3815ICT-Software Engineering Workshop 4

Activity 1

Study the documentation of BECIE. In MacOS and in Linux you need to run the command

java -cp becie.jar berp.BERP

to start BECIE. Create a system in BECIE called FourBitCounter. This system contains five component: one TIMER and four Light: D0 to D3. It has the following requirements

1. When FourBitCounter is started, it is in the state of Ready. While in this state, the internal count of FourBitCounter is zero (FourBitCounter->Count=0) and the four lights are in their state of Off each.

2. While in the Ready state, FourBitCounter can accept an environment event called Start. Once it receives the event Start, FourBitCounter will be in the state of Working.

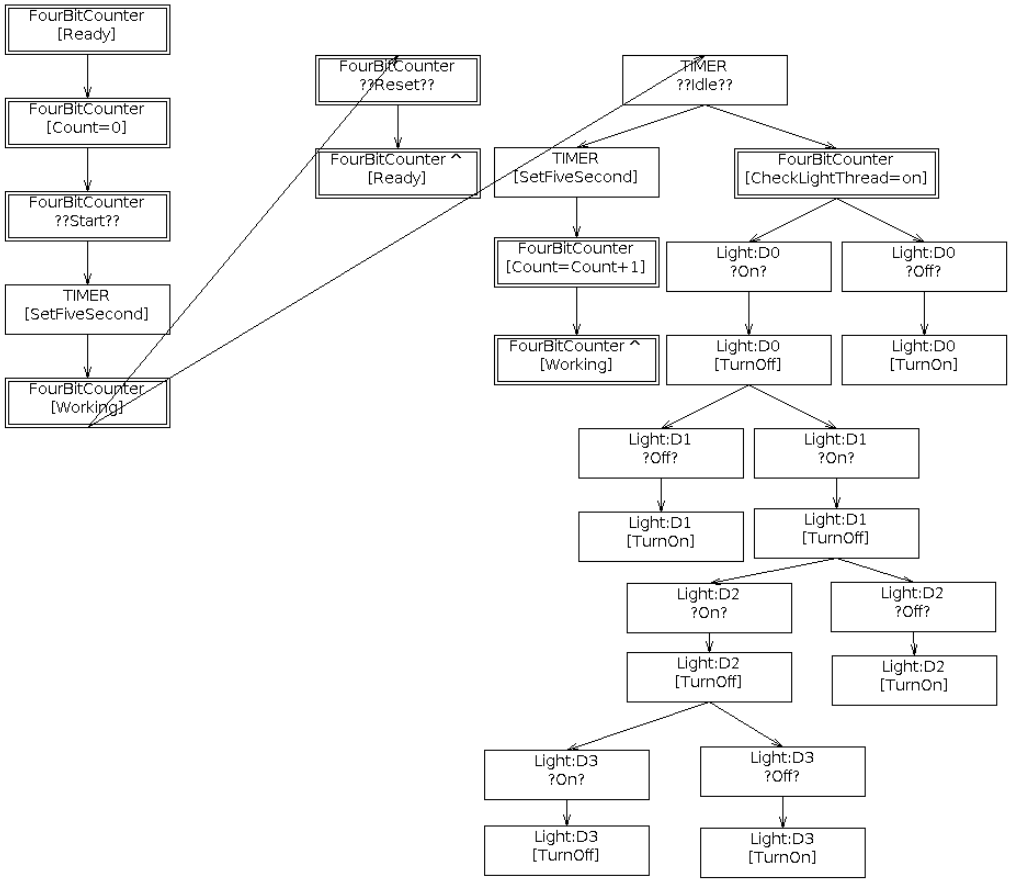
3. When FourBitCounter is in the state of Working, its internal count will automatically increase by one for every five seconds. Also, when the internal count is changed, the four lights will change to reflect the value of Count in binary. For example, when FourBitCounter->Count=11, the four lights will be in the following states:

D3=On, D2=Off, D1=On, and D0=Off

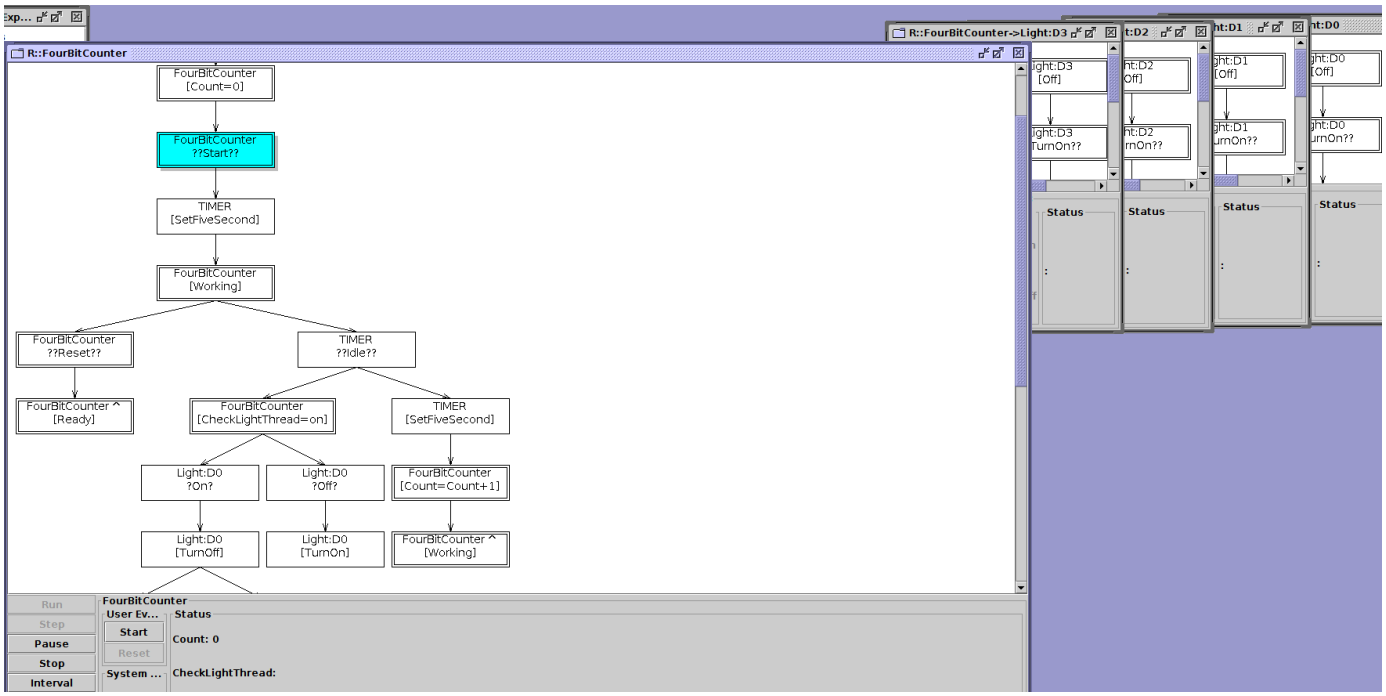
as in the Figure 1. When FourBitCounter is in the state of Working, it can accept an external event Reset. Once it receives the event Reset, the internal count will be 0 and all lights will be Off, and the system will back to the state of Ready.

Response 1

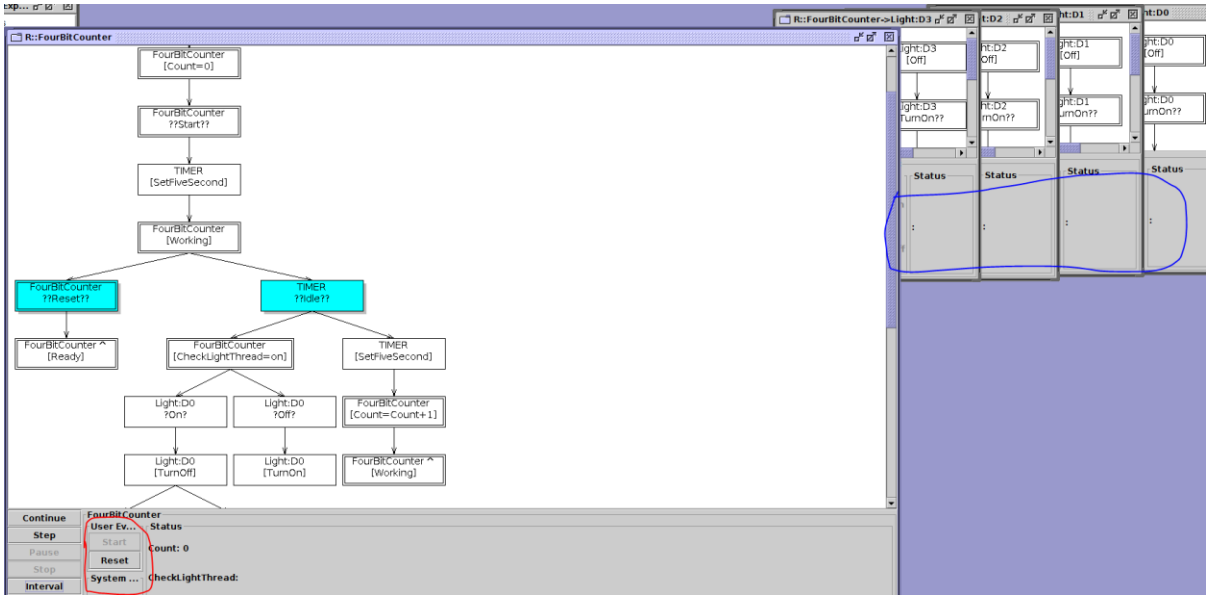
I managed to replicate the FourBitCounter system successfully and also managed to get it to run. The first figure refers to the complete design within the Becie System.



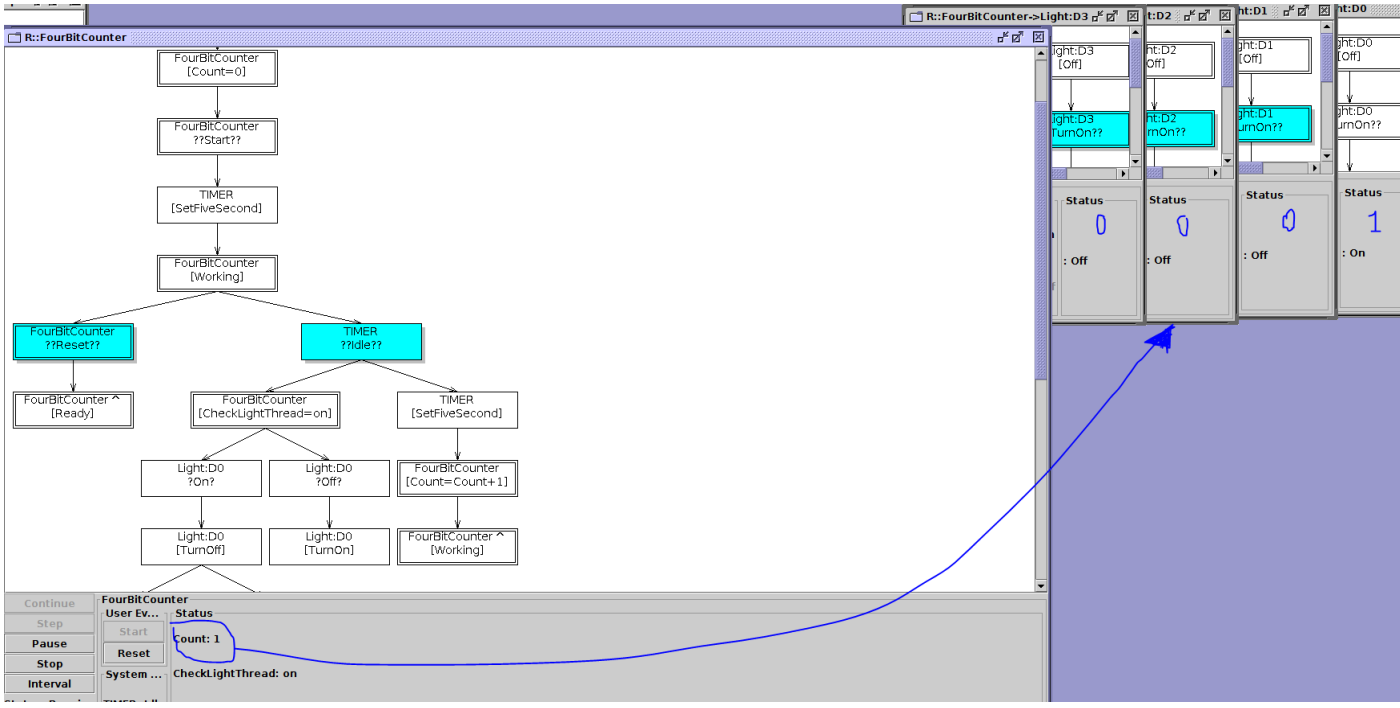
The system utilizes two standard components being TIMER and Light in order to display the count of FourBitCounter in binary form using lights. The count increments every 5 seconds and the Binary Counter displays the new counts. Ahead is the initial state before the start event has been received.



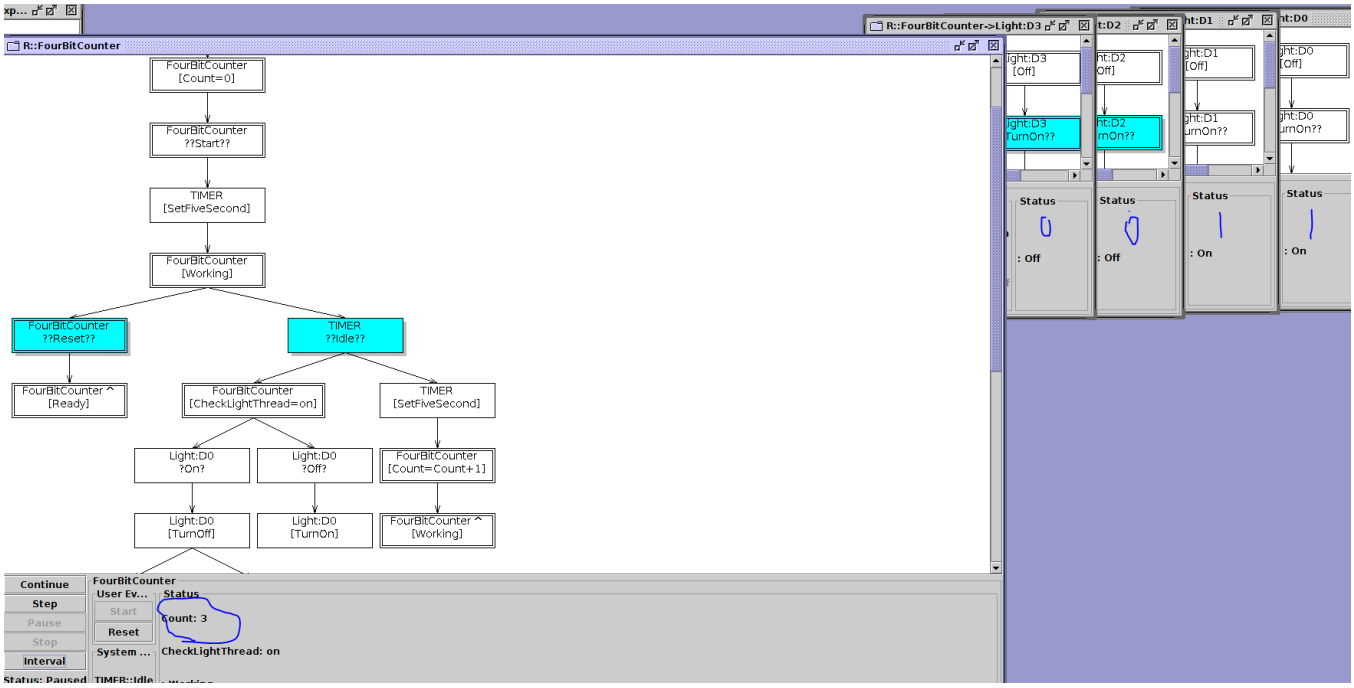
Once the start event is received, The timer will be set and an opportunity to reset the counter is presented while the timer runs down. The lights are off as is shown in blue and the reset option is available as shown in red.

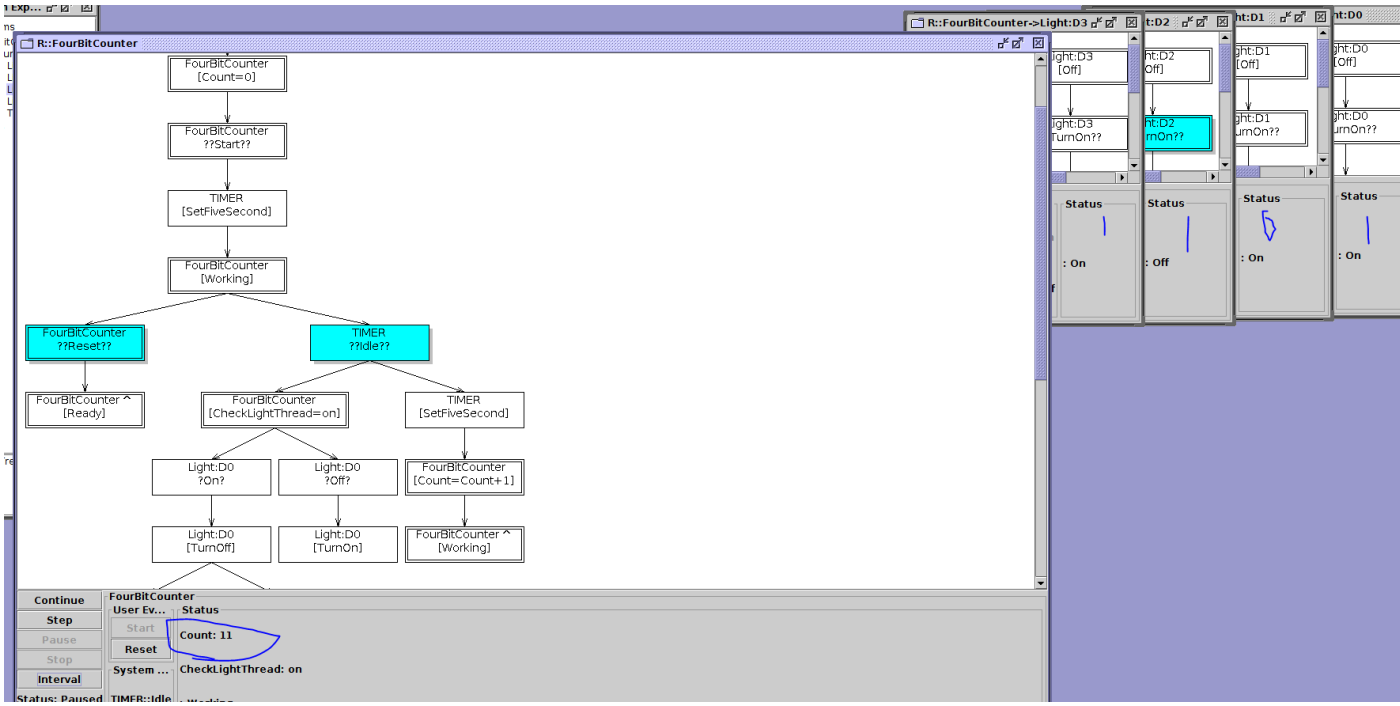


Once 5 seconds has passed, the counter will increment and the process will continually loop while the lights become active in reflecting the count number.



A few more examples are available.





Activity 2

Several Logic-labelled finite-state machines (LLFSMs) can be executed concurrently under a deterministic schedule. Typically each LLFSM get an opportunity to execute the activities in its current state. This is usually called to execute its ringlet. The turn of execution just loops around the machines listed in the scheduler.

Look at the LLFSM presented in Figure 2 (also drawn slightly differently in Figure 3) and predict what its its output in the following three scenarios.

1. You execute the machine in Figure 2 alone. ./clfsm/clfsm SimpleSuspendResume

2. You execute the master before the slave. ./clfsm/clfsm SimpleSuspendResume Simple

3. You execute the slave before the master. ./clfsm/clfsm Simple SimpleSuspendResume

Do not execute any of these scenarios yet. The exercise consist on you interpreting the model. Do you anticipate differences even between these executions?

Once you have written down what you expect the output to be, execute the LLFSM in a ROS (Robotic Operating System) catkin workspace provided for this lab. Copy the actual output of the execution into your report for this workshop and discuss the differences.

Response 2

Scenario 1 - Execute the machine in Figure 2 alone

The state **MASTER_INITIAL** will begin and the machine 'simple' will immediately be put in a state of suspension and masterCount will initialise at 0. The is_suspended clause will be satisfied so the *onExit* condition will execute and masterCount will increment by 1. After one second, the **RESTART** state will commit its *onEntry* command, resuming 'simple' and incrementing the counter. This will trigger the is_running clause which will move the state back to **MASTER_INITIAL** which will repeat the process, incrementing the masterCount and suspending 'simple'. This will continue infinitely and each *onEntry* and *onExit* will send a string to be output to console.

MASTER STATE: INITIAL – OnEntry COUNTER: 0

MASTER STATE: INITIAL – OnExit COUNTER: 1

MASTER STATE: RESTART – OnEntry COUNTER: 2

MASTER STATE: RESTART – OnExit COUNTER: 3

MASTER STATE: INITIAL – OnEntry COUNTER: 4

```
student@Virtualbox:~/Desktop/catkin_ws/devel/lib$ ./clfsm/clfsm SimpleSuspendRes
ume.machine/
MASTER STATE: INITIAL -  OnEntry COUNTER: 0
MASTER STATE: INITIAL -  Internal COUNTER: 1
MASTER STATE: INITIAL -  Internal COUNTER: 2
MASTER STATE: INITIAL -  Internal COUNTER: 3
MASTER STATE: INITIAL -  Internal COUNTER: 4
MASTER STATE: INITIAL -  Internal COUNTER: 5
MASTER STATE: INITIAL -  Internal COUNTER: 6
MASTER STATE: INITIAL -  Internal COUNTER: 7
```

The fact that 'Simple' did not exist must have made the is_suspended('simple') function false. The machine continually reverted back to its internal command and incremented the counter infinitely.

Scenario 2 - Execute the master before the slave

MASTER_INITIAL will immediately suspend ‘Simple’ machine. The master count will initiate and print

MASTER STATE: INITIAL – OnEntry COUNTER: 0

Because the ‘Simple’ machine has been successfully suspended, **MASTER_INITIAL** will be able to initiate an exit to print

MASTER STATE: INITIAL – OnExit COUNTER: 1

After One Second, **RESTART** will execute its OnEntry which will print

MASTER STATE: RESTART – OnEntry COUNTER : 2

RESTART will also resume the ‘Simple’ machine before executing its OnExit command and printing

MASTER STATE: RESTART – OnExit COUNTER : 3

Since it takes one second before the **MASTER_INITIAL** executes its OnEntry, the ‘Simple’ Machine will have an opportunity to run. It will print its own program and print

STATE: Initial – OnEntry COUNT: 0

‘Simple’ Machine will continue until **MASTER_INITIAL** executes its OnEntry again and suspends the ‘Simple’ Machine again. This will create a loop which allows ‘Simple’ machine to continue running only while it is resumed by the **RESTART** state. ‘Simple’ Machine will continue until it has completed execution at which point **MASTER_INITIAL** will no longer be able to suspend it. At this point **MASTER_INITIAL** will continue to run its internal function.

```
STATE: NEXT -   Internal COUNT: 82
STATE: NEXT -   Internal COUNT: 83
STATE: NEXT -   Internal COUNT: 84
STATE: NEXT -   Internal COUNT: 85
MASTER STATE: INITIAL -  OnEntry COUNTER: 0
MASTER STATE: INITIAL -  Internal COUNTER: 1
MASTER STATE: INITIAL -  OnExit COUNTER: 2
MASTER STATE: RESTART -  OnEntry COUNTER: 3
MASTER STATE: RESTART -  Internal COUNTER: 4
STATE: NEXT -  OnEntry COUNT: 86
STATE: NEXT -   Internal COUNT: 87
MASTER STATE: RESTART -  OnExit COUNTER: 5
STATE: NEXT -   Internal COUNT: 88
STATE: NEXT -   Internal COUNT: 89
STATE: NEXT -   Internal COUNT: 90
```

As predicted, **MASTER_INITIAL** and **RESTART** continued a loop in which 'simple' machine was able to run between being suspended and passed by the MASTER program. I did not predict that there would be an opportunity for both **MASTER_INITIAL** and **RESTART** to run their own internal programs. Perhaps this was due to the delay between the suspension and resume functions. The **MASTER_INITIAL** also managed to run its own internal function before the 'Simple' machine managed to start running in the beginning. Running the Simple Machine first may make the difference to this.

Scenario 3 - Execute the slave before the master

I believe there will be little difference between this scenario and the previous. The difference is that the 'Simple' Machine will have an opportunity to run itself before the **MASTER_INITIAL** begins the cycle. The loop will still be the same however.

```
student@Virtualbox:~/Desktop/catkin_ws/devel/lib$ ./clfsm/clfsm Simple SimpleSuspendResum
e
STATE: Initial - OnEntry  COUNT: 0
STATE: Initial -  OnExit COUNT: 1
MASTER STATE: INITIAL -  OnEntry COUNTER: 0
MASTER STATE: INITIAL -  Internal COUNTER: 1
MASTER STATE: INITIAL -  OnExit COUNTER: 2
MASTER STATE: RESTART -  OnEntry COUNTER: 3
MASTER STATE: RESTART -  Internal COUNTER: 4
STATE: NEXT -  OnEntry COUNT: 2
STATE: NEXT -   Internal COUNT: 3
MASTER STATE: RESTART -  OnExit COUNTER: 5
STATE: NEXT -   Internal COUNT: 4
STATE: NEXT -   Internal COUNT: 5
STATE: NEXT -   Internal COUNT: 6
```

```
STATE: NEXT -   Internal COUNT: 92
STATE: NEXT -   Internal COUNT: 93
STATE: NEXT -   Internal COUNT: 94
STATE: NEXT -   Internal COUNT: 95
MASTER STATE: INITIAL -  OnEntry COUNT: 0
MASTER STATE: INITIAL -  Internal COUNT: 1
MASTER STATE: INITIAL -  OnExit COUNT: 2
MASTER STATE: RESTART -  OnEntry COUNT: 3
MASTER STATE: RESTART -  Internal COUNT: 4
STATE: NEXT -  OnEntry COUNT: 96
STATE: NEXT -   Internal COUNT: 97
MASTER STATE: RESTART -  OnExit COUNT: 5
STATE: NEXT -   Internal COUNT: 98
STATE: NEXT -   Internal COUNT: 99
STATE: NEXT -   Internal COUNT: 100
STATE: NEXT -   Internal COUNT: 101
```

The 'Simple' Machine had an opportunity to run before it was suspended by **MASTER_INITIAL**. It then ran according to the pattern displayed in the second scenario. The loop continued endlessly.

Activity 3

The ROS-Kinetic machine in the labs has already MiPalCASE installed (an editor of LLFSMs written in swift), but you can also use MIEdit (an editor of LLFSMs written in java you can download www.mipal.net.au/downloads.php) and expand a state so that the entire panel of the editor shows the OnEntry, OnExit and Internal sections of a state. Navigate the catkin_ws provided for this workshop and open the machines in Figure 2 (alternatively Figure 3). The path is

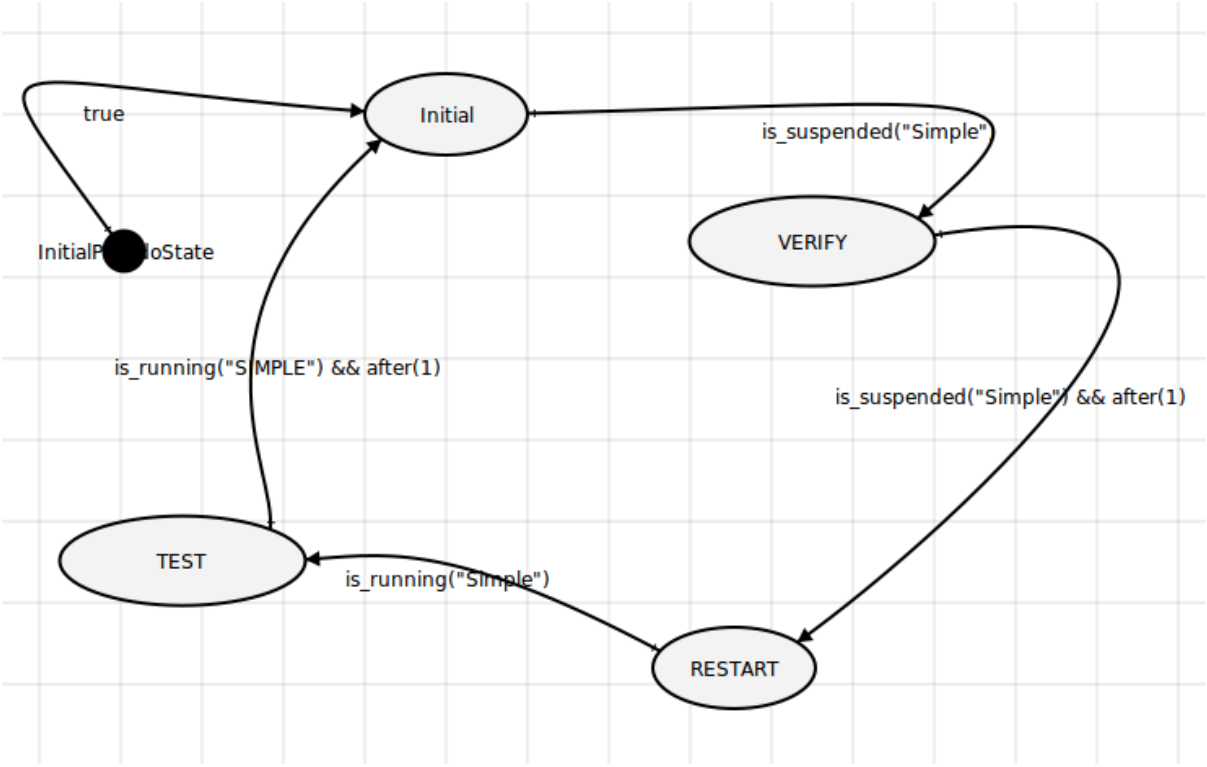
catkin_ws/src/SimpleSuspendResume/machine/SimpleSuspendResume.machine

1. Is Figure 2 (alternatively Figure 3 any different to what actually runs?

2. Take a snapshot if you find at least one difference that results in different behaviour.

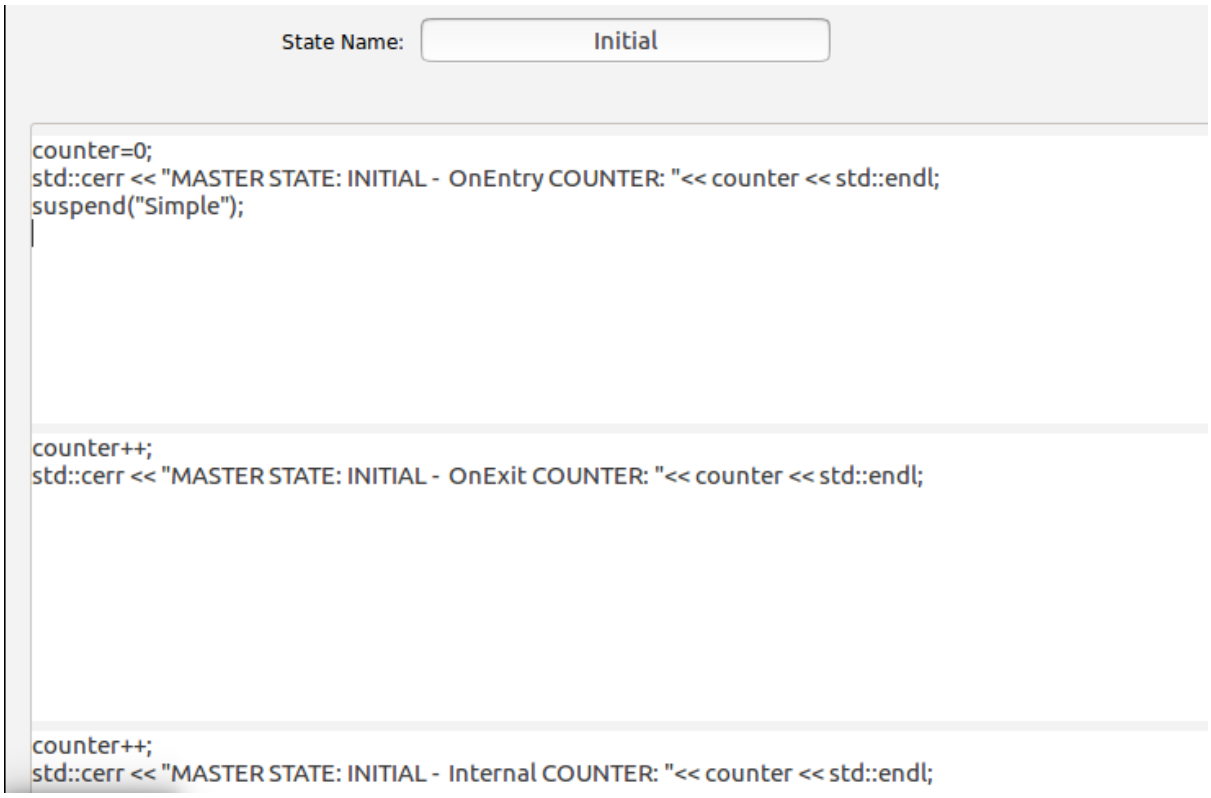
Response 3

Running the SimpleSuspendResume Machine in the MiPalCase environment yielded the following results:



The machine displayed exactly the same functionality as it did within figure 2:

There is no difference between the original figure and the MiPalCase version of it. Assuming neither of them have been connected to the Simple machine, the two should run identically. The commands are accessible in MiPalCase by clicking them:


State Name:
Initial
counter=0;
std::cerr << "MASTER STATE: INITIAL - OnEntry COUNTER: "<< counter << std::endl;
suspend("Simple");
counter++;
std::cerr << "MASTER STATE: INITIAL - OnExit COUNTER: "<< counter << std::endl;
counter++;
std::cerr << "MASTER STATE: INITIAL - Internal COUNTER: "<< counter << std::endl;