View Editor

Overview

View Editor is the feature set in SceneComposer for designing and managing **View**s. You can compose screen-level Views by combining Scenes and existing Views, structure their transitions as a tree, and export the result as a Global State Machine asset. In addition, you can visually define event logic and screen transitions for each View as **Action Chains** and reuse them as conditional processing based on input devices and the solution state by using the **ConditionOutputs** defined in ConditionBlocks created in the **Condition Editor**.

This page provides the View Editor feature set used to work with them. The View Editor consists of the following four panels.

> For details on Views, see the View overview page.

View Definition Editor	An editor for defining screen-level Views by combining Scenes and existing Views, and for managing screen composition information. Views defined here are referenced as the common screen unit from the **View Tree**, the state machine, and the **Action Chain Editor**.
View Tree	An editor for arranging Views in the solution as a tree and designing screen transitions based on that structure. The structure you configure is exported as a Global State Machine and used as the candidate targets for the **JumpToView** Action behavior configured in the **Action Chain Editor**.
Action Chain Editor	This editor lets you visually define event processing and screen transitions for a View as trigger-based flowcharts. For each View selected in the **View Tree**, you associate an Action Chain and use **ConditionOutputs** created in the **Condition Editor** as branch conditions.
Condition Editor	The Condition Editor is an editor for visually defining conditions that are evaluated in Action Chains or assigned to the Condition properties of behaviors/Fusion as **ConditionOutputs** inside ConditionBlocks. The created ConditionOutputs can be reused as branch conditions in the Action Chain Editor and referenced from multiple locations in the solution as shared conditional logic.

Used together, these let you define View-based screen composition and transition logic in a consistent, unified way.

View

View sits above Scene in the hierarchy: you can treat a single Scene as one screen, or group multiple Scenes and/or existing Views into a single screen state. Each View is a logical unit that specifies what content to display and how to display it on a given screen. On the engine side, Views are identified by the dedicated identifier type `ViewIdentifier`.

A View is authored primarily in the **View Definition Editor**, and the result is referenced as a common screen-level object from the **View Tree**, the **State Machine**, and the **Action Chain Editor**. Screen showing and hiding is handled automatically as part of internal processing, so users do not need to configure display-control elements directly. In the View Tree, you define the transition structure between multiple Views as a tree, and this information is exported as a **Global State Machine** asset. Furthermore, by combining the **Action Chain Editor** and the **Condition Editor**, you can design view-level screen transitions and event logic that respond to input devices and the solution state.

By introducing Views, you can keep Scene-level layout definitions separate from screen-level transition/control logic, while organizing and maintaining the solution's overall screen composition consistently at the View level.

Create a View (from a Scene)

When you create a View starting from a Scene, that Scene is automatically associated with the newly created View. You can use it immediately as the View that displays that Scene.

1. In **Solution Explorer**, select any **Scene**, then choose **Create View...** from the context menu.
   - The **Add New View** dialog opens.
2. Configure the required settings and click **OK**.
   - The value set in **Name** is used as the View's display name in **Solution Explorer**.
   - The same value is also assigned automatically as the **View Identifier**.
3. A new **View** is created in **Solution Explorer** with the selected **Scene** as its child.

This operation automatically establishes the association between the View and the Scene.

Create a View (New Item)

If you create an empty View, you must later add and associate Scenes in the **View Definition Editor**. This method lets you flexibly build composite Views that combine multiple Scenes.

1. In **Solution Explorer**, open the context menu at the desired location and choose **Add > New Item**.
   - The **Add New View** dialog opens.
2. From the left menu, select **View**.
   - The View settings appear on the right.
3. Configure the required settings and click **OK**.
   - The value set in **Name** is used as the View's display name in **Solution Explorer**.
   - The same value is also assigned automatically as the **View Identifier**.
4. A new **View** is created in **Solution Explorer**.

Because a View created by this method has no Scene associated yet, you need to add the Scene(s) you want to display later in the **View Definition Editor**.

View Definition Editor

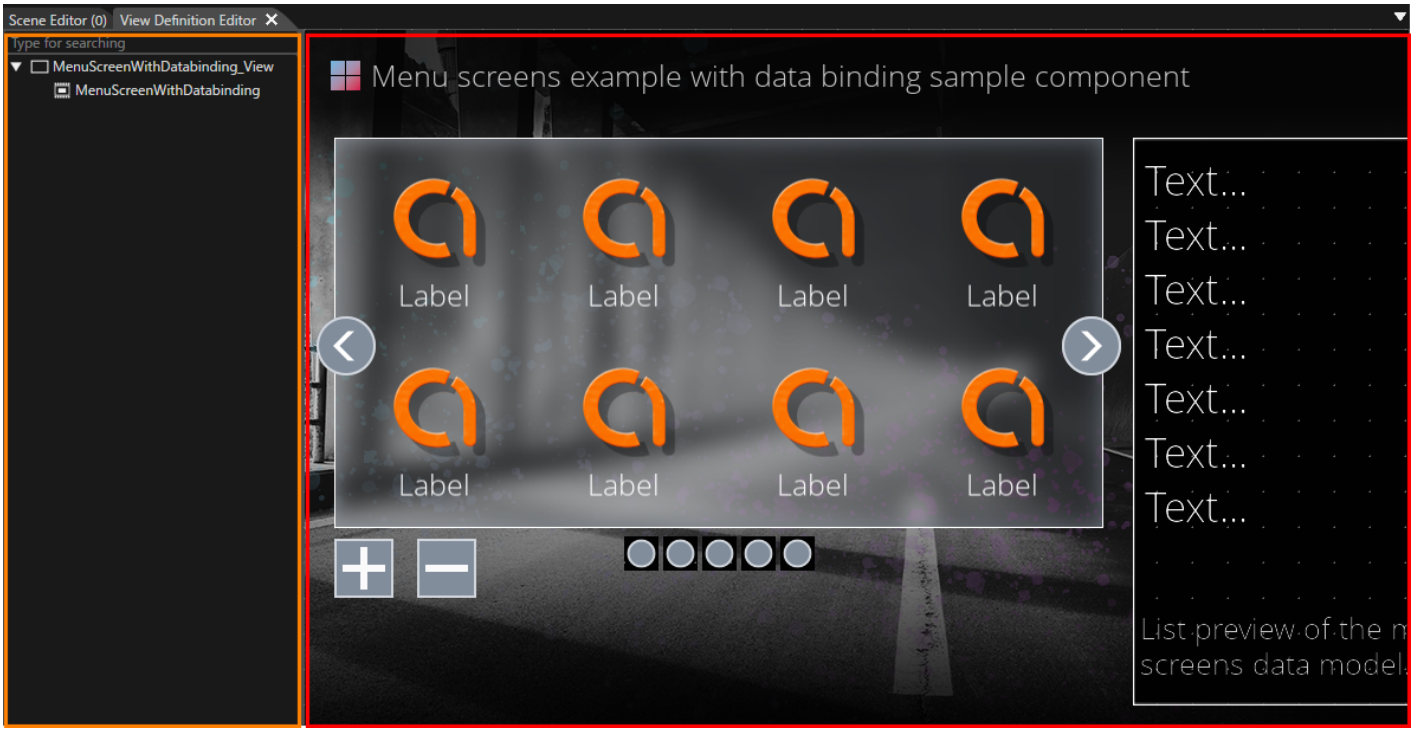
View Definition Editor lets you define “screen-level Views” by combining Scenes created in SceneComposer with existing **Views**, and make those definitions available to the **View Tree**, the **State Machine**, and the **Action Chain Editor**.

Only **Views** are displayed as nodes in the tree of the **View Definition Editor** panel. Therefore, if you want to use a **Scene**, you must always include it as a component of one of the **Views**. For details, see **View composition** later in this document. With this structure, the state machine can consistently treat “which screen (View) to activate” at the View level, while still incorporating Scene-based content into the screen composition.

View Definition Editor UI

Panel layout

Double-click any **View** in **Solution Explorer** to open the **View Definition Editor** panel.



UI item	Description
View Composition pane (orange frame in the figure above)	Lists which **Scenes/Views** the selected **View** is composed of. Add or remove Scenes and existing Views by **drag and drop**.

UI item	Description
Preview area (red frame in the figure above)	Displays preview images of the **Scenes/Views** associated with the View in the right pane, where you can also adjust their layout positions. Useful for improving visual identification when referencing this View from the **State Machine** or **View Tree**.

Basic operations

This section outlines common procedures for defining and editing Views with the **View Definition Editor**.

> For instructions on creating a **View**, see the View feature overview page.

View composition

In the **View Definition Editor** panel, you can use the tree structure to edit or delete child Views under the currently opened View. You can also review the entire tree rooted at that View on the tree, including the hierarchical relationships of its child Views and their descendant Views.

> The composition editing described here applies to Views that were created as **new Views**.
>
> Views that were automatically generated from a Scene are treated as simple Views that display the corresponding Scene, and you cannot edit the tree structure in the View Definition Editor or attach child Views or additional Scenes under them.

1. Double-click the View in Solution Explorer whose composition you want to edit.
   - The View Definition Editor panel opens and displays the composition of the selected View.
2. In the View Composition pane, edit the elements included in the View.
   - If the target View does not yet have a Scene as a child element, you can add a Scene or an existing View by dragging and dropping it from Solution Explorer.
   - If the target View already has a *Scene* as a child element, you cannot add any additional Scenes or Views (a View that contains a Scene as a child is always treated as a leaf node).
   - To remove unnecessary elements (Scenes/Views), select them and choose **Delete** from the context menu.

Behavior when dragging and dropping a Scene

Only **Views** are shown as nodes in the tree of the View Definition Editor panel. If you want to work with a **Scene**, you add it as internal content of an existing View.

- When you drag and drop a Scene from Solution Explorer into the **Composition** pane of the View currently open in the View Definition Editor, the **Create View** dialog appears if there is no existing View that already has that Scene as its child.
- If you accept the creation in the dialog, a new View that contains the Scene as its child is automatically generated and added to the tree as a child View of the drop target View.

With this mechanism, the View Definition Editor tree can consistently treat the unit of screen transitions and visibility control as **Views**, while still allowing you to build Scene-based content within those Views.

Grandchild View

In the View Definition Editor panel, a single operation can add Views only up to **the child View directly under the currently open View** (one hierarchy level down). To create a structure with three or more hierarchy levels, such as **Parent View > Child View > Grandchild View**, you must use only Views that meet the following requirements as the starting point:

- The View you want to use as the parent level must be a **newly created View**.
- The View you want to use as the parent level must **not** already have a **Scene** as its child element.
- Views that were automatically generated from a Scene, or Views that already have a **Scene** as a child, **cannot** be used as the starting point for building a grandchild hierarchy (these Views are always treated as **leaf nodes**).

Using a View that meets these requirements as the starting point, create a three-level structure (**Parent View > Child View > Grandchild View**) as follows:

1. Open the View you want to use as the parent level, and add a child View under it (**Parent View > Child View**).
2. Double-click the View you added as the child View in step 1 to open it, and then add another View under it as its child (**Child View > Grandchild View**).

After performing these steps, reopen the original parent View. You can then confirm in the tree that a three-level hierarchy (**Parent View > Child View > Grandchild View**) is displayed.

View Tree

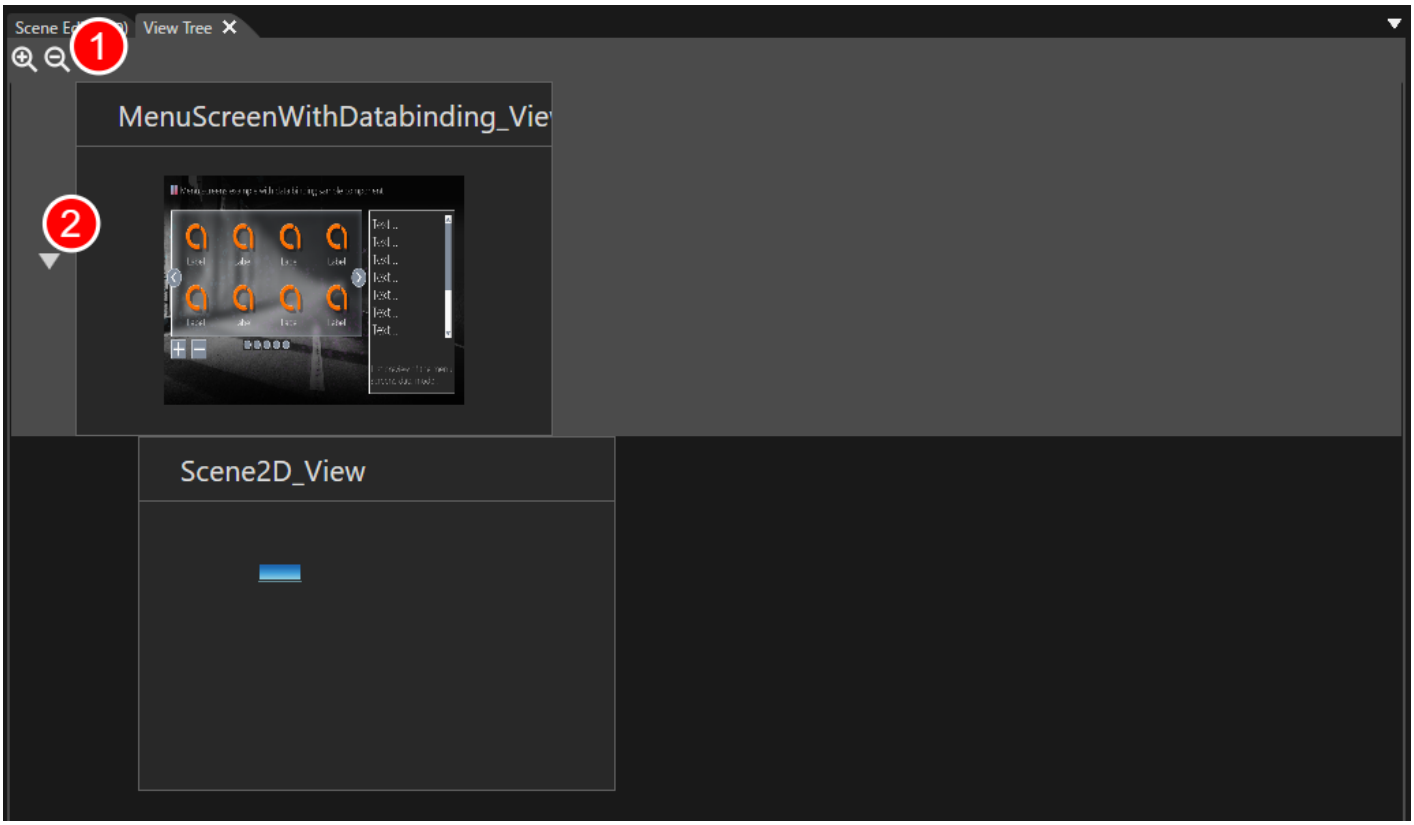
View Tree is the editor for arranging **View** in the solution as a tree and composing screen transition logic based on that structure. When you define transition relationships between Views on the tree, that information is exported as the **Global State Machine** at asset generation and becomes available as destination candidates for the **JumpToView** action behavior.

When you set a parent–child relationship between Views, the child View is activated in sync when the parent View becomes active, allowing you to manage groups of screen elements that should always be shown together. In addition, by associating an Action Chain created in the **Action Chain Editor** with each View, you can define that View's event logic and conditional transitions.

View Tree panel UI

The **View Tree** panel is hidden by default. To show the panel, do either of the following:

- Select **View** > **View Editor** > **View Tree** from the toolbar menu.
- In **Solution Explorer**, double-click an existing **View Tree**.



1	**Zoom icon**	Zooms the previews of Views in the View Tree.

2	**Expand/Collapse icon**	Appears when a View has child Views. Click to toggle the visibility of the child Views.

Basic operations

Create a new View Tree

Follow these steps to create a new **View Tree**:

1. In **Solution Explorer**, open the context menu at the desired location.
2. Choose **Add > New Item**.
   - The **Add New Item** dialog opens.
3. In the dialog's left pane, select **View Tree**, then click **OK**.
   - The value you enter in **Name** becomes the new View Tree's name.
4. A new **View Tree** is created in **Solution Explorer**.
5. Double-click the newly created View Tree to launch the **Condition Editor**.

> For the View Tree you created, you can use the standard operations in **Solution Explorer** to **copy**, **duplicate**, **delete**, and **rename** it.

Place Views

Drag and drop Views that you have defined beforehand in the **View Definition Editor** onto the View Tree panel.

- Dropping onto a blank area of the View Tree panel places the Views side by side (as peers).
- Dropping onto an existing View places the dropped View as that View's child.

> When a parent–child relationship is set between Views, the child View's event logic is activated in sync when the parent View becomes active.

Configuring in Action Chain Editor

For each View placed on the **View Tree**, you can configure its event logic in the **Action Chain Editor**. You can launch the Action Chain Editor in either of the following ways:

- Double-click the View you want to configure.
- Select the View and choose **Open Action Chain Editor** from the context menu.

Exported as a Global State Machine

- The information you build on the View Tree (relationships between Views and their transition logic) is exported as a **Global State Machine** when assets are generated.
- The exported state machine is then executed on the engine side via **JumpToView**-related events/actions to control the actual screen transitions.

Action Chain Editor

Action Chain Editor is a visual editor for composing a **View**'s event logic on a per-trigger basis. Starting from user input or system events, you can define transitions between Views, play animations, and build complex event-driven behaviors—visually and intuitively—as Action Chain diagrams composed of **Trigger**, **Action**, and **Condition** nodes.

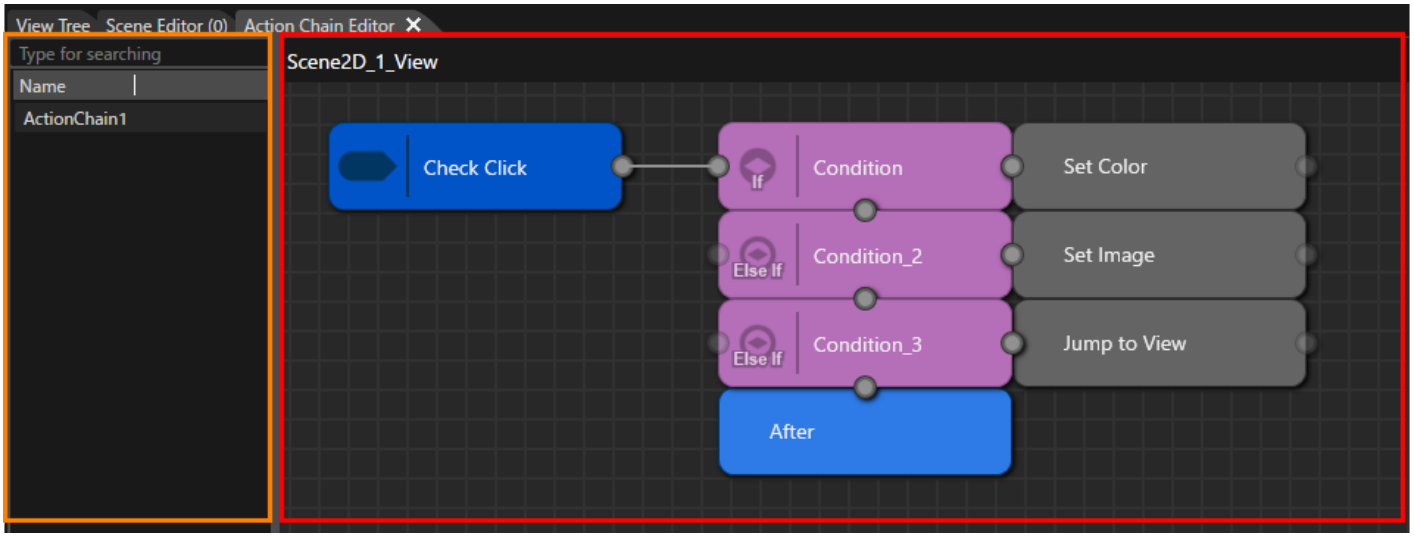
> Operations on the **Action Chain Editor** panel are available only after you have created a **View** in the **View Tree**.
> In the Action Chain Editor, a Condition node references the Condition Output created in the **Condition Editor**. You must define a Condition Block—and within it the Condition Output that represents the evaluation result—beforehand.

Action Chain Editor panel UI

Panel layout

The Action Chain Editor consists primarily of the following two panes.

> By default, the **Action Chain Editor** panel is hidden. To display it, choose **View > View Editor > Action Chain Editor** from the menu bar.



UI items	Description
Left pane (orange frame in the figure above)	Manages multiple **Action Chain** diagrams in a list. You can organize diagrams by feature or by input device, and perform operations such as creating, opening, renaming, and deleting.

UI items	Description
Right pane (red frame in the figure above)	The workspace for editing the selected **Action Chain** diagram. Place and connect **Trigger**, **Action**, and **Condition** nodes to define the processing flow.

Left pane

On the left pane, you can use the context menu to perform the following:

Menu	Description
New Diagram	Create a new Action Chain diagram.
Open	Open the diagram selected in the left pane.
Rename	Change the name of the diagram selected in the left pane.
Delete	Delete the diagram selected in the left pane.

Right pane

On the right pane of the Action Chain diagram, you can place and connect the following nodes from the context menu. The properties of each node placed on the diagram can be edited in the Properties panel.

Menu	Description
Add Trigger	Add Trigger Select a trigger node that serves as the entry point of the Action Chain. Use this menu to select the **Condition behavior** or **Behavior Building Block** (including Condition behaviors) that you want to use as the trigger. You can also create it by dragging and dropping a Condition behavior from the toolbox.
Add Action	Places a node that represents the action to execute. Select the **Action behavior** (for example, switching Views or playing an animation) or a **Behavior Building Block** (including Action behaviors) that you want to use. You can also create it by dragging and dropping an Action behavior from the toolbox.
Add Condition	Select the **ConditionOutput** created in the **Condition Editor**, or a **Behavior Building Block** (including Condition behaviors). Branches the subsequent flow (true/false) based on the evaluation result. • If no ConditionOutputs are defined in the *Condition Editor*, this menu is not shown. • ConditionOutputs that are not configured correctly are not listed in this menu
Add After	Defines processing that must run afterwards regardless of whether the condition evaluated to true or false. Use this to express post-processing or common flows.
Add Else	Used together with **Add Condition** menu to construct an if/else if/else logic structure.
Copy	Shown when a node is selected on the right pane. Copies the selected node.
Paste	Shown after performing Copy or Cut. Places a duplicate of the copied/cut node onto the diagram.

Menu	Description
Cut	Shown when a node is selected on the right pane. Cuts the selected node.
Delete	Shown when a node is selected on the right pane. Deletes the selected node.

Basic operations

Create a new diagram

To create a new Action Chain diagram, first select the target View in the **View Tree** panel.

1. In **View Tree**, select the View you want to define an Action Chain for.
2. Open the **Action Chain Editor** using either of the following:
   - Double-click the target View
   - Right-click the target View and choose **Open Action Chain Editor**.
3. In the **left pane** of the Action Chain Editor, open the context menu and select **New Diagram**.
   - A new **Action Chain** appears in the left pane.
   - Rename the diagram as needed.

Place a Trigger node

In an Action Chain diagram, start by placing a **Trigger** node that specifies “when to run.” The trigger hands off execution to the subsequent **Condition** group.

Place it on the right pane in either of the following ways:

- Right-click the right pane and choose **Add Trigger**, then select the trigger behavior to use from the list.
- Drag the desired trigger behavior from the **toolbox** onto the right pane.

Place ConditionOutput and Action nodes

A **ConditionOutput** (defined within a Condition Block) and an Action together represent one “if line.”

> A **ConditionOutput** node must be created beforehand in the **Condition Editor**.

- On the **right pane**, open the context menu and choose **Add Condition** to place a ConditionOutput node.
  - This node references a ConditionOutput created in the **Condition Editor**.

- For processing that should run when the ConditionOutput evaluates to **true**, choose **Add Action** to place an **Action** node.
  - Placing an Action node immediately to the **right** of a ConditionOutput has the following meaning:
    `Condition_1 + Action_1 → if (Condition_1) { Action_1(); }`
- Placing another ConditionOutput **vertically below** an existing ConditionOutput creates an **if / else if / else** block that is evaluated **top to bottom** (see the next section, “if/else if/else logic”).

if/else if/else logic

In an Action Chain diagram, you express `if / else if / else` by how you arrange **ConditionOutputs** and **Actions**.

- **Single row (side by side)**
  - Placing a **ConditionOutput** on the left and an **Action** on the right represents `if (Condition_X) { Action_X(); }` (see Case 2 below).
- **Stacked conditions in the same column**
  - From top to bottom, the rows are evaluated as `if` → `else if` → `else` (see Case 3).
  - Rows that include a **ConditionOutput** correspond to `if`/`else if`, and a row that has only an **Action** corresponds to `else` (see Case 6).
- **Nested if**
  - Connecting from an **Action** node to another **ConditionOutput** node creates a nested `if`, which evaluates an additional condition after that Action (see Cases 2 and Case 7).

Using After nodes

An **After** node is used to gather the “exits” of multiple branches into one.

- If you connect lines from multiple **Condition** rows to the same **After** node, all of those branches merge at that After.
  - Any **Action** placed to the right of an **After** node will always run next, regardless of which path was taken.
- If you insert an **After** node and then start a new **Condition** group, you can build a structure that “executes multiple independent `if` blocks in sequence under the same trigger” (see Cases 4 and 5 below).
- Even if an **After** node is present, processing ends there unless you connect an outgoing line. Therefore, placing an After node is not mandatory; however, adding one helps to:
  - Clearly mark the end of a branching block, and
  - Make it easier to add common post-processing later.

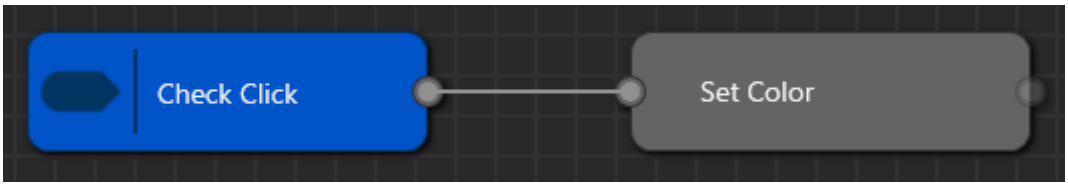
Action Chain Configuration Examples

This section shows how node arrangements on the **Action Chain** diagram (right pane) correspond to example **if** statements.

Case 1: Basic single-condition **if**

The simplest Action Chain: one condition and one action.

- A trigger (**Check Click**) detects the event; if the condition evaluates to **true**, a single action (**Set Color**) runs.




```
if (Check Click)
{
    Set Color();
}
```

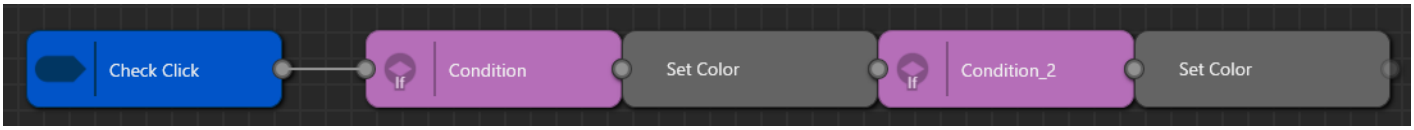
Case 2: Nested **if** (side-by-side layout)

This configuration shows how to represent a nested `if` that evaluates another condition within the branch of an existing condition.

Arrange the nodes in a single row as: **Trigger -> Condition -> Action (Set Color) -> Condition_2 -> Action (Set Color)**.

- First, if **Condition** is true, run **Set Color**.
- Then evaluate **Condition_2**; if it is true, also run **Set Color**.

With this **horizontal (side-by-side) layout**, you can express “an `if` inside an `if`” as a nested conditional that is evaluated after the preceding **Action**.




```
if (Check Click)
{
    if (Condition)
    {
        Set Color();
```

```
        if (Condition_2)
        {
            Set Color();
        }
    }
}
```

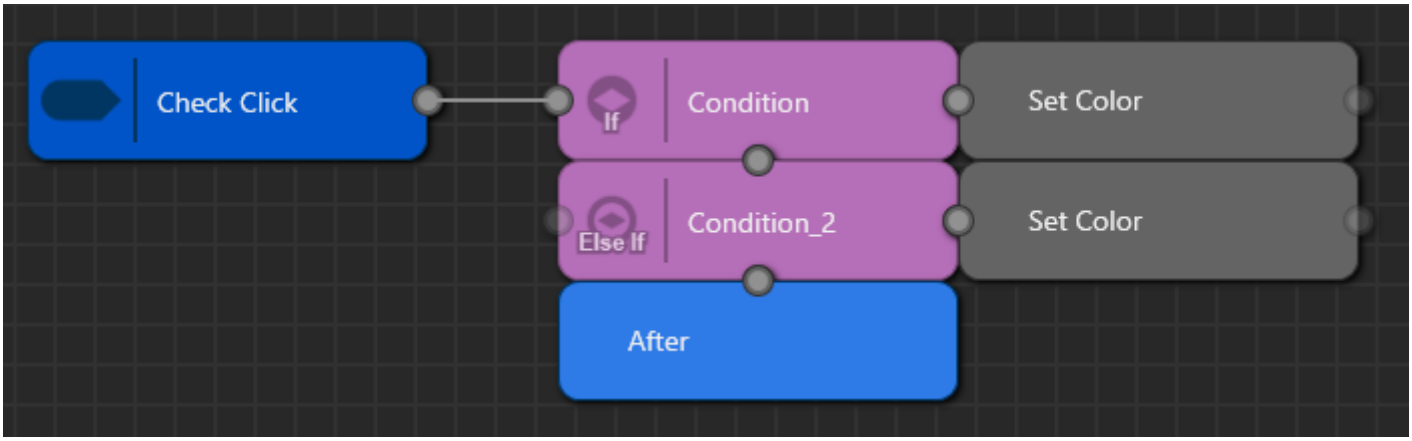
Case 3: Exclusive branching (if / else if)

This configuration shows how to evaluate multiple conditions for the same event **exclusively** (`if / else if`) so that **only one** action runs.

Place two rows—**Condition** and **Condition_2**—stacked vertically **one column to the right of the Trigger**, and place their corresponding Actions (**Set Color** / **Set Color**) to the right of each row. This represents:

- Upper row: `if (Condition) { SetColor(); }`
- Lower row: `else if (Condition_2) { SetColor(); }`

To make the branch's **common exit** explicit, place a single **After** node at the end of the block formed by **Condition** and **Condition_2**. The After node denotes "the exit for this `if`/`else if` group," and all subsequent processing should connect to the **right** of the After node. With this layout, it is visually clear that **regardless of which condition is met**, the flow ultimately **merges** into the same continuation.




```
if (Check Click)
{
    if (Condition)
    {
        Set Color();
    }
    else if (Condition_2)
    {
```

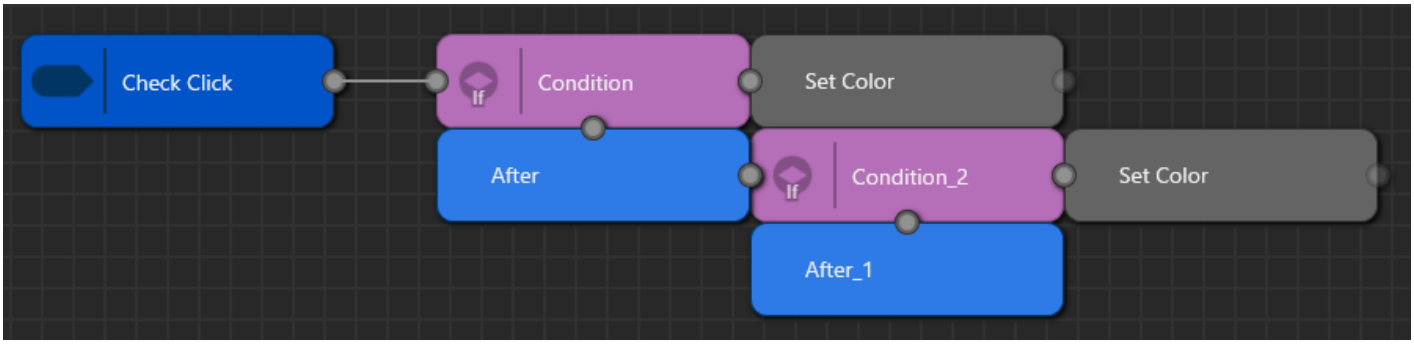
```
        Set Color();
    }
}
```

Case 4: Two independent **if** blocks under the same Trigger

This configuration evaluates two Conditions **independently and in sequence** within a single Trigger.

- The **After** placed beneath the `Condition + Set Color` row is a mandatory exit: the flow passes through it **whether** `Condition` **is true or false**, and then proceeds to `Condition_2` .
  - As a result, `Condition_2` is **always** evaluated, regardless of the outcome of `Condition`.
- The **After_1** placed beneath the `Condition_2 + Set Color` row marks the **end** of the entire chain.

By contrast with **Case 3**, where stacking ConditionOutputs in the same column and using a **single After** yields **exclusive branching** (i.e., the next Condition is checked **only if** the previous one was false), **Case 4** inserts an intermediate **After** between the rows so that **both** `if` **blocks are evaluated** in order.



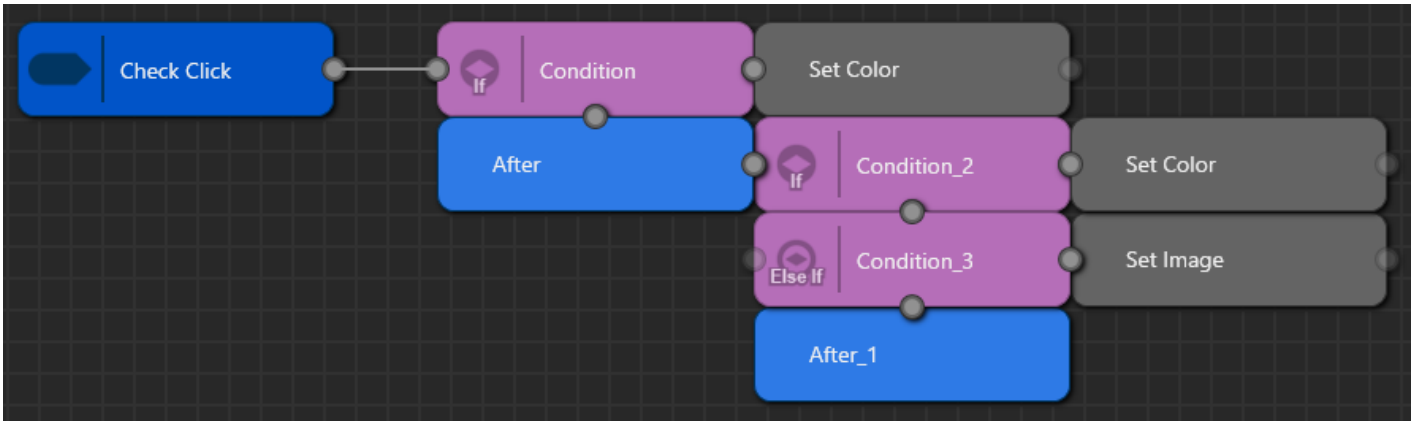

```
if (Check Click)
{
    if (Condition)
    {
        Set Color();
    }
    if (Condition_2)
    {
        Set Color();
    }
}
```

Case 5: Independent **if** + subsequent **if/else if** group

This configuration combines a single standalone `if` followed by an `if / else if` branching group under the same Trigger.

- **Row 1** (`Condition + Set Color`) corresponds to:
  `if (Condition) { SetColor(); }`
  - The **After** directly beneath Row 1 is a mandatory exit. The flow passes through it **whether `Condition` is true or false**, and then proceeds to the group formed by **Condition_2** and **Condition_3**.
- **Rows 2 and 3** (`Condition_2 + Set Color`, `Condition_3 + Set Image`) represent:
  `if (Condition_2) { SetColor(); } else if (Condition_3) { SetImage(); }`
  - When **Condition_2** is true, **only** `SetColor()` runs; **only when it is false** is **Condition_3** evaluated.
- The **After_1** beneath these two rows is the **common exit** for the `if / else if` group.

Compared with **Case 4**, where both **Condition_1** and **Condition_2** were independent `if` blocks, **Case 5** has an **independent** `if` for **Condition_1**, followed by an **`if / else if` group** formed by **Condition_2** and **Condition_3**.



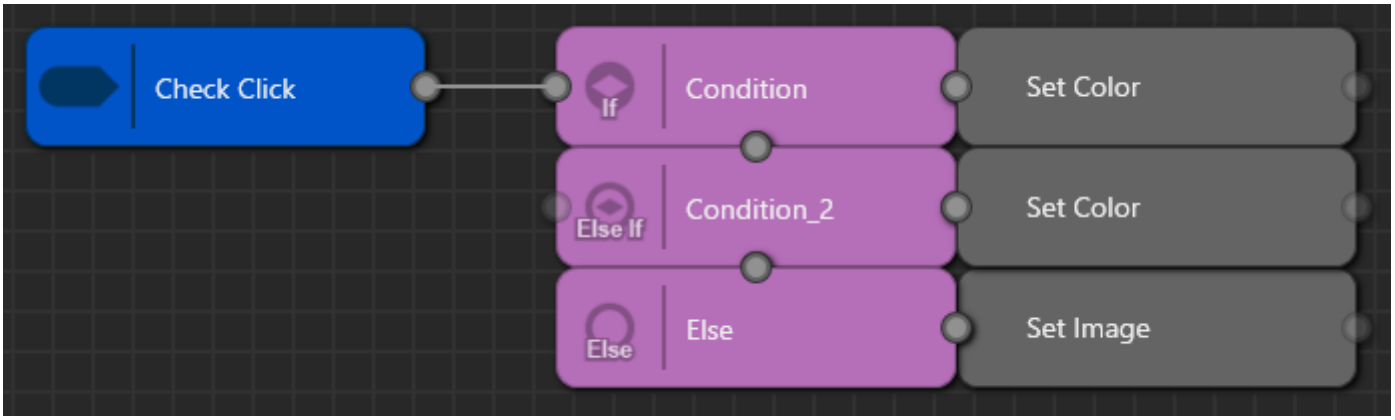

```
if (Check Click)
{
    if (Condition)
    {
        Set Color();
    }
    if (Condition_2)
    {
        Set Color();
    }
    else if (Condition_3)
    {
        Set Image();
    }
}
```

Case 6: Three-way branching (if / else if / else)

This configuration shows a classic **exclusive** branch with three paths under a single Trigger.

- **Row 1: “Condition + Set Color”**
  Corresponds to `if (Condition) { SetColor(); }`.
  Set Color runs only when **Condition** is true.
- **Row 2: “Condition_2 + Set Color”**
  Corresponds to `else if (Condition_2) { SetColor(); }`.
  Set Color runs only when **Condition** is false **and Condition_2** is true.
- **Row 3: “Else + Set Image”**
  Corresponds to `else { SetImage(); }`.
  Set Text runs only when both **Condition** and **Condition_2** are false.

The key point is that **exactly one** Action runs: evaluation proceeds **top to bottom** and executes the Action on the **first** row whose condition matches.



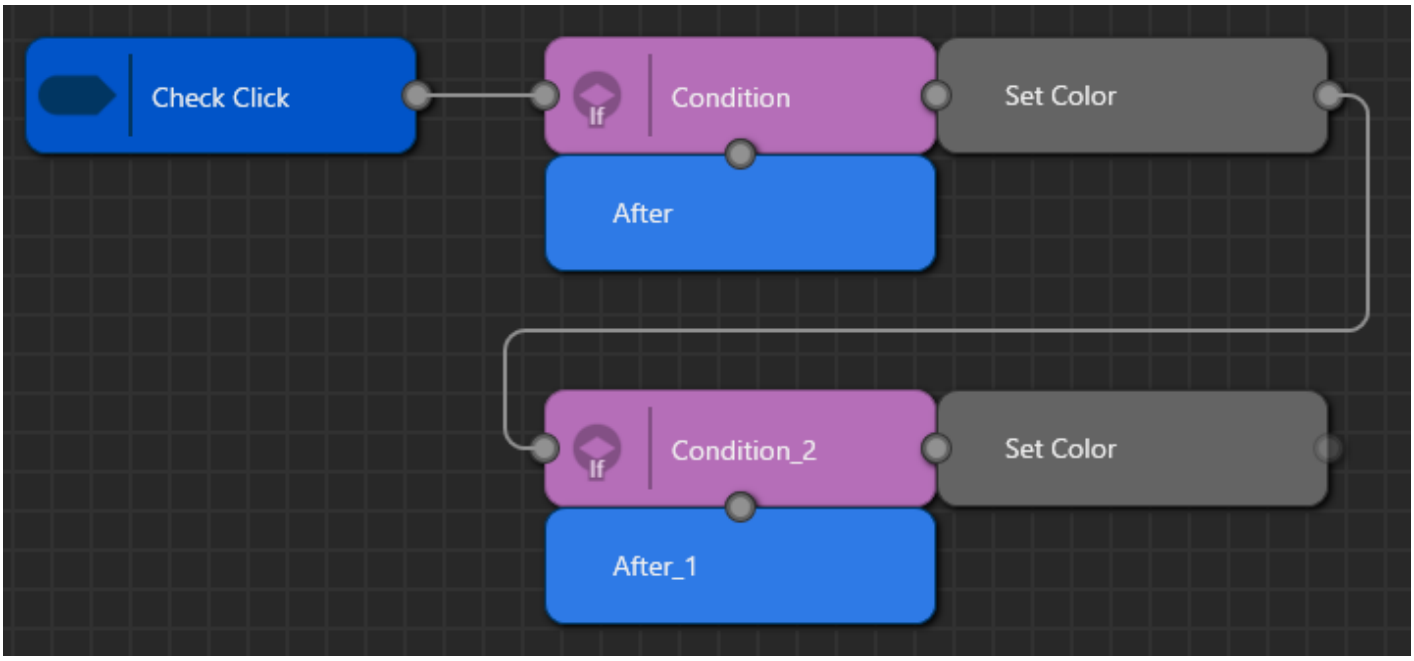

```
if (Check Click)
{
    if (Condition)
    {
        Set Color();
    }
    else if (Condition_2)
    {
        Set Color();
    }
    else
    {
        Set Image();
    }
}
```

Case 7: Nested `if` (After-based layout)

This configuration represents a nested `if` where one **ConditionOutput** is evaluated only inside the branch of another.

- The flow goes from the Trigger **Check Click** to **Condition**—this is the outer `if (CheckClick)`.
- The row **“Condition + Set Color”** corresponds to `if (Condition) { SetColor(); … }`.
- Because there is a connector from **Set Color** down to **Condition_2**, **Condition_2** is evaluated **only when** `Condition` **is true**. If `Condition` is false, the flow does **not** proceed to `Condition_2`.
- The lower row **“Condition_2 + Set Color”** corresponds to `if (Condition_2) { SetColor(); }`. Thus, **only when both** `Condition` **and** `Condition_2` are true do `SetColor()` and then `SetColor()` run in sequence.
- **After** and **After_1** serve as visual **exits/merge points** for the **Condition** block and the **Condition_2** block, respectively.

In contrast to **Case 4**, where two ConditionOutputs are treated as **two independent** `if`**s** that are both evaluated, **Case 7** places **Condition_2** entirely **inside Condition**, forming a classic **nested** `if` structure.




```
if (Check Click)
{
    if (Condition)
    {
        Set Color();

        if (Condition_2)
        {
```

```
            Set Color();
        }
    }
}
```

Condition Editor

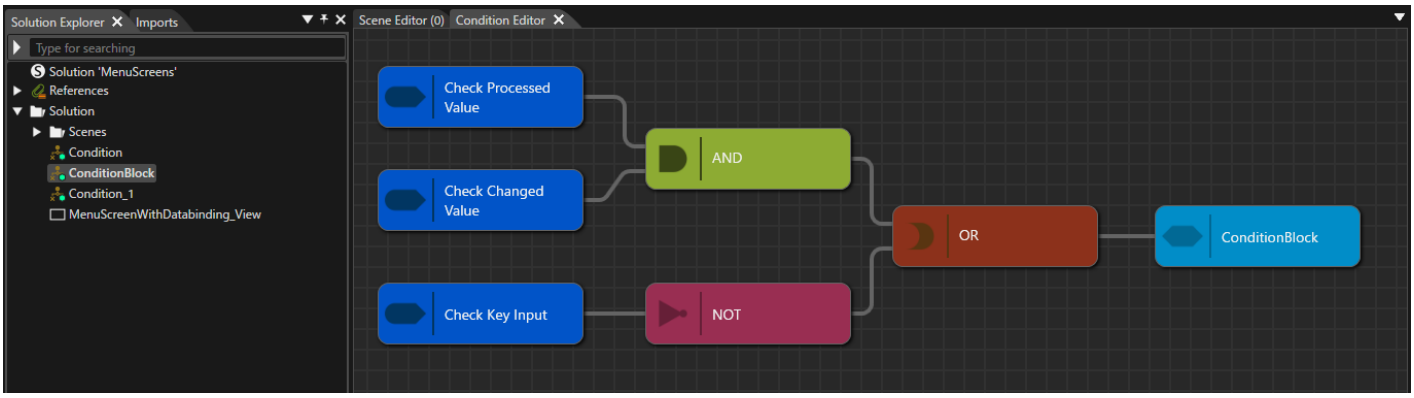
The **Condition Editor** is a visual editor for defining conditional logic that branches the processing flow based on input devices and the solution state. Conditional logic is managed in **Condition Blocks**. Within a Condition Block, you define one or more **ConditionOutput**s that represent the evaluation results (true/false). ConditionOutputs can be referenced from Condition nodes in the **Action Chain Editor**. They can also be assigned to the Condition property of behaviors and Fusion, allowing the same conditional logic to be reused in multiple places across the solution.

Tool UI overview

Panel layout

Double-click any **Condition Block** in **Solution Explorer** to open the **Condition Editor** panel. On this panel, you build a logical expression by placing and connecting **Check / AND / OR / NOT** nodes. Condition Blocks you create here can be referenced from the **Action Chain Editor**.

> By default, the **Condition Editor** panel is hidden. To display it, choose **View > View Editor > Condition Editor** from the menu bar.



On the **Condition diagram**, you can place and connect nodes via the context menu. When you open a **Condition Block** in the Condition Editor, a default ConditionOutput node is placed automatically on the diagram. ConditionOutputs can be added or deleted as needed. The properties of nodes placed on the diagram can be edited in the **Properties panel**.

> A **ConditionOutput** node represents the evaluation result (true/false) of the logical expression defined in the **Condition Block**. The logical expression you create in the Condition Editor is stored under each **ConditionOutput** node name and can be referenced from the **Action Chain Editor**. The same ConditionOutput can also be assigned to the Condition property of behaviors and **Fusion**.

Node type	Description
Add Check	Select a **Check** node that serves as the entry point for configuring the condition expression. Use this menu to select the **Condition behavior** or **Behavior Building Block** (including Condition behaviors) that you want to use as the starting point. You can also create it by dragging a Condition behavior from the toolbox onto the diagram.
Add AND / OR / NOT	These are logical operator nodes that combine multiple **Check** nodes and sub-conditions. Use **AND / OR / NOT** to express composite conditions.
Add OUTPUT	Add a new **ConditionOutput** node.
Copy / Cut / Delete / Paste	This menu becomes available when a node on the diagram is selected. It lets you **copy** , **cut**, **delete**, and **paste** the selected node.

Basic operations

Create a new Condition Block

Follow these steps to create a new **Condition Block**:

1. In **Solution Explorer**, open the context menu at the desired location.
2. Choose **Add > New Item**.
   - The **Add New Item** dialog opens.
3. In the dialog's left pane, select **Condition Block**, then click **OK**.
   - The value you enter in **Name** becomes the new Condition Block's name.
   - The **Category** setting determines which category (location) the Condition Block is registered under in the toolbox.
4. A new **Condition Block** is created in **Solution Explorer**.
   - At the same time, the Condition Block you created is added to **Behaviors** in the **toolbox**.
   - For details on where it is placed, see Reusing a Condition Block below.
5. Double-click the newly created Condition Block to launch the **Condition Editor**.
   - On the Condition Editor panel diagram, there is a ConditionOutput node named **Condition**.

Reusing a Condition Block

The created ConditionBlock and each ConditionOutput within the ConditionBlock are registered in the toolbox. By dragging and dropping a ConditionBlock or ConditionOutput from the toolbox, you can reuse the same condition logic in the following places:

- **Condition Editor**: Place it as a Check node that references an existing ConditionOutput.
- **Action Chain Editor**: Assign it to the **Condition** property of a Condition node.
- **Behaviors/Fusion**: Assign the ConditionOutput to the **Condition** property of the corresponding behavior.

Also, where a **Condition Block** appears within the toolbox's Behaviors categories depends on whether the Category property is set.

- If **Category is set**: A category with the same name is added under the toolbox's Behaviors, and the Condition Block is shown there.
- If **Category is not set**: The Condition Block is automatically shown under an appropriate category in the toolbox's Behaviors, depending on the conditions you configured.

To check which category it appears in, select the target Condition Block in the Solution Explorer, then see **General > Category** in the Properties panel.

> For the created Condition Block and the **ConditionOutput** nodes placed within the Condition Block, you can use the standard operations in **Solution Explorer** to copy, duplicate, delete, and rename them.

Creating a Condition diagram

1. In **Solution Explorer**, double-click any Condition Block.
   - The **Condition Editor** panel opens.
   - On the panel, there is a ConditionOutput node named **Condition**.
2. On the right pane, open the context menu, choose **Add Check**, and select the Condition behavior to use.
   - It is placed on the diagram as a **Check** node.
3. Place **AND / OR / NOT** nodes as needed and build the logical expression.
4. As needed, add **AND / OR / NOT** nodes and connect the **Check** nodes to construct the expression.
5. Connect the **ConditionOutput** node at the end of the diagram.
   - Logical expressions created in the Condition Editor are saved under the name of each ConditionOutput node and can be referenced from Condition nodes in the Action Chain Editor.
   - The same ConditionOutput can also be assigned to the **Condition** property of the corresponding behaviors/Fusion for reuse.