i>[condition1    condition2    condition3]</i></p>	 <pre>graph LR; S(( )) -- "[condition1 &amp;&amp; condition2 &amp;&amp; condition3]" --&gt; T(( ))</pre>  <pre>graph LR; S(( )) -- "[condition1    condition2    condition3]" --&gt; T(( ))</pre>
<p>TWO OR MORE CONDITIONS, MULTILINE FORM: (The use of different logical operators in this form is not allowed. Use sub conditions instead.)</p> <p><i>[condition1 ... &amp;&amp; condition2 ... &amp;&amp; condition3]</i></p> <p><i>[condition1 ...    condition2 ...    condition3]</i></p> <p><i>[condition1 ... &amp;&amp; condition2 ... &amp;&amp; condition3]</i></p> <p><i>[condition1 ...    condition2 ...    condition3]</i></p> <p><i>[condition1 ...    condition2 ...    condition3]</i></p> <p><i>[condition1 ...    condition2 ...    condition3]</i></p>	 <pre>graph LR; S(( )) -- "[condition1 ... &amp;&amp; condition2 ... &amp;&amp; condition3]" --&gt; T(( ))</pre>  <pre>graph LR; S(( )) -- "[condition1 ...    condition2 ...    condition3]" --&gt; T(( ))</pre>

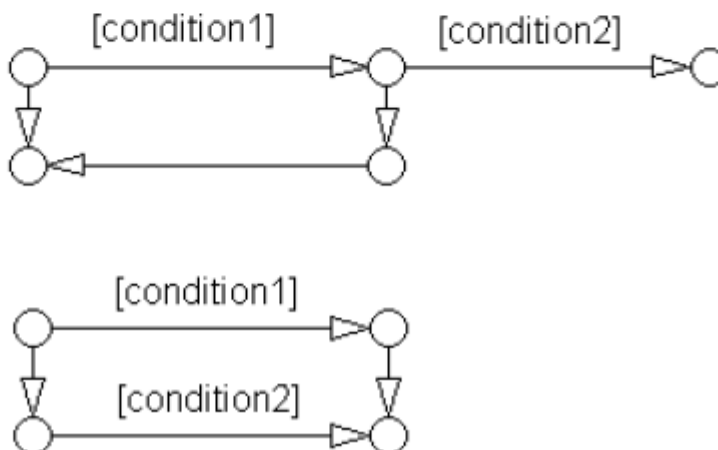
**CONDITIONS WITH SUBCONDITIONS:**  
(The use of different logical operators to connect sub conditions is not allowed. The use of brackets is mandatory.)



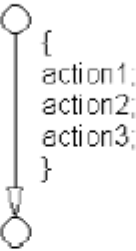

*[(condition1a || condition1b) ...  
&& (condition2a || condition2b) ...  
&& (condition3)]*  
*[(condition1a && condition1b) ...  
|| (condition2a && condition2b) ...  
|| (condition3)]*



**CONDITIONS THAT ARE VISUALLY SEPARATED:**  
(This form can be combined with the preceding patterns.)

*[condition1 && condition2]*  
*[condition1 || condition2]*



<p><b>ONE CONDITION ACTION:</b> action;</p>	 
<p><b>TWO OR MORE CONDITION ACTIONS, MULTILINE FORM:</b> (Two or more condition actions in one line are not allowed.) action1; ... action2; ... action3; ...</p>	
<p><b>CONDITION ACTIONS, WHICH ARE VISUALLY SEPARATED:</b> (This form can be combined with the preceding patterns.) action1a; action1b; action2; action3;</p>	

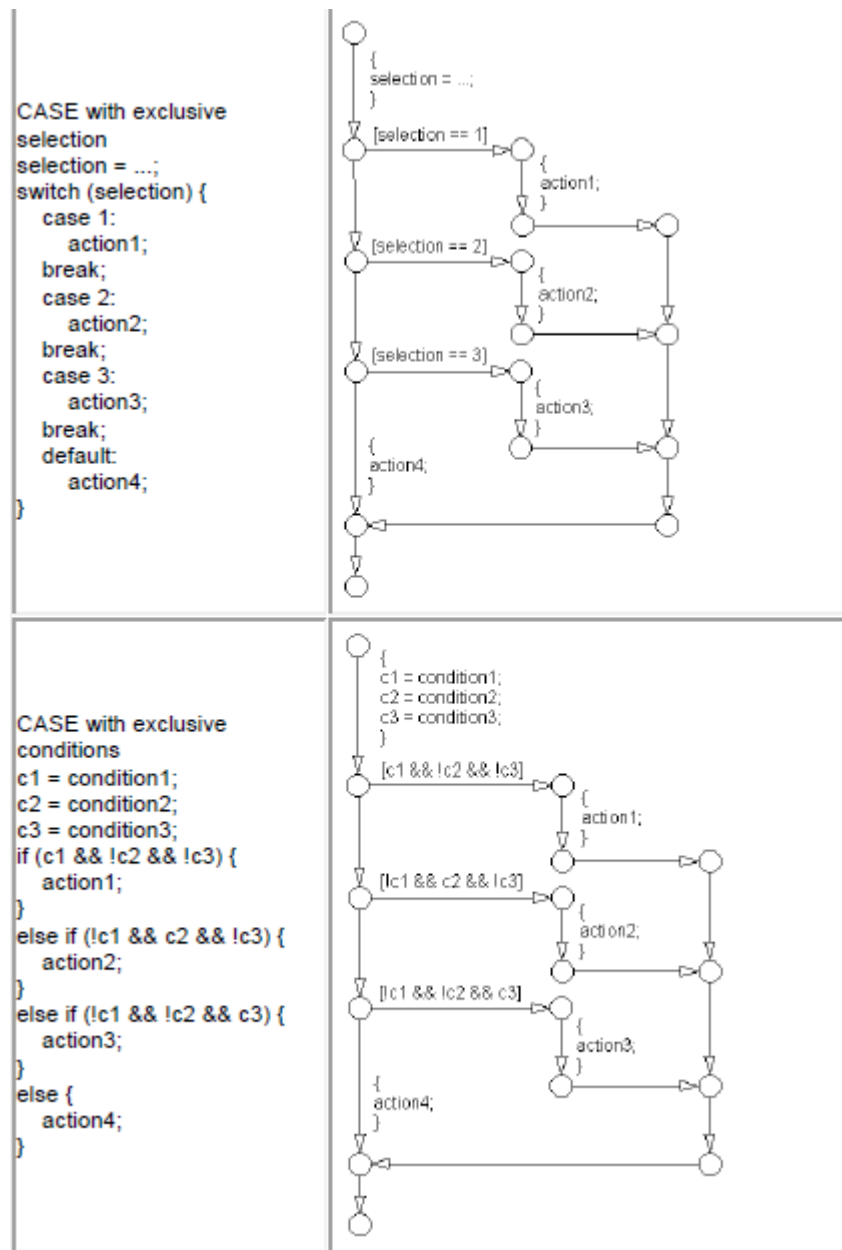
8.5.3. db\_0134: Flowchart patterns for If constructs

The following patterns are used for If constructs within Stateflow Flowcharts:

Equivalent Functionality	Flowchart Pattern
<b>IF THEN</b> if (condition){ action; }	
<b>IF THEN ELSE</b> if (condition) { action1; } else { action2; }	
<b>IF THEN ELSE IF</b> if (condition1) { action1; } else if (condition2) { action2; } else if (condition3) { action3; } else { action4; }	

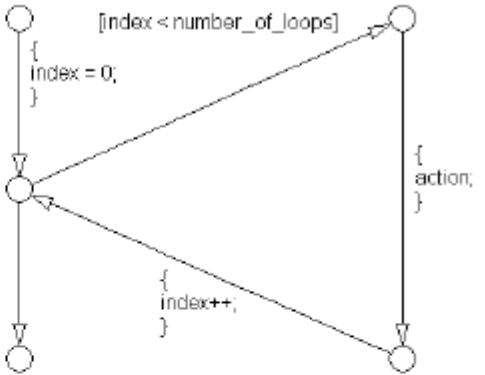
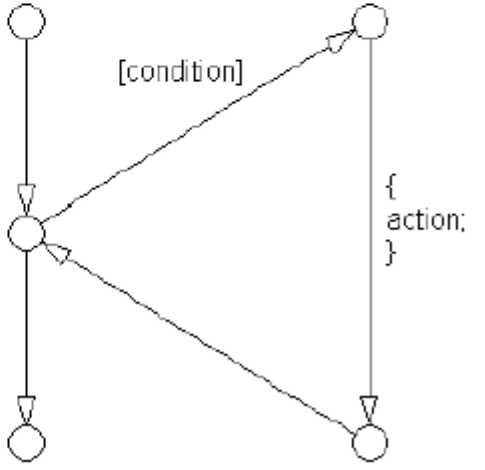
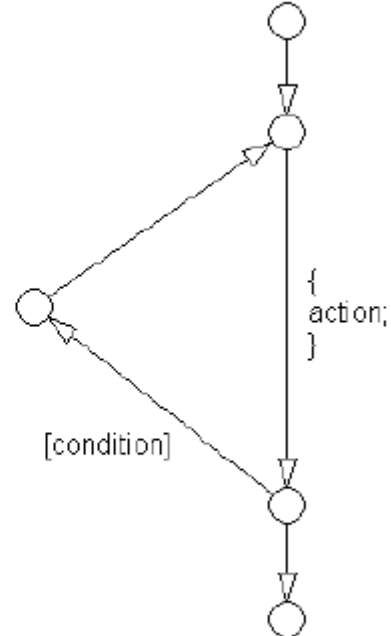
<b>Cascade of IF THEN</b> if (condition1) { action1; if (condition2) { action2; if (condition3) { action3; } } }	
---	--

#### 8.5.4. db\_0159: Flowchart patterns for case constructs



#### 8.5.5. db\_0135: Flowchart patterns for loop constructs

The following patterns must be used to create Loops within Stateflow Flowcharts:

Equivalent Functionality	Flowchart Pattern
<p><b>FOR LOOP</b>  for  (index=0;index&lt;number_of_loops;index++)  {    action;  }</p>	
<p><b>WHILE LOOP</b>  while (condition) {    action;  }</p>	
<p><b>DO WHILE LOOP</b>  do {    action;  }  while (condition);</p>	

#### 8.6.1. na\_0038: Levels in Stateflow charts

The number of nested States should be limited, typically 3 per level. If additional levels are required, use sub-charts.

#### 8.6.3. na\_0040: Number of states per container

The number of viewable States per container should be limited, typically to 6 to 10 states per container. The number is based on the visible states in the diagram.

#### 8.6.4. na\_0041: Selection of function type

Stateflow supports three types of functions: Graphical, MATLAB and Simulink. The appropriate function depends on the type of operations required:

- ☐ Simulink
  - ☐ Transfer functions
  - ☐ Integrators
  - ☐ Table look-ups
- ☐ MATLAB
  - ☐ Complex equations
  - ☐ If / then / else logic
- ☐ Graphical functions
  - ☐ If / then / else logic

#### 8.6.5. na\_0042: Location of Simulink functions

When deciding whether to embed Simulink functions inside a Stateflow chart, the following conditions make embedding the preferred option. If the Simulink functions:

- ☐ Use only local Chart data or
- ☐ Use a mixture of local Chart data and inputs from Simulink or
- ☐ Are called from multiple locations within the chart or
- ☐ Are not called every time step

### 10.1.2. na\_0019: Restricted Variable Names

To improve the readability of the MATLAB code, avoid using reserved C variable names. For example, avoid using const, TRUE, FALSE, infinity, nil, double, single, or enum in MATLAB Function code. These names may conflict with the compiler after C code is generated from the MATLAB code.

Avoid using variable names that conflict with MATLAB Functions, for example "conv".

Reserved key words are defined in Simulink Coder > User's Guide > Code Generation > Configuration > Code Appearance.

#### 10.1.3. na\_0025: MATLAB Function Header

MATLAB Functions must have a descriptive header. Header content may include, but is not limited to, the following types of information:

- ☐ Function name
- ☐ Description of function
- ☐ Assumptions and Limitations
- ☐ Description of changes from previous versions
- ☐ Lists of inputs and outputs

#### 10.2.1. na\_0034: MATLAB Function block input/output settings

All inputs and outputs to MATLAB Function blocks should have the data type explicitly defined, either in the Model Explorer or at the start of the function. This provides a more rigorous data type check for MATLAB Function blocks and prevents the need for using assert statements.

#### 10.2.2. na\_0024: Global Variables

The preferred method for accessing common data is with signal lines. However, if required, Data Store Memory can be used to emulate global memory.

**Example:**

In this example, the same Data Store Memory (ErrorFlag\_DataStore) is written to two separate MATLAB Functions.

#### 10.3.1. na\_0022: Recommended patterns for Switch / Case statements

Switch / Case statements must use constant values for the “Case” arguments. Input variables cannot be used in the “Case” arguments

#### 10.4.1. na\_0016: Source lines of MATLAB Functions

The length of MATLAB functions should be limited, with a recommended limit of 60 lines of code. This restriction applies to MATLAB Functions that reside in the Simulink block diagram and external MATLAB files with a .m extension.

If sub-functions are used, they may use additional lines of code. Also limit the length of sub-functions to 60 lines of code.

#### 10.4.2. na\_0017: Number of called function levels

The number of levels of sub-functions should be limited, typically to 3 levels. MATLAB function blocks that resides at the Simulink block diagram level counts as the first level, unless it is simply a wrapper for an external MATLAB file with a .m extension.

This includes functions that are defined within the MATLAB block and those in separate .m files.

Standard utility functions, such as built in functions like sqrt or log, are not included in the number of levels.

Likewise, commonly used custom utility functions can be excluded from the number of levels.

#### 10.4.3. na\_0021: Strings

The use of strings is not recommended. MATLAB Functions store strings as character arrays. The arrays cannot be resized to accommodate a string value of different length, due to lack of dynamic memory allocation. Strings are not a supported data type in Simulink, so MATLAB Function blocks cannot pass the string data outside the block.