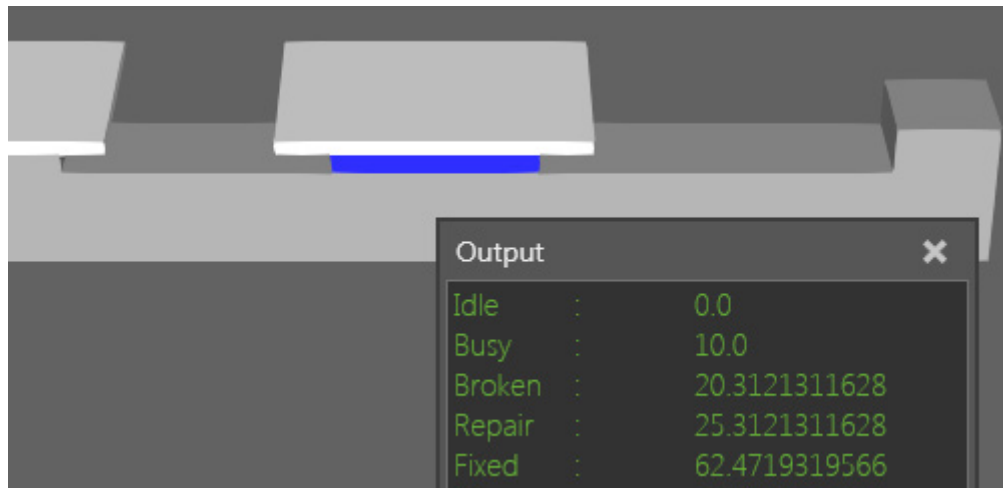


Component States

Visual Components 4.0 | Version: March 1, 2017



A component can be modeled to have one or more defined states, which you can control using Python API. States are created in a Statistics behavior and can be used to control the functionality of other behaviors. For example, when a component is in a Broken state, you can disable paths.

In this tutorial you learn how to:

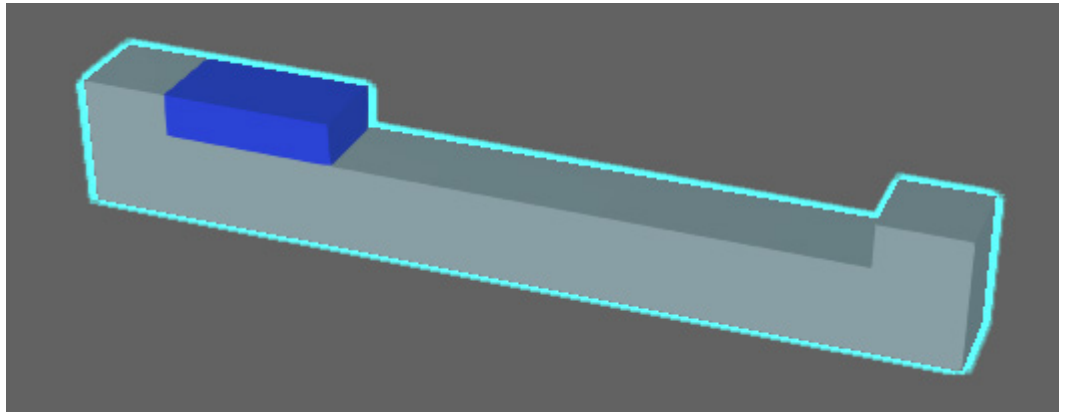
- Create default and custom states in a component.
- Write a component script that can control and record component states during a simulation.
- Write a component script that can simulate meantime between failures (MTBF) and meantime to repair (MTTR) states in a component.
- Export state statistics collected during a simulation to a CSV file.

Support
support@visualcomponents.com

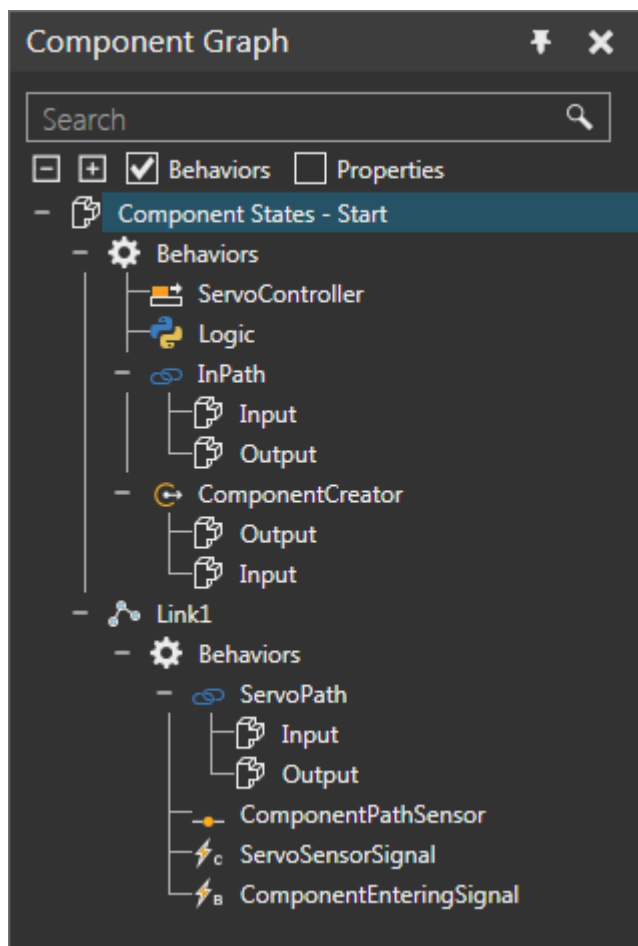
Community
community.visualcomponents.net

Getting Started

1. Open the **ComponentStatesStart.vcmx** file for this tutorial.

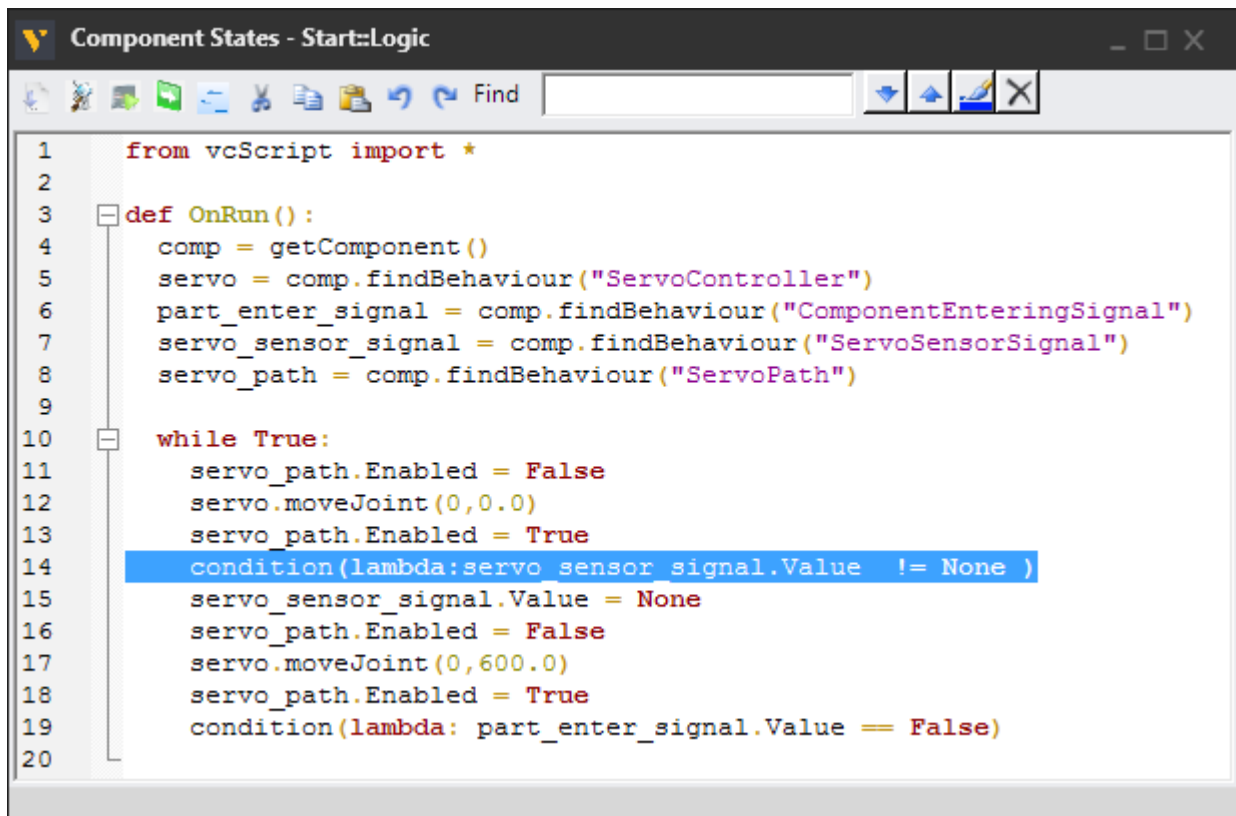


2. Click the **Modeling** tab, and then in the Component Graph panel, select the **Behaviors** check box, and then expand the component node tree.



The component is modeled to create parts during a simulation and move them along a path. After a part reaches a sensor in Link1, the part is moved to the other side of the component via a platform driven by a servo controller. A Python Script is used to define the logic of the process and manage the servo controller.

3. In the Component Graph panel, under the root node, double-click **Logic** to access its script editor.

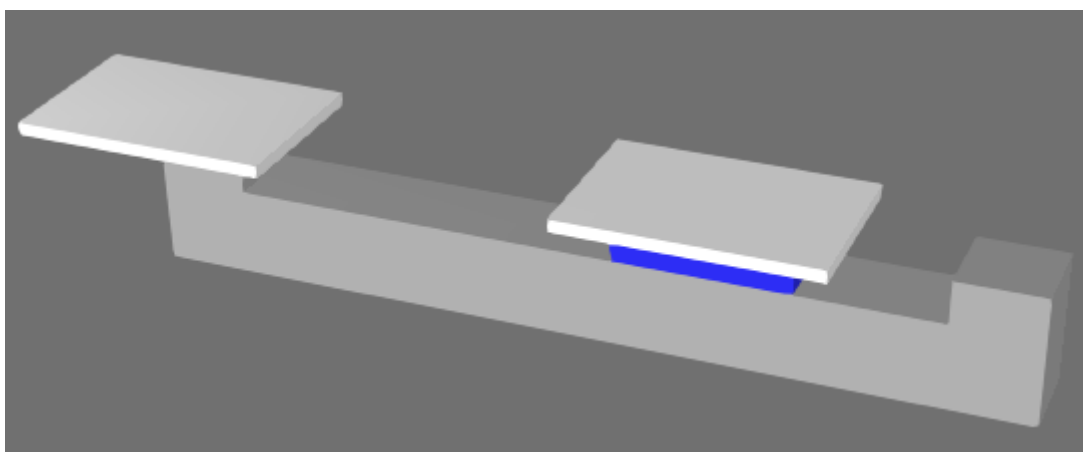


```
1  from vcScript import *
2
3  def OnRun():
4      comp = GetComponent()
5      servo = comp.findBehaviour("ServoController")
6      part_enter_signal = comp.findBehaviour("ComponentEnteringSignal")
7      servo_sensor_signal = comp.findBehaviour("ServoSensorSignal")
8      servo_path = comp.findBehaviour("ServoPath")
9
10     while True:
11         servo_path.Enabled = False
12         servo.moveJoint(0,0.0)
13         servo_path.Enabled = True
14         condition(lambda: servo_sensor_signal.Value != None )
15         servo_sensor_signal.Value = None
16         servo_path.Enabled = False
17         servo.moveJoint(0,600.0)
18         servo_path.Enabled = True
19         condition(lambda: part_enter_signal.Value == False)
20
```

4. In the script editor, enable **Trace execution**. This will allow you to know what line of code is being executed in the script.

NOTE! Signals are connected to the script and used as conditions for moving parts.

5. Run the simulation, verify parts are moved from one end to the other end of the component, and then reset the simulation.



A Statistics behavior can be added to collect data on the states of a component. Initially, a Statistics behavior does not have any states, so you will need to create them.

6. Add a **Statistics** behavior, and then in the Properties panel, click **Create default states**. This will add eight states that you can map to system states.

State Name		System State
Warmup		WarmUp
Break		Break
Setup		WarmUp
Idle		Idle
Busy		Busy
Blocked		Blocked
Broken		Fail
Repair		Repair
Click To Add Row		

Create default states

A **system state** is a constant and used for data collection. A **state** is defined by its label and mapped to a system state. For example, the default Warmup and Setup states are mapped to a WarmUp system state. That means whenever a component is in a Warmup or Setup state, data is being generated for the WarmUp system state.

Define Idle and Busy States

It is possible to define the state of a component by editing the State property of its Statistics behavior, which is a vcStatistics object. For example, you can change the state of a component to record when the component is idle and busy. To deal with state changes, you can use the OnStateChange event.

1. Access the **Logic** script editor, and then in the OnRun event, create a variable for the Statistics behavior.

```
def OnRun():  
...  
statistics = comp.findBehaviour("Statistics")
```

2. In the while loop, set the state of the component to "Idle" immediately before the condition evaluating the ServoSensorSignal.

```
while True:  
...  
statistics.State = "Idle"  
condition(lambda:servo_sensor_signal.Value != None )
```

3. Now set the state of the component to "Busy" immediately before the servo controller moves a joint to 600mm.

```
while True:  
...  
statistics.State = "Busy"  
servo.moveJoint(0,600.0)
```

4. Create an event handler for the OnStateChange event that prints feedback about the state of the component, and then compile the code.

```
from vcScript import *  
  
def recordStateChange(stats,sim_time,state,comp):  
    print state, "\t\t", sim_time  
  
def OnRun():  
...  
statistics = comp.findBehaviour("Statistics")  
statistics.OnStateChange = recordStateChange
```

5. Run the simulation, verify state changes are printed in the Output panel, and then reset the simulation.

```
Output
3      :      0.0
4      :      10.0
3      :      27.4
4      :      37.5
3      :      54.9
4      :      65.0
```

The state of a component is printed as an integer that corresponds to a system state constant. You can modify the event handler to print more specific data about a state by using the States and SystemStates properties of a vcStatistics object.

6. Edit the event handler to print a description about the new state by comparing the state integer to data given by the SystemStates property, and then compile the code.

```
def recordStateChange(stats,sim_time,state,comp):
    for s in stats.SystemStates:
        if state in s:
            state = s[0] #system state description
            print state, "\t\t", sim_time
```

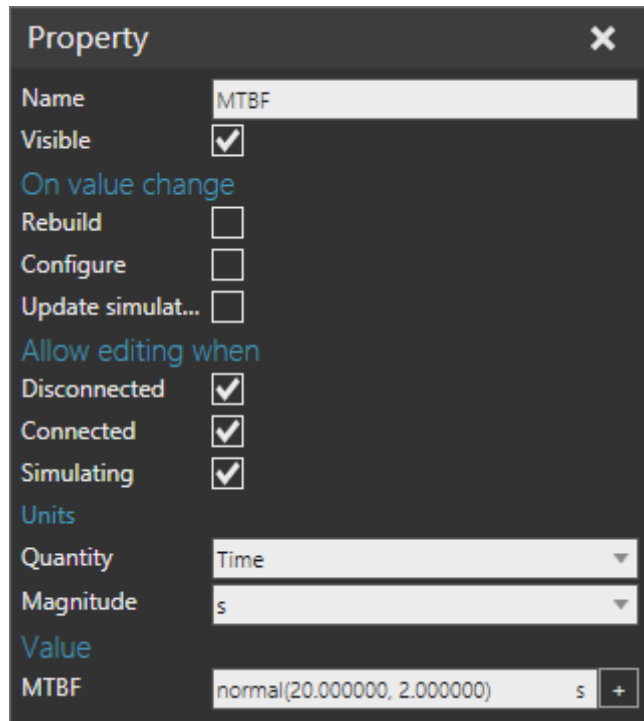
7. Run the simulation, verify state descriptions are printed in the Output panel, and then reset the simulation.

```
Output
System_Idle      :      0.0
System_Busy      :      10.0
System_Idle      :      27.4
System_Busy      :      37.5
System_Idle      :      54.9
System_Busy      :      65.0
```

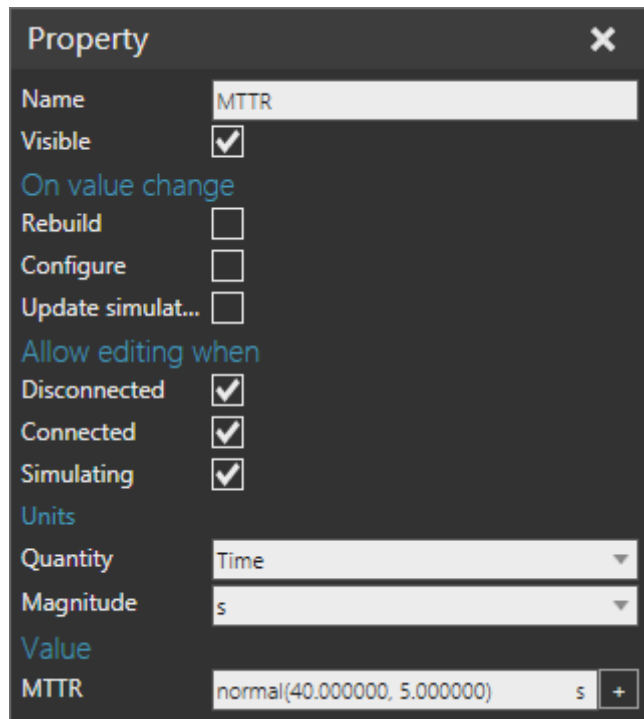
Define Failure and Repair States

It is possible to simulate a machine failure and the amount of time it takes to repair the machine.

1. Add a **Distribution** property, and then in the Property task pane, rename it "MTBF" for Mean Time Between Failure, and then set Quantity to **Time** and Value to a **normal distribution** with a **20.0** second average and a standard deviation of **2.0** seconds.



2. Add a **Distribution** property, and then in the Properties panel, rename it "MTTR" for Mean Time To Repair, and then set Quantity to **Time** and Value to a **normal distribution** with a **40.0** second average and a standard deviation of **5.0** seconds.

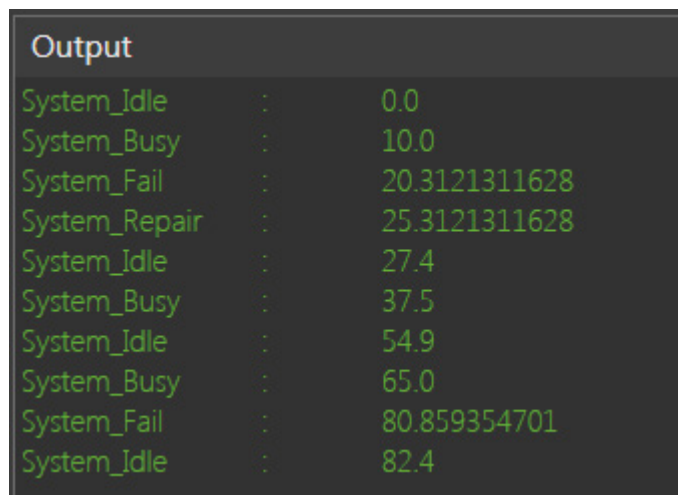


3. Add a **Python Script** behavior, and then in its editor, use the OnRun event to create a while loop that puts the component in Broken and Repair states, and then compile the code.

```
from vcScript import *

def OnRun():
    comp = getComponent()
    statistics = comp.findBehaviour("Statistics")
    while True:
        delay(comp.MTBF)
        statistics.State = "Broken"
        delay(5)
        statistics.State = "Repair"
        delay(comp.MTTR)
```

4. Run the simulation, verify in the Output panel that the component goes through Fail and Repair states, and then reset the simulation.












Output	
System_Idle	: 0.0
System_Busy	: 10.0
System_Fail	: 20.3121311628
System_Repair	: 25.3121311628
System_Idle	: 27.4
System_Busy	: 37.5
System_Idle	: 54.9
System_Busy	: 65.0
System_Fail	: 80.859354701
System_Idle	: 82.4

The time stamps of your System_Fail and System_Repair states might be different. You should notice that the Logic script is ignoring failure and repair times. That is, the component should not be moving parts while the machine is broken and being repaired. One solution is to create a new state that indicates the component is fixed.

Define Custom State

1. In the Component Graph pane, select the **Statistics** behavior.
2. In the Properties panel, under State Name, click an empty cell and name it **Fixed** and then map it to **Idle**, thereby creating a new custom state.

State Name		System State
Warmup		WarmUp
Break		Break
Setup		WarmUp
Idle		Idle
Busy		Busy
Blocked		Blocked
Broken		Fail
Repair		Repair
Fixed		Idle
Click To Add Row		

3. In the PythonScript editor, set the component to a Fixed state after completing a repair, and then compile the code.

```
while True:  
...  
    delay(comp.MTTR)  
    statistics.State = "Fixed"
```

4. In the Logic editor, modify the event handler to print the name of the current state, suspend the OnRun event if the state is Broken or Repair, resume the OnRun event if the state is Fixed, and then compile the code..

```
def recordStateChange(stats,sim_time,state,comp):  
    print stats.State, "\t\t", sim_time  
  
    if stats.State == "Broken" or stats.State == "Repair":  
        suspendRun()  
    elif stats.State == "Fixed":  
        resumeRun()
```

5. Run the simulation, verify in the Output panel that the state of a component goes from Broken to Repair to Fixed, and then reset the simulation.

```
Output
Idle      :      0.0
Busy     :     10.0
Broken   :    20.3121311628
Repair   :    25.3121311628
Fixed    :    62.4719319566
Idle     :    68.6719319566
Busy     :    68.6719319566
Broken   :    80.859354701
Idle     :    84.85
Repair   :    85.859354701
Fixed    :   119.859134453
Busy     :   119.859134453
```

There might be some timing issues with the execution of both scripts. For example, you might see the component go to an Idle state before it is being repaired. One way you can try to avoid this issue is to create a condition that checks the current state of the component before transitioning to a new state.

6. In the Logic script editor, modify the event handler to only print state information, and then create a function that returns a True value if the state of the component is not Broken or Repair, otherwise the function returns a False value.

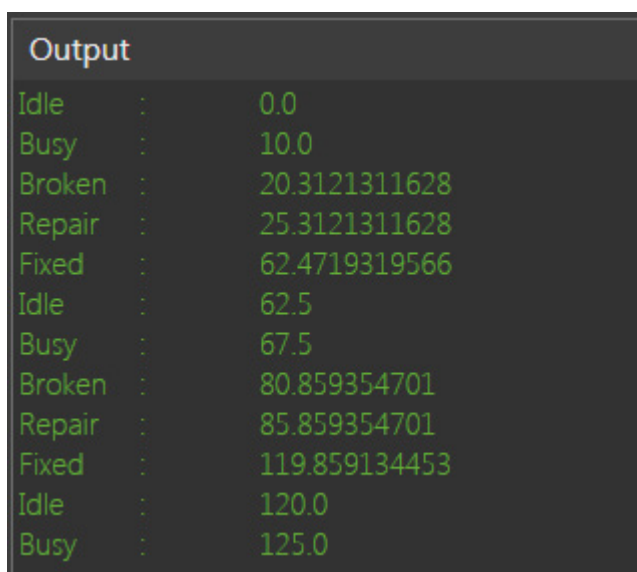
```
def recordStateChange(stats,sim_time,state,comp):
    print stats.State, "\t:\t", sim_time

def checkState(stats):
    state = stats.State
    if state == "Broken":
        return False
    elif state == "Repair":
        return False
    else:
        return True
```

7. In the OnRun event, add a condition that calls your state checking function before you set the component in an Idle state, and then compile the code.

```
def OnRun():
...
    while True:
...
        condition(lambda: checkState(statistics))
        statistics.State = "Idle"
```

8. Run the simulation, verify in the Output panel that the Idle state does not jump ahead of the Repair state, and then reset the simulation.



State	Time
Idle	0.0
Busy	10.0
Broken	20.3121311628
Repair	25.3121311628
Fixed	62.4719319566
Idle	62.5
Busy	67.5
Broken	80.859354701
Repair	85.859354701
Fixed	119.859134453
Idle	120.0
Busy	125.0

9. In the Logic script editor, modify the event handler to stop and start the paths of the component based on its state, and then compile the code.

```
def recordStateChange(stats,sim_time,state,comp):
    print stats.State, "\t\t", sim_time

    comp = GetComponent()
    paths = [comp.findBehaviour("InPath"),comp.findBehaviour("ServoPath")]
    if stats.State == "Broken" or stats.State == "Repair":
        for p in paths: p.Enabled = False
    elif stats.State == "Fixed":
        for p in paths: p.Enabled = True
```

10. Run the simulation, verify the paths of the component stop when the component is broken and being repaired and restart when the component is fixed, and then reset the simulation.

Export State Statistics

Statistics collected during a simulation can be exported and saved to a file.

1. Add a **Button** property, and then in the Property panel, rename it **Export To CSV**.
2. Add a **Python Script** behavior, and then in its script editor, add the following code to define an event handler for the OnChanged event of the button that exports the state names and times to a CSV file, and then compile the code.

```
from vcScript import *
import csv
import os.path

def exportStateStatistics(property):
    #setup the file
    username = os.getenv("username")
    path = os.path.join("C:\\Users\\", username, "Desktop")
    f = open(path+"\\example.csv", "w")

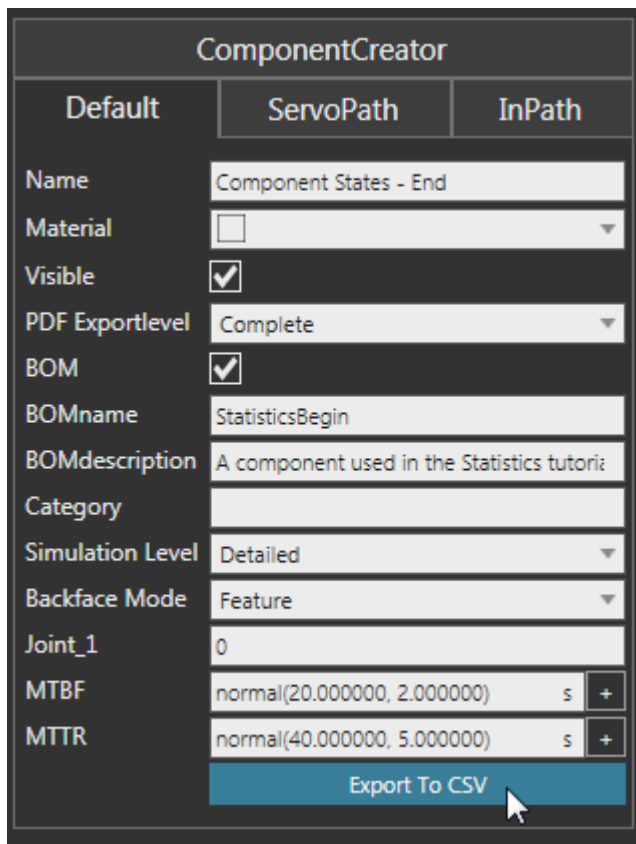
    #collect state data and write to file
    state_writer = csv.writer(f, delimiter=",")
    state_writer.writerow(getStateData())
    f.close()

def getStateData():
    list = []
    statistics = comp.findBehaviour("Statistics")
    for state in statistics.States:
        list.append(state[0])
        list.append(statistics.getTime(state[0]))
    return list

comp = getComponent()
button = comp.getProperty("Export To CSV")
button.OnChanged = exportStateStatistics
```

NOTE! The script will create an "example.csv" file in your Desktop directory.

3. Run the simulation to allow state statistics to be collected for a few minutes in simulation time, and then reset the simulation.
4. In the Component Graph panel, select the **root node**, and then in the Component Properties panel, Default tab, click **Export to CSV**.



5. On your device, verify the state statistics of the component was exported to a CSV file.

Review

In this tutorial you learned how to create and define component states during a simulation. You know how to monitor state change events and simulate meantime between failures (MTBF) and meantime to repair (MTTR). You also know how to use states to control the functionality of other behaviors in a component. Finally, you know how to export state data to a file.