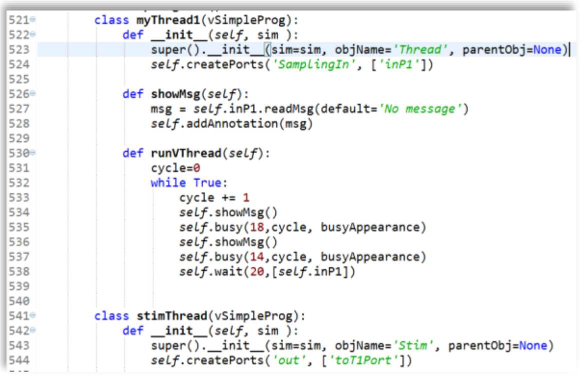
Moddy System Simulator User Guide

Issue:	1.3
Valid for Moddy Version	1.3.0
Date of Issue:	2017-09-07
Author(s):	© 2017 Klaus Popp

Model

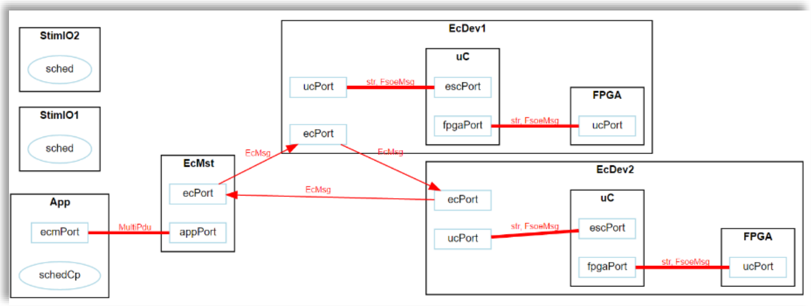


```
class myThread1(vSimpleProg):
    def __init__(self, sim ):
        super().__init__(sim=sim, objName='Thread', parentObj=None)
        self.createPorts('SamplingIn', ['inP1'])

    def showMsg(self):
        msg = self.inP1.readMsg(default='No message')
        self.addAnnotation(msg)

    def runVThread(self):
        cycle=0
        while True:
            cycle += 1
            self.showMsg()
            self.busy(18,cycle, busyAppearance)
            self.showMsg()
            self.busy(14,cycle, busyAppearance)
            self.wait(20,[self.inP1])


class stimThread(vSimpleProg):
    def __init__(self, sim ):
        super().__init__(sim=sim, objName='Stim', parentObj=None)
        self.createPorts('out', ['toT1Port'])
```

Simulate

Analyze

App
EcMst
EcDev1
EcDev2
cycleTrg
2.2ms
3.0ms
4.0ms
5.0ms
6.0ms
6.6ms



Contents

1 Introduction

Please see installation instructions at the end of the document!

1.1 What is Moddy

Moddy is a simulator to model and analyze the timing behavior of systems consisting of objects that communicate via messages. The objects are called "parts" in Moddy, to be consistent with the term SysML uses in internal block diagrams.

Moddy was written to analyze complex systems in the concept phase to validate the suitability of the concept. Moddy cannot prove that your concept will work, but it can help you to determine very quickly that it WILL NOT work. In this case, you will have to change your concept and analyze it again.

You describe the structure and the behavior of your model via a program written in "python" language.

After the simulation run, Moddy can produce a number of result files that you will analyze to evaluate whether the model behaves as expected. These result files are

- A sequence diagram
- An event trace
- A structure graph (block diagram)

Further result files, such as wave forms may be added in the future.

Moddy's Simulator is a classic "Discrete Event simulator". From Wikipedia https://en.wikipedia.org/wiki/Discrete_event_simulation:

A discrete-event simulation (DES) models the operation of a system as a discrete sequence of events in time. Each event occurs at a particular instant in time and marks a change of state in the system. Between consecutive events, no change in the system is assumed to occur; thus the simulation can directly jump in time from one event to the next.

1.2 Why another Simulator?

I was looking for a simulator that

- Supports Quick Model generation: Meaning: Modelling shall be doable by non-programmers in an intuitive and easy-to-remember way
- Is suitable to model communication between objects
- Can visualize those communication at least in form of sequence diagrams
- Is open source or at least affordable

I could not find commercial or open source tools that provided those features.

1.3 Quick Start

The following chapter describes the basic functions of Moddy by using the example "1_hello" tutorial model.

In a Moddy model, a system is composed of parts that communicate with other parts through messages.

In the “hello” tutorial, we simulate a conversation between two people, “Bob” and “Joe”, therefore “Bob” and “Joe” are the two parts used in the model.

1.3.1 Model Structure

The structure of the system is modelled via parts, ports and bindings between ports.

In the “hello” tutorial, we have two parts, Bob and Joe. Each part has an output port (mouth) and an input port (ears).

To simulate the time that Bob and Joe need for thinking, we create also a timer for both.

To model the behavior of Bob, we define a class “Bob”, which creates the ports and timers:

```
class Bob(simPart):
    def __init__(self, sim, objName):
        # Initialize the parent class
        super().__init__(sim=sim, objName=objName)

        # Ports
        self.createPorts('in', ['ears'])
        self.createPorts('out', ['mouth'])

        # Timers
        self.createTimers(['thinkTmr'])
        self.reply = ""

      ...
```

The main program creates an instance of Bob and Joe like this

```
    simu = sim()

    bob    = Bob( simu, "Bob" )
    joe    = Joe( simu, "Joe" )
```

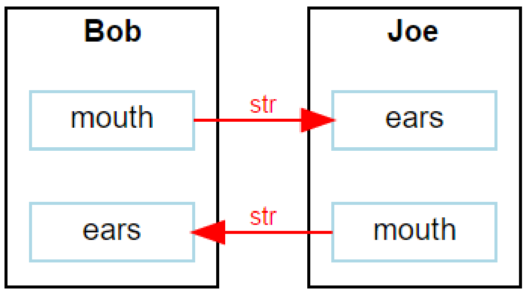
To allow Joe to hear what Bob says, we “bind” the ears of Joe to the mouth of Bob and vice versa. A binding is always initiated from the output port; you call the output port’s bind function and specify the input port that shall be bound:

```
    bob.mouth.bind(joe.ears)
    joe.mouth.bind(bob.ears)
```

Moddy can also output the structure of the model. After you created the structure, call:

```
    # Output model structure graph
    moddyGenerateStructureGraph(simu, '1_hello_structure.svg')
```

This is the resulting structure graph:



1.3.2 Model Behavior

To model the behavior of parts, your model:

- Sends messages to other parts
- Reacts on received messages from other parts
- Reacts on timer events

This means, the simulator calls the model code only whenever such an event occurs.

For example, Bob's behavior will be modelled as follows:

```
class Bob(simPart):
    …
    def earsRecv(self, port, msg):
        if msg == "Hi, How are you?":
            self.reply = "How are you?"
        else:
            self.reply = "Hm?"

        self.thinkTmr.start(1.4)
        self.setStateIndicator("Think")


    def thinkTmrExpired(self, timer):
        self.setStateIndicator("")
        self.mouth.send(self.reply, 1)
```

1.3.3 Ports and Messages

A message is send always from an "Output Port" to an "Input Port".

A part can have many Ports to communicate with other parts.

A message is send via the sending port's *send()* method:

```
        self.mouth.send(self.reply, 1)
```

In this example, *self.reply* is the message; here it is a string.

The second parameter defines the flight time, i.e. how long it takes until the message arrives at the input port.

On the receiver side, the part that owns the input port must define a “receive function”, which gets passed the message just received:

```
    def earsRecv(self, port, msg):
        if msg == "Hi, How are you?":
            self.reply = "How are you?"
        else:
            self.reply = "Hm?"

        self.thinkTmr.start(1.4)
        self.setStateIndicator("Think")
```

Note: This receive routine must be called always *<portName>Recv.*

1.3.4 Timers

A part can have many timers to control its own behavior.

A timer is stopped by default.

You start the timer via timer.start(timeout).

You stop (cancel) the timer via timer.stop()

You can restart an already running timer via timer.restart(timeout).

```
def thinkTmrExpired(self, timer):
        self.setStateIndicator("")
        self.mouth.send(self.reply, 1)
```

Note: The expiration routine must be called always *<timerName>Expired.*

1.3.5 Running Simulator

After the parts and bindings were created, the simulator can run:

```
    # let simulator run
    simu.run(stopTime=12.0)
```

Here we stop the simulator after 12 seconds. If no limit is given, the simulator would run until no more events to execute.

On the python console, the simulator outputs the simulation trace:

```
TRC:       0.0s >MSG    Bob.mouth(OutPort) // req=0.0s beg=0.0s end=1.0s dur=1.0s msg=[Hi Joe]
SIM: Simulator 0.9.0 starting
TRC:       1.0s <MSG    Joe.ears(InPort) // req=0.0s beg=0.0s end=1.0s dur=1.0s msg=[Hi Joe]
TRC:       1.0s ANN     Joe(Block) // got message Hi Joe
TRC:       1.0s T-START Joe.thinkTmr(Timer) // 2.0s
TRC:       1.0s STA     Joe(Block) // Think
TRC:       3.0s T-EXP   Joe.thinkTmr(Timer)
TRC:       3.0s STA     Joe(Block) //
```

```
TRC:       3.0s >MSG    Joe.mouth(OutPort) // req=3.0s beg=3.0s end=4.5s dur=1.5s msg=[Hi, How are you?]
TRC:       4.5s <MSG    Bob.ears(InPort) // req=3.0s beg=3.0s end=4.5s dur=1.5s msg=[Hi, How are you?]
TRC:       4.5s T-START Bob.thinkTmr(Timer) // 1.4s
TRC:       4.5s STA     Bob(Block) // Think
TRC:       5.9s T-EXP   Bob.thinkTmr(Timer)
TRC:       5.9s STA     Bob(Block) //
TRC:       5.9s >MSG    Bob.mouth(OutPort) // req=5.9s beg=5.9s end=6.9s dur=1.0s msg=[How are you?]
TRC:       6.9s <MSG    Joe.ears(InPort) // req=5.9s beg=5.9s end=6.9s dur=1.0s msg=[How are you?]
TRC:       6.9s ANN     Joe(Block) // got message How are you?
TRC:       6.9s T-START Joe.thinkTmr(Timer) // 2.0s
TRC:       6.9s STA     Joe(Block) // Think
TRC:       8.9s T-EXP   Joe.thinkTmr(Timer)
TRC:       8.9s STA     Joe(Block) //
TRC:       8.9s >MSG    Joe.mouth(OutPort) // req=8.9s beg=8.9s end=10.4s dur=1.5s msg=[Fine]
TRC:      10.4s <MSG    Bob.ears(InPort) // req=8.9s beg=8.9s end=10.4s dur=1.5s msg=[Fine]
TRC:      10.4s T-START Bob.thinkTmr(Timer) // 1.4s
TRC:      10.4s STA     Bob(Block) // Think
TRC:      11.8s T-EXP   Bob.thinkTmr(Timer)
TRC:      11.8s STA     Bob(Block) //
TRC:      11.8s >MSG    Bob.mouth(OutPort) // req=11.8s beg=11.8s end=12.8s dur=1.0s msg=[Hm?]
SIM: Simulator stopped at 11.8s
```

1.3.6 Visualizing Results

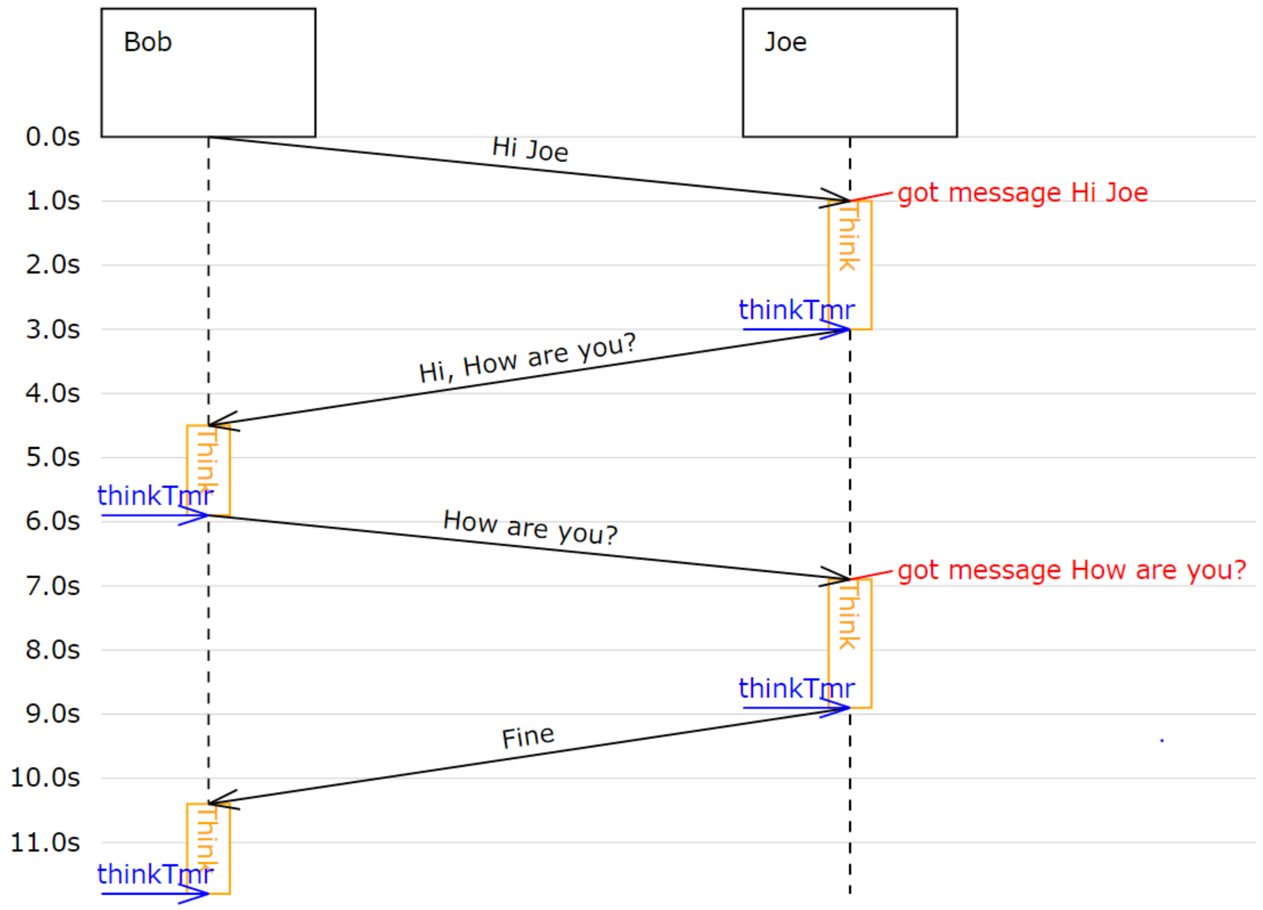
1.3.6.1 Sequence Diagram

Moddy can generate a sequence diagram from the simulation results. The sequence diagram is generated as a scalable vector graphics (SVG), either pure or embedded in HTML.

After simulation run, the following code generates the sequence diagram:

```
    moddyGenerateSequenceDiagram( sim=simu,
                                  fileName="1_hello.html",
                                  fmt="svgInHtml",
                                  excludedElementList=[],
                                  timePerDiv = 1.0,
                                  pixPerDiv = 30)
```

This is the result:



Notes:

- The black arrows are messages
- The blue arrows are timer expiration events
- The orange boxes are visualized "states" or "activities" that were generated by the model via *setStatusIndication()*.
- The red messages are annotations that were generated by the model via *addAnnotation()*

1.3.6.2 Trace Table

Moddy can also generate a table in form of a .csv (comma separated value file). This is how to generate it:

```
moddyGenerateTraceTable(simu, '1_hello.csv' )
```

This would be the result when open in Excel:

#time	Action	Object	Port/Tmr	Value	requestTime	startTime	endTime	flightTime
0	>MSG	Bob	Bob.mouth(OutPort)	Hi Joe	0	0	1	1
1	<MSG	Bob	Joe.ears(InPort)	Hi Joe	0	0	1	1
1	ANN	Joe	Joe(Block)	got message Hi Joe				
1	T-START	Joe	Joe.thinkTmr(Timer)	2				
1	STA	Joe	Joe(Block)	Think				
3	T-EXP	Joe	Joe.thinkTmr(Timer)					
3	STA	Joe	Joe(Block)					
3	>MSG	Joe	Joe.mouth(OutPort)	Hi, How are you?	3	3	4,5	1,5
4,5	<MSG	Joe	Bob.ears(InPort)	Hi, How are you?	3	3	4,5	1,5
4,5	T-START	Bob	Bob.thinkTmr(Timer)	1,4				
4,5	STA	Bob	Bob(Block)	Think				
5,9	T-EXP	Bob	Bob.thinkTmr(Timer)					
5,9	STA	Bob	Bob(Block)					
5,9	>MSG	Bob	Bob.mouth(OutPort)	How are you?	5,9	5,9	6,9	1
6,9	<MSG	Bob	Joe.ears(InPort)	How are you?	5,9	5,9	6,9	1
6,9	ANN	Joe	Joe(Block)	got message How are you?				
6,9	T-START	Joe	Joe.thinkTmr(Timer)	2				
6,9	STA	Joe	Joe(Block)	Think				
8,9	T-EXP	Joe	Joe.thinkTmr(Timer)					
8,9	STA	Joe	Joe(Block)					
8,9	>MSG	Joe	Joe.mouth(OutPort)	Fine	8,9	8,9	10,4	1,5
10,4	<MSG	Joe	Bob.ears(InPort)	Fine	8,9	8,9	10,4	1,5
10,4	T-START	Bob	Bob.thinkTmr(Timer)	1,4				
10,4	STA	Bob	Bob(Block)	Think				
11,8	T-EXP	Bob	Bob.thinkTmr(Timer)					
11,8	STA	Bob	Bob(Block)					
11,8	>MSG	Bob	Bob.mouth(OutPort)	Hm?	11,8	11,8	12,8	1

2 Detailed User Guide

2.1 Core Simulator

2.1.1 Initializing the Simulator

The simulator is the first class you must instantiate. The simulator constructor has no arguments:

```
from moddy import *
simu = sim()
```

The *simu* variable has to be passed to the parts.

2.1.2 Simulation Time

The simulator time is the virtual time seen by the model while the simulation is running.

It has nothing to do with the real time of the computer running the simulation.

It is important to understand that the simulation time does NOT advance while the model is executing a callback function (like a message receive or timer expiration callback). This means all python statements in a callback routine are executed at the same simulation time.

2.1.2.1 Time Unit

Moddy's simulation time unit is seconds, i.e. the value 1.0 means one second. Because the simulator uses a floating point type for the simulation time, any time unit can be used by the model.

The globals "ms" (=1E-3), "us" (=1E-6) and "ns" (=1E-9) can be used conveniently to specify short times. Example:

```
n2.toArmPort.send('down', 1*ms)
```

The 1*ms passes the value (1*1E-3) = 0.001 to the send function.

2.1.2.2 Setting the Time Unit for Trace Outputs during Simulation

To define the time unit for simulation outputs, use

```
simu.setDisplayTimeUnit( unit )
```

where unit can be "s", "ms", "us" or "ns".

This setting affects only the trace outputs during the simulation run:

```
TRC:       8.9s STA     Joe(Block) //
TRC:       8.9s >MSG    Joe.mouth(OutPort) // req=8.9s beg=8.9s end=10.4s dur=1.5s msg=[Fine]
```

It does not affect the Sequence diagram time unit or the csv table output

2.1.2.3 Current Simulation Time

The simulator provides a *time()* method which can be used by parts to determine the current simulation time. The time returned is the current simulation time in seconds.

Within a part you access the *time()* function through

```
self._sim.time()
```

2.1.3 Parts

Parts are the component of the system that communicate to each other. They are the base elements to form the structure of a model.

You create a part by creating an instance of a class derived from *simPart*. This derived class contains the model code implementing the behavior of the part:

- The parts initialization code
- The port receive methods
- The timer expired methods
- Optionally: Methods that are executed at the beginning and the end of the simulation (better place it in method *startSim()*)

Here is a simple example of a part class derived from *simPart*.

For Python beginners: The "self" variable that is used in ever method is the reference to the own part (comparable with "this" in C++).

```
from moddy import *

class Bob(simPart):
    def __init__(self, sim, objName):
        # Initialize the parent class
        super().__init__(sim=sim, objName=objName)

        # Ports
        self.createPorts('in', ['ears'])
        self.createPorts('out', ['mouth'])

        # Timers
        self.createTimers(['thinkTmr'])
        self.reply = ""

    def earsRecv(self, port, msg):
        if msg == "Hi, How are you?":
            self.reply = "How are you?"
        else:
            self.reply = "Hm?"

        self.thinkTmr.start(1.4)
        self.setStateIndicator("Think")


    def thinkTmrExpired(self, timer):
        self.setStateIndicator("")
        self.mouth.send(self.reply, 1)
```

2.1.3.1 simPart Constructor Parameters

Each part must call the simPart's constructor in it's *__init__* method.

```
class simPart(simBaseElement):
    """Simulator block"""
    def __init__(self, sim, objName, parentObj = None ):
```

The parameters are:

- sim: The simulator instance
- objName: The name of the part (for example "engine")
- parentObj: The parent part (for example the "car" if the current part is "engine"). If this parameter is omitted, "no parent", i.e. top level part is assumed.

Example:

```
class Bob(simPart):
    def __init__(self, sim, objName):
        # Initialize the base class
        super().__init__(sim=sim, objName=objName)
```

2.1.3.2 simPart Methods Called on Start and End of Simulation

Optionally, a part may define methods that are called at the start or end of simulation.

```
    def startSim(self):
        '''Called from simulator when simulation begins'''

    def terminateSim(self):
        '''Called from simulator when simulation stops.
           Terminate block (e.g. stop threads)'''
```

If present, the *startStim()* method is called at the start of the simulation (i.e. at simulation time 0). The simulator calls the *startSim()* method of all parts at the beginning of its *run()* method, in the order the parts have been created. The typical actions in the *startSim()* routine are

- starting timers
- sending initial messages

If present, the *terminateSim()* method is called by the simulator when the simulation is terminated, in the order the parts have been created. Typical actions of *terminateSim()* are

- stopping threads
- closing files

2.1.3.3 Nested Parts

You can model parts that are composed of other parts. For example, the part “engine” may be part of the part “car”.

To model this, you use the simPart constructors “parentObj” argument.

```
class Engine(simPart):
    def __init__(self, sim, objName, car):
        # Initialize the base class
        super().__init__(sim=sim, objName=objName, parentObj=car)
        …

class Car(simPart):
    def __init__(self, sim, objName):
        # Initialize the base class
        super().__init__(sim=sim, objName=objName)
        self.engine = Engine(sim, 'engine', self)
```

Note: The part hierarchy has no relevance for the simulator. The part hierarchy however is displayed in the structure graph, trace output and in the sequence diagrams.

2.1.4 Message Communication

Parts can communicate only via messages that are sent from an output port to an input port. More about messages in chapter 2.1.4.5 “Messages”.

Output ports can only send messages, they cannot receive.

Input ports can only receive messages, they cannot send.

There is also an IO Port, but this is nothing else as an object containing one input and one output port.

In general, a message that is sent via an output port, is received after the “flight Time” at the input port. The flight time simulates the transmission time of the message.

2.1.4.1 Creating Ports

All ports of a part must be explicitly created by a part. This is usually done in the constructor (*__init__* method) of the part owning the port. Note that a port is always owned by exactly one part.

There are two ways to create ports:

Using the low level methods:

- *newInPortI()*
- *newOutPort()*
- *newIOPort()*

Example:

```
class Bob(simPart):
    def __init__(self, sim, objName):
        ...
        self.ears = self.newInputPort( ‘ears’, self.earsRecv )
```

This creates a new input port with the name ‘ears’ and assigns the method *earsRecv()* as the callback method. The resulting port object is assigned to the part variable *ears*.

The creation of an IO port is similar:

```
        self.myIOPort1 = self.newIOPort( ‘ioPort1’, self.ioPort1Recv )
```

An output port has no receive callback method:

```
        self.myOutPort1 = self.newOutoutPort( ‘outPort1’ )
```

To reduce the amount of typing, you can call the higher level function *createPorts()*.

```
class Bob(simPart):
    def __init__(self, sim, objName):
        ...
        self.createPorts('in', ['ears'])
```

This essentially does exactly the same as the low level function above. It creates a new port object, creates a new part variable *self.ears* and assigns the callback method *earsRecv*.

This means that your callback function MUST always be named *<portName>Recv*.

You can also create multiple ports with one call:

```
        self.createPorts('in', ['inPort1', 'inPort2', 'inPort3'])
```

Output ports and IO ports can be created with the same method:

```
        self.createPorts('out', ['outPort1', 'outPort2', 'outPort3'])
        self.createPorts('io', ['ioPort1', 'ioPort2', 'ioPort3'])
```

Usually, you will always use the high level method, unless you need to create several input ports that have the same receive method.

2.1.4.2 Binding Ports

Before a message can be sent between parts, someone must bind the output port of one object to the input port of another object.

Note: Input and output ports that shall be bound must belong to different parts!

This binding is done normally by the main program. If you have parts that are composed of other parts, then the internal bindings within the top level part are done in the top level parts constructor.

To bind an input port to an output port, you call the output ports bind method:

```
    bob.mouth.bind(joe.ears)
```

You can bind several input ports to one output board to simulate multicast transfer.

```
    bob.mouth.bind(joe.ears)
    bob.mouth.bind(john.ears)
    bob.mouth.bind(paul.ears)
```

You cannot bind several output ports to one input port!

You can also bind IO ports to each other. In this case the output port of the first IO port is bound to the input port of the second port and vice versa. It doesn't matter on which IO port you call the bind method.

```
class myPart(simPart):
    def __init__(self, sim, objName):
        ...
        self.createPorts('io', ['ioPort1'])


part1 = myPart( simu, ‘part1’ )
part2 = myPart( simu, ‘part2’ )
part1.ioPort1.bind( part2.ioPort1 )
```

If you need to bind an IO port to a normal input and normal output port, you can do it like this:

Consider part1 has an ioport called "ioPort1" and part2 has one input port "inPort1" and an output port "outPort1":

```
      part1.ioPort1._outPort.bind( part2.inPort1 )
      part2.outPort1.bind( part1.ioPort1._inPort )
```

2.1.4.3 Sending Messages

A message between two parts is sent via an output port's send routine:

```
      send( msg, flightTime )
```

Where

- msg is the message you want to send (More about messages in chapter 2.1.4.5 "Messages")
- flightTime is the transmission time of the message; i.e. how long it takes until the message arrives at the input port. flightTime must be a positive value in seconds. flightTime can be 0; in this case, the message arrives without delay at the bound input ports.

The send method has no return value.

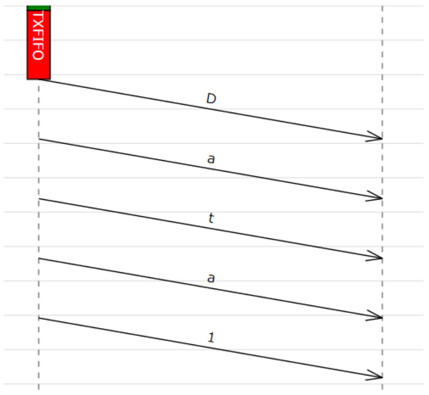
What happens if you call the send method on a port which is already sending a message (meaning: the flight time of one or more previous messages has not elapsed)?

In this case, the output port queues the pending messages one after each other. When a messages flight time has elapsed, the next message from the queue is sent. This simulates the behavior of a serial transmission. See the following snapshot from the tutorial "2_sergw".

Here, multiple send() calls are issued at the same simulation time.

```
                self.busy( len(msg) * 20*us, 'TXFIFO', whiteOnRed)
                # push to serial port
                for c in msg:
                    self.serPort.send( c, serFlightTime(c))
```

This results in the following sequence diagram. You see that the next character is fired when the previous character transmission ends:



2.1.4.4 Receiving Messages

When a message is received on an input port, the input ports callback method is called. This receive method must be provided by the model, usually in the class derived from *simPart*.

When you have created the port with *createPorts()*, the method is called *<portName>Recv*:

```
class Bob(simPart):
    def __init__(self, sim, objName):
        ...
        self.createPorts('in', ['ears'])

    def earsRecv(self, port, msg):
        if msg == "Hi, How are you?":
            self.reply = "How are you?"
        else:
            self.reply = "Hm?"

        self.thinkTmr.start(1.4)
```

The receive callback method gets passed two parameters:

- **port** is the input port on which the message was received. You can use this to find out which port received the message in case you have assigned the same receive method to multiple ports
- **msg** is the message just received

The receive callback method does not return a value.

Note that you get a copy of the message sent by the caller, so you can modify or even delete the message content without affecting the sender or other receivers of the message. More specifically, msg is a “deep copy” of the original message; this means that also objects that are referenced in the message are copied.

The usual task of receive callback method is to start timers or to send messages to other objects.

2.1.4.5 Messages

What type of data can be transferred between output and input ports?

Generally, any valid python object can be a Moddy message, such as:

- Numbers, e.g. 1.0
- Strings, e.g. ‘abc’
- Lists, e.g. [‘abc’, 1.0, 2]
- User defined classes

Moddy does not force any specific type to be used as a message. However, the model must be written in a way that the receiver understands the messages the sender might generate.

Here is an example of a user defined message. It simulates the behaviour of “Fail Safe over EtherCAT” (it does not really implement all fields of FSoE, it only includes the necessary information needed for that model):

```
class FsoeMsg:
    def __init__(self, addr, seq, data ):
        self.addr = addr    # FSoE Address
        self.seq = seq      # sequence number
        self.data = data    # FSoE Payload
```

To send such a message, you could call from a part’s method:

```
        self.outPort1.send( FsoeMsg(self.fsoeAddr, seq, ‘TESTDATA’, msgFlightTime )
```

Restriction: Do NOT include a reference to the simulator instance, parts, ports or timers into the user defined messages! This may cause an exception when sending such messages (because it causes endless recursion while trying to make a deep copy of the message).

2.1.4.5.1 Message Content Display

How are message contents displayed in sequence diagrams and trace tables?

Moddy calls the object’s *__str__()* method. This method should generate a user readable string with the message content.

For the built-in classes, python defines the *__str__()* method. For user defined classes, you must implement it:

```
class FsoeMsg:
    ...
    def __str__(self):
        return "FSoE @%d#%d %s" % (self.addr, self.seq, self.data)
```

Example output: “FSOE @2#3 ‘ABC’” -> meaning: FSOE message to slave 2, sequence 3, with data ‘ABC’.

2.1.4.5.2 Checking Message Types

If the receiver wants to check if the received message is of the correct type, he can use the python *type()* method, here are some examples:

```
if type(msg) is int:
   …
if type(msg) is float:
   …
if type(msg) is str:
   …
if type(msg) is list:
   …
if type(msg) is FsoeMsg:
   …
```

2.1.4.5.3 Message Colors

You can influence the color in which messages are displayed in the sequence diagram.

By default, the messages are drawn in "black".

You can assign a color to an output port, e.g.

```
myOutPort.setColor("green")
```

In this case, all messages sent via this output ports are drawn in green.

You can even assign different colors to individual messages. To do so, create a member *msgColor* inside a (user defined) message:

```
class FsoeMsg:
    def __init__(self, addr, seq, data, msgColor=None ):
        self.addr = addr    # FSoE Address
        self.seq = seq      # sequence number
        self.data = data    # FSoE Payload
        if msgColor is not None: self.msgColor = msgColor
```

If the *msgColor* member exists, this message will get the define *msgColor.*, overriding the color which might have been assigned to the port.

2.1.4.5.4 Simulating Lost Messages

You can force messages to be lost to simulate disturbed communication.

Therefore, output port and I/O Ports provide an API to inject a "message lost error". If you call the *injectLostMessageErrorBySequence()* method of an output port you can force one or more of the following messages sent on this port to be lost. *injectLostMessageErrorBySequence(0)* will force the next message sent on that port to be lost, *injectLostMessageErrorBySequence(1)* the next but one message and so forth.

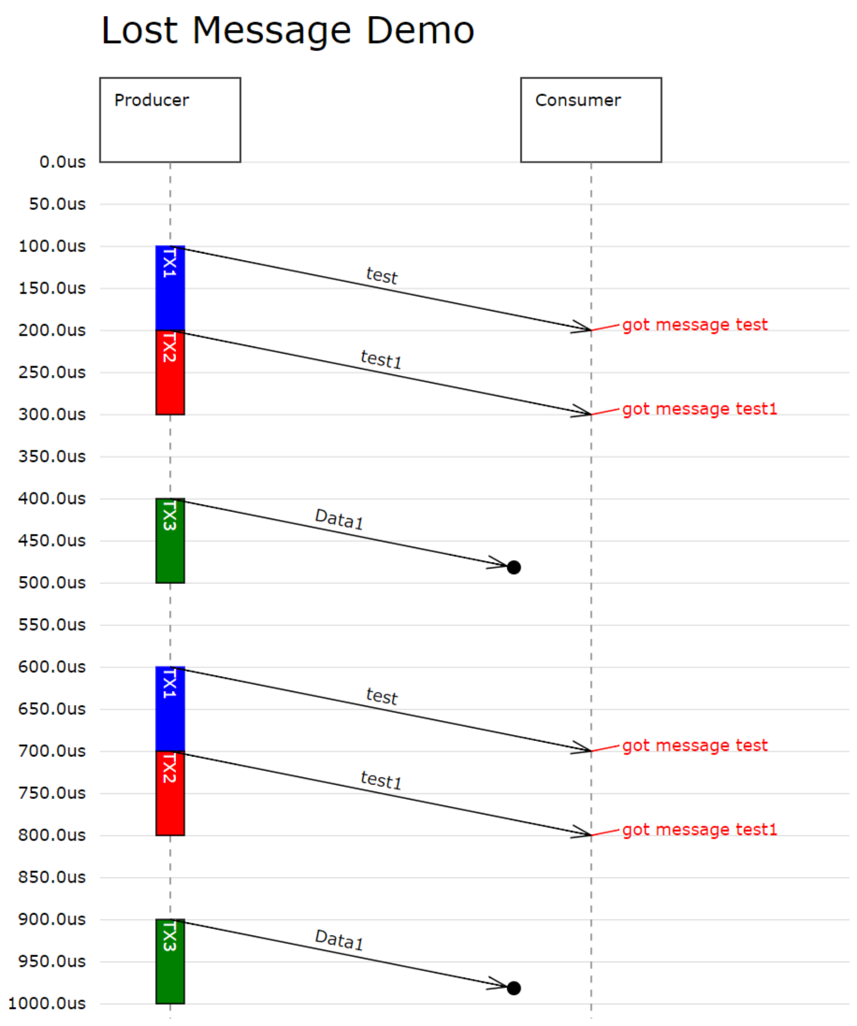
Lost messages will not arrive at the input port.

For example:

```
class Producer(vSimpleProg):
    def __init__(self, sim):
        super().__init__(sim=sim, objName="Producer", parentObj=None)
        self.createPorts('out', ['netPort'])

    def runVThread(self):
        self.netPort.injectLostMessageErrorBySequence(2)
        self.netPort.injectLostMessageErrorBySequence(5)
        self.netPort.injectLostMessageErrorBySequence(6)
        while True:
            self.wait(100*us)
            self.netPort.send('test', 100*us)
            self.busy(100*us, 'TX1', whiteOnBlue)
            self.netPort.send('test1', 100*us)
            self.busy(100*us, 'TX2', whiteOnRed)
            self.wait(100*us)
            self.netPort.send('Data1', 100*us)
            self.busy(100*us, 'TX3', whiteOnGreen)
```

will force the 3rd, 6th and 7th message on the *netPort* to be lost.

In the sequence diagrams, lost messages are indicated by a bubble in front of the message arrow:



In the simulator trace output, a lost message is shown as a normal message reception event, but with the additional "(LOST)" string:

```
TRC:    500.0us <MSG    Consumer.netPort(InPort) // (LOST) req=400.0us beg=400.0us
end=500.0us dur=100.0us msg=[Data1]
```

2.1.5 Timers

Timers are – beside messages - another way of triggering actions in a part.

Typically, timers are used to

- Trigger periodic actions
- Provide timeout for message reception

A timer is started and stopped from a part's methods. When the timer expires, an "expiration callback" method is called within the part-

2.1.5.1 Creating Timers

A part can have any number of timers.

All timers of a part must be explicitly created by a part. This is usually done in the constructor (*__init__* method) of the part owning the timer. There are two ways to create timers:

Using the low level method *newTimer()*:

Example:

```
class Bob(simPart):
    def __init__(self, sim, objName):
        ...
        self.thinkTmr = self.newTimer( ‘thinkTmr’, self.thinkTmrExpired )
```

This creates a new timer with the name *thinkTmr* and assigns the method *thinkTmrExpired()* as the callback method. The resulting timer object is assigned to the part variable *thinkTmr*.

To reduce the amount of typing, you can call the higher level function *createTimers()*.

```
class Bob(simPart):
    def __init__(self, sim, objName):
        ...
        self.createTimers(['thinkTmr'])
```

This essentially does exactly the same as the low level function above. It creates a new timer object, creates a new part variable *self.thinkTmr* and assigns the callback method *thinkTmrExpired*.

This means that your callback function MUST always be named *<timerName>Expired*.

You can also create multiple timers with one call:

```
        self.createTimers(['tmr1', 'tmr2'])
```

2.1.5.2 Starting and Stopping Timers

Each timer has two states: Started and Stopped.

You start a timer with *start()* or *restart()*.

```
      self.thinkTmr.start(2)

      self.thinkTmr.restart(2)
```

Both expect the timer’s expiration time, relative to the current simulation time, in seconds (in the examples above: 2 seconds). They do not return a value.

The *start()* method throws an exception if you use it on a timer which is already started. The *restart()* method starts a timer with the specified time regardless of the timers state. Both methods bring the timer into the started state.

You stop a timer with *stop()* method. It brings the timer into the stopped state. In other words, you cancel the timer.

```
      self.thinkTmr.stop()
```

2.1.5.3 Timer Expiration Callback

When a timer expires, the timer's callback method is called. This callback method must be provided by the model, usually in the class derived from *simPart.*

When you have created the port with *createTimers()*, the method is called <*timerName*>*Expired*:

```
class Bob(simPart):
    def __init__(self, sim, objName):
        ...
        self.createTimers(['thinkTmr'])

    def thinkTmrExpired(self, timer):
        self.mouth.send(self.reply, 1)
```

The timer expired callback gets passed the *timer* parameter, which is the timer object that has expired. You can use this to find out which timer expired in case you have assigned the same callback methods to multiple timers.

The callback method does not return a value.

The usual task of receive callback method is to re-start this timer, start other timers or to send messages to other objects.

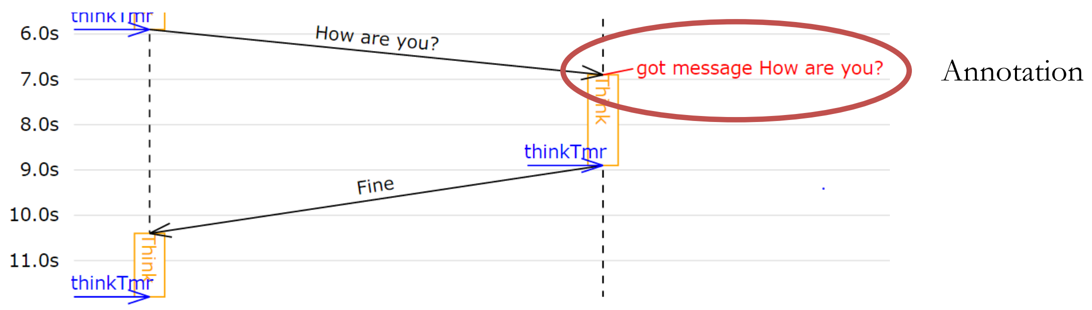
2.1.6 Annotations

The model can add annotations to the output (i.e. sequence diagrams and trace tables) to visualize special events in the model.

You add an annotation by calling the simPart's *addAnnotation()* method:

```
class Joe(simPart):

    def earsRecv(self, port, msg):
        self.addAnnotation('got message ' + msg)
```

In a sequence diagram, an annotation is displayed on the part's life line at the current simulation time:



The *addAnnotation()* method expects a string as its argument. It must be a single-line string. No special characters such as newline are allowed.

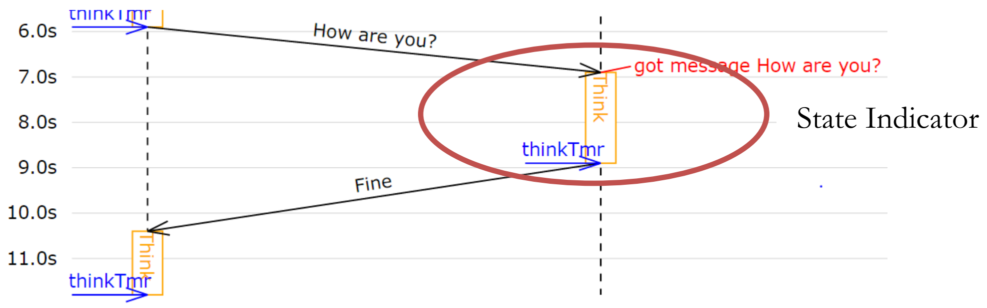
2.1.7 State Indications

To visualize a part's state or to indicate which activity the part is currently performing, you can use state indications. For this, you call the part's *setStateIndicator()* method:

```
class Bob(simPart):
    …
    def earsRecv(self, port, msg):
        ...
        self.setStateIndicator("Think")


    def thinkTmrExpired(self, timer):
        self.setStateIndicator("")
```

The first parameter to setStateIndicator() is *text*:

- A non-empty string indicates the start of a new state or activity. The text is displayed in sequence diagrams in vertical direction.
- An empty string ends the state or activity



The second, optional parameter to setStateIndicator is *appearance*. With this parameter, you can control the colors of state indicator. If present, it must be a python dictionary like this:

```
    {'boxStrokeColor':'black', 'boxFillColor':'green', 'textColor':'white'}
```

Where:

- *boxStrokeColor:* is the color of the status box's border
- *boxFillColor:* is the color of box body
- *textColor:* is the color of the text

Color names are according to the SVG specification, see http://www.december.com/html/spec/colorsvg.html for an overview.

If the appearance argument is omitted, it defaults to:

```
    {'boxStrokeColor':'orange', 'boxFillColor':'white', 'textColor':'orange'}
```

2.1.8 Watching Variables

Sometimes it is useful to watch the value of variables and how they are changing during simulation.

Tell Moddy which variables to watch. You do this by calling the *newVarWatcher()* method of a moddy part:

```
class VarChanger(vSimpleProg):
    def __init__(self, sim):
        super().__init__(sim=sim, objName="VC", parentObj=None)
        self.var1 = 0
        # self.var2 is created during execution

    def runVThread(self):
        while True:
            self.var1 = 1
            self.wait(2)

            self.var1 = 2
            self.var2 = "ABC"

if __name__ == '__main__':
    simu = sim()

    vc = VarChanger(simu) # create the moddy part
    var1watch = vc.newVarWatcher('var1', "0x%08x")
    var2watch = vc.newVarWatcher('var2', "%s")
```

newVarWatcher expects as the first parameter the name of the variable to watch as a string, e.g. “var1”, or “subobj.var2” etc. The variable doesn’t need to exist yet. As shown in the above example for *var2*, it can be created during runtime.

The second parameter to *newVarWatcher* is the format string that is used to format the variables value, e.g. "0x%08x".

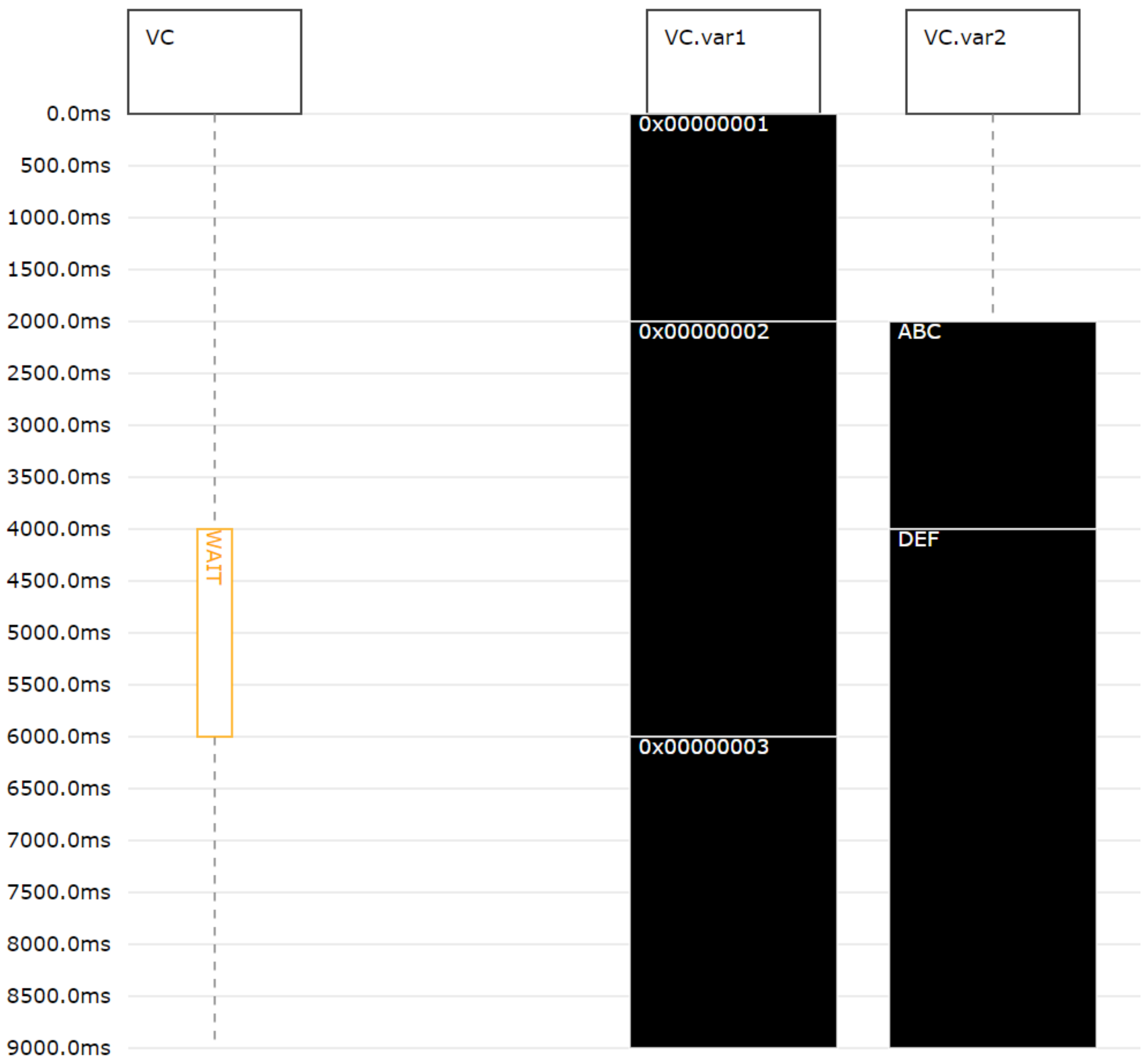
When you run now the simulator, you will see a “VC” event every time the value of a watched variable changes:

```
0.0s VC      VC.var1(WatchedVar) // 0x00000001
```

Watched variables can be included in the sequence diagram. In the following examples, you see the status of a Moddy part *VC*, and its two internal variables *var1* and *var2*. Variable traces will be shown always on the right side of the sequence diagram.

You define with the parameter *showVarList* which variable traces are shown in the sequence diagram:

```
    moddyGenerateSequenceDiagram( sim=simu,
                                  fileName="4_varwatch.html",
                                  fmt="svgInHtml",
                                  showPartsList=['VC'],
                                  showVarList=['VC.var1', 'VC.var2'],
                                  excludedElementList=['allTimers'],
                                  timePerDiv = 0.5,
                                  pixPerDiv = 30)
```



2.1.9 Running the Simulation

After the instantiation of the simulator, the creation of the parts, ports and the binding of the ports, you can run the simulation:

```
# let simulator run
simu.run(stopTime=12.0, maxEvents=10000, enableTracePrinting=True)
```

This will run the simulation until one of the following conditions are met:

- *stopTime* has been reached
- The simulator has processed a maximum number of event *maxEvents*. If this argument is omitted, it defaults to 10000
- The simulator has no more events to execute

- An exception was raised (either by the model or the simulator)

With *enableTracePrinting*, you can control whether the simulator prints each event while executing.

The run method does not return a value.

2.1.9.1 Catching Model Exceptions

When the model code throws an exception, the simulator stops. If you want to generate the output files even in this case (to show the events that have been recorded until then), you can catch the simulators exception like this:

```
# let simulator run
    try:
        simu.run(stopTime=12*ms)

    except: raise
    finally:
        # create SVG drawing
        moddyGenerateSequenceDiagram( sim=simu,
                                      ...
```

2.2 Modelling with Finite State Machines

Instead of using the event-callback based mechanism described in the previous chapter, it is sometimes easier to describe your model with finite state machines (FSMs).

In moddy, FSMs are defined using the python language directly in your model code by creating a class derived from the Moddy *Fsm* class.

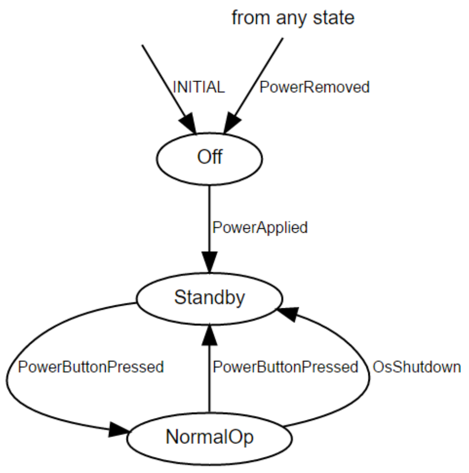
Moddy FSMs features:

- Simple FSM specification using a python dictionary
- Simple interface to simulator events (Message reception and timer events)
- Generation of graphical representation of the state machine
- FSM-events trigger transitions between states
- Action-callbacks for state-entry, state-exit, state-do
- Support for nested state machines, including orthogonal (parallel) nested state machines

Intentionally NOT supported are actions on transitions, because they make FSMs more difficult to read.

2.2.1 Specifying the FSM

Let's begin with a simple FSM, describing the states of a (simplified) computer:



Note that this figure is generated by Moddy using *moddyGenerateFsmGraph()* described in ???

In moddy, this state machine can be expressed as follows:

```
class Computer(Fsm):

    def __init__(self, parentFsm=None):

        transitions = {
            '':
                [('INITIAL', 'Off')],
            'Off':
                [('PowerApplied', 'Standby')],
            'Standby':
                [('PowerButtonPressed', 'NormalOp')],
            'NormalOp':
                [('PowerButtonPressed', 'Standby'),
                 ('OsShutdown', 'Standby')],
            'ANY':
                [('PowerRemoved', 'Off')]
        }

        super().__init__( dictTransitions=transitions, parentFsm=parentFsm )
```

The state and the transitions are specified as a python dictionary:

`{ state1: [transition1, transition2, …], state2: [transition1, transition2, …], … }`

The dictionary keys are the states. In the example above, the state machines regular states are 'Off', 'Standby' and 'NormalOp' (the two special states '' and 'ANY' are described below).

Each dictionary element's value is a list of transitions valid in the specified state. Each transition is specified as a tuple:

`(event, target-state)`

In the example above, the `('PowerApplied', 'Standby')` transition is valid in the 'Off' state. It defines that the 'PowerApplied' event leads to state 'Standby'

There are two special states:

- The start state is specified with an empty string. When the state machine is started, it automatically executes the "INITIAL" event, which transits to the initial state ('Off' in our example).
- Transitions defined in the pseudo state 'ANY' are valid in any state. In our example, the 'PowerRemoved' event leads to 'Off' state, regardless of the current FSM state.

2.2.2 Generating FSM Events

Before you can work with an FSM, you must instantiate and start it:

```
comp = Computer()

comp.startFsm()
```

The *startFsm()* method issues automatically the 'INITIAL' event and transits to the initial state.

To force transitions of the FSM, your model generates FSM events using the *event()* method. The current FSM state can be read from the *state* member:

```
comp.event('PowerApplied')
print("State %s" % comp.state)
comp.event('PowerButtonPressed')
print("State %s" % comp.state)
comp.event('PowerRemoved')
print("State %s" % comp.state)
```

This produces the following output:

```
State Standby

State NormalOp

State Off
```

Note that the *event()* method return value tells you if the event caused a state change (True) or not (False).

2.2.3 Action Callbacks

You can define callbacks that are executed when

- A state is entered (Entry Action)
- A state is left (Exit Action)
- A state is entered or a self-transition is execute (Do Action)

These callbacks are optional. They are only executed when the corresponding methods are defined in the FSM class.

The callbacks must be named according to the following convention:

State_<state>_<*action*>

These callbacks have no parameters.

Example:

```
class Computer(Fsm):
        ...
        def State_Off_Entry(self):
            print("State_Off_Entry")

        def State_Off_Exit(self):
            print("State_Off_Exit")
```

You can also define callbacks that are executed in any state:

```
        def State_ANY_Entry(self):
            print("Any State Entry")
```

2.2.3.1 The Do Action

The Do Action is executed when

- A state is entered, directly after the Entry Action
- On a state self-transition

A state self-transition is a transition whose source and target state are identical. In this case, the Exit and Entry Actions are NOT executed, but the Do Action is executed.

2.2.3.2 Executing Application Specific, State-Dependent Callbacks

The mechanisms to execute the Entry/Exit/Do Actions can be used also directly for application specific purposes.

Consider the following situation: The object implementing the FSM should receive data from another object, but how the data is processed, depends on the FSMs state.

For this purpose, the FSM class provides *execStateDependentMethod()*.

Example: The “Computer” shall receive data only when it is in “NormalOp” State:

```
class Computer(Fsm):

         ...

        def State_NormalOp_ReceiveData(self, data):
            print("NormalOp_ReceiveData", data)
        def State_Standby_ReceiveData(self, data):
            print("Standby_ReceiveData", data)
```

Now, when you execute

```
comp.execStateDependentMethod( ‘ReceiveData’, True, data=‘123’ )
```

the State_<*state*>_ReceiveData function with the argument data=‘123’ of the corresponding state is executed (or none if no such method exists for the current state).

The detailed signature is

```
methodWasCalled = execStateDependentMethod(methodName, deep, *args, **kwargs):
```

where

- *methodName* is the method name to call.

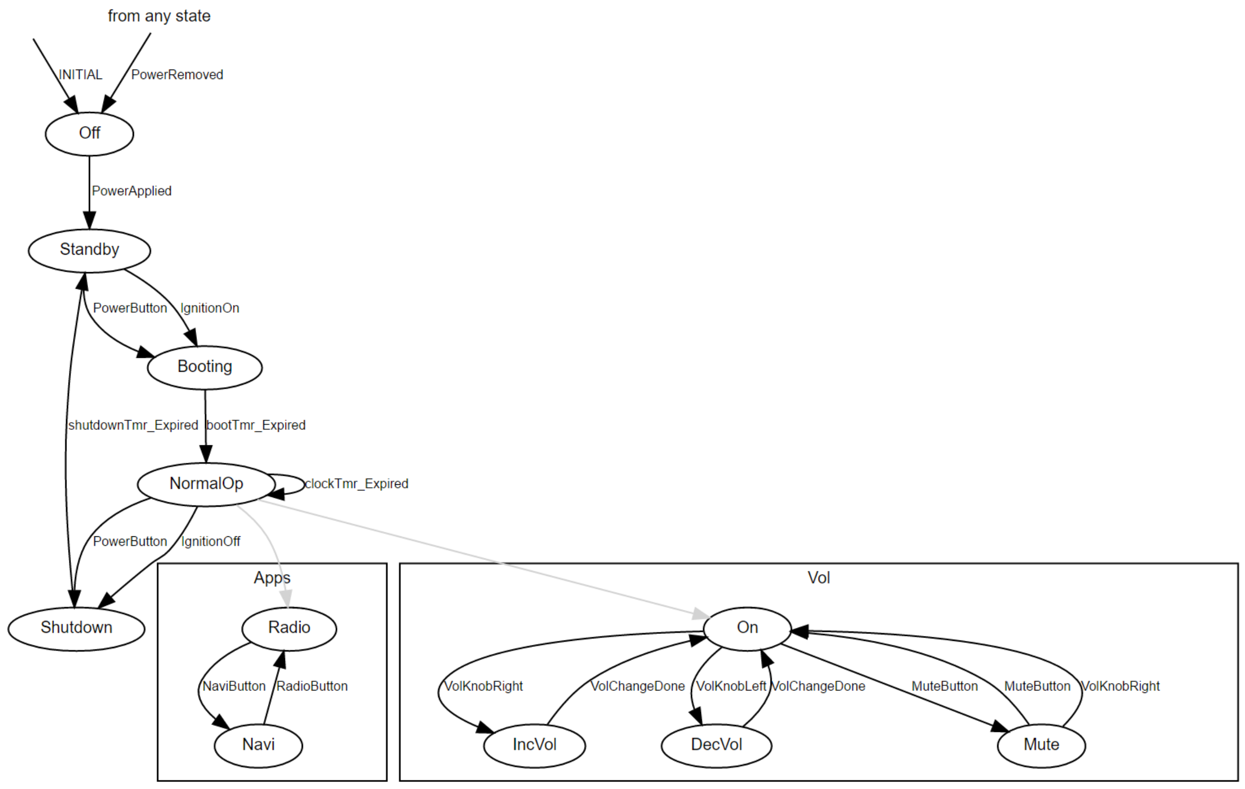
    - First 'State_ANY_<*methodName*>' is called (if it exists), then
    - 'State_<*state*>_<*methodName*>' is called (if it exists) where state is the current FSM state.
- If *deep* is True, the call is also propagated to all currently active sub-state machines, before the calls to the top level state machine
- **args* and **kwargs* are the positional and named arguments to pass to the method
- The return value is True if at least one method could be called

2.2.4 Nested FSMs

Nested FSMs are needed if one or more of the FSMs states contains sub-states.

The following example shows a state machine with two sub-state machines. This example can be found in the moddy tutorial *3_carinfo.py*. It simulates the behaviour of a simplified car infotainment system. The main state machine models the devices global state. The "NormalOp" states has two substates: The first is "Apps" which switches between the Radio Application and the Navigation Application. The second is "Vol" which controls the audio volume based on button events. The two sub state machines are both active at the same time (as long as the main state machine is in the "NormalOp" state.

Note that the graphical representation of sub-states in Moddy is a little unconventional, mainly due to the limitations of the GraphViz tool. The sub-states are drawn in separated container boxes, whereas they are normally drawn inside the containing super-state.



To specify sub-states in a FSM, specify the sub-states name and the sub-state class as follows:

```
class CarInfoSystem(simFsmPart):
```

```
...
class FSM(Fsm):
    def __init__(self):

        transitions = {
                [('INITIAL', 'Off')],
            'Off':
                [('PowerApplied', 'Standby')],
            'Standby':
                [('PowerButton', 'Booting'),
                 ('IgnitionOn', 'Booting')],
            'Booting':
                [('bootTmr_Expired', 'NormalOp')],
            'NormalOp':
                 # The following two lines specify nested state machines,
                 # executing in parallel
                [('Apps' , CarInfoSystem.FSM.ApplicationsFsm ),
                 ('Vol' , CarInfoSystem.FSM.VolumeFsm ),
                 # Normal tranisitions
                 ('PowerButton', 'Shutdown'),
                 # ...
        }

        super().__init__( dictTransitions=transitions )

    # Nested state machine CarInfo System Applications
    class ApplicationsFsm(Fsm):

        def __init__(self, parentFsm):

            transitions = {
                '':
                    [('INITIAL', 'Radio')],
                'Radio':
                    [('NaviButton', 'Navi')],
                'Navi':
                    [('RadioButton', 'Radio')]
            }

            super().__init__( dictTransitions=transitions, parentFsm=parentFsm )

    …
```

This means sub-states are defined in the list of transitions. If the Fsm class code detects that a class reference is given instead a string with a target state, it treats this as a sub-state machine specification.

2.2.4.1 Sub-State Machine Instantiation

A sub-state machine object is created when the super-state is entered. (In our example above, the ApplicationFsm and VolumeFsm objects are created when the super-state 'NormalOp' is entered). The sub-state machine then automatically execute the 'INITIAL' event to transit to their initial state.

2.2.4.2 Sub-State Machine Termination

A sub-state machine object is terminated when the super-state exits.

When the super-state exits, the Exit action of the current states are called: First, the Exit Action of the current state in the sub state machines, and then the Exit Action of the current state of the super-state machine. Then the sub-state machine objects are deleted.

2.2.4.3 Sub-State Machine Event Reception

All currently active sub-state machines receive the events that are generated at the super-FSM.

The super-FSM first forwards any event to all its sub-state machines. Only if none of the sub-state machines "knows" the event, the event is processed at the super-FSM itself. A sub-state machine is said to "know" an event if the event is listed in the transitions list passed to the constructor (it doesn't matter in which state).

In the "carinfo" example, if the Main-FSM is in the 'NormalOp' state and receives the 'NaviButton' event, the event is processed only in the ApplicationFsm sub-state machine, because only this FSM knows the 'NaviButton' event.

However, if the Main-FSM is in the 'NormalOp' state and receives the 'PowerButton' event, the event is processed in the Main-FSM, because no sub-state machine knows this event.

2.2.4.4 Accessing Super-FSMs from Sub-State Machines

Sometimes sub-state machine watnt to send events to higher level state machines, to force a ransition at the higher level state machine.

A sub-state machine can get a reference to the next higher level FSM using the _parentFsm member:

```
self._parentFsm.event('MyEvent')
```

If you want to send an event to the highest level FSM, use the *topFsm()* method:

```
self.topFsm().event('MyEvent')
```

2.2.5 Modelling a Moddy Part with a Finite State Machine

Moddy provides a convenience class *simFsmPart* to specify a Moddy Part with a FSM.

The *simFsmPart* class provides the following features:

- The FSMs state is represented on the sequence diagram's life line via the status box
- Messages received on input ports and timer expiration events can generate FSM events

2.2.5.1 Creating a simFsmPart

To create a part that is controlled by a FSM, create a sub-class of *simFsmPart*. You see in the example below that you need to create a nested class containing the FSM (here called class "FSM") and pass an instance of this class to the *simFsmPart's* constructor:

```
from moddy import *

class CarInfoSystem(simFsmPart):

    def __init__(self, sim, objName):
        statusBoxReprMap = {
             ...
        }


        super().__init__(sim=sim, objName=objName, fsm=self.FSM(),
             statusBoxReprMap=statusBoxReprMap)

        # Ports & Timers
        self.createPorts('in', ['powerPort', 'ignitionPort', 'buttonPort'])
        self.createPorts('out', ['audioPort', 'visualPort'])
        self.createTimers(['bootTmr', 'shutdownTmr', 'clockTmr'])


    class FSM(Fsm):
        def __init__(self):

            transitions = {
                '': # FSM uninitialized
                    [('INITIAL', 'Off')],
                'Off':
                   ...
```

2.2.5.2 Representation of the FSM State on the Life Line

An instance of *simFsmPart* automatically shows its FSM state on the part's life line.

Restriction: The life line cannot show states of nested FSMs, only the state of the highest level FSM can be shown.

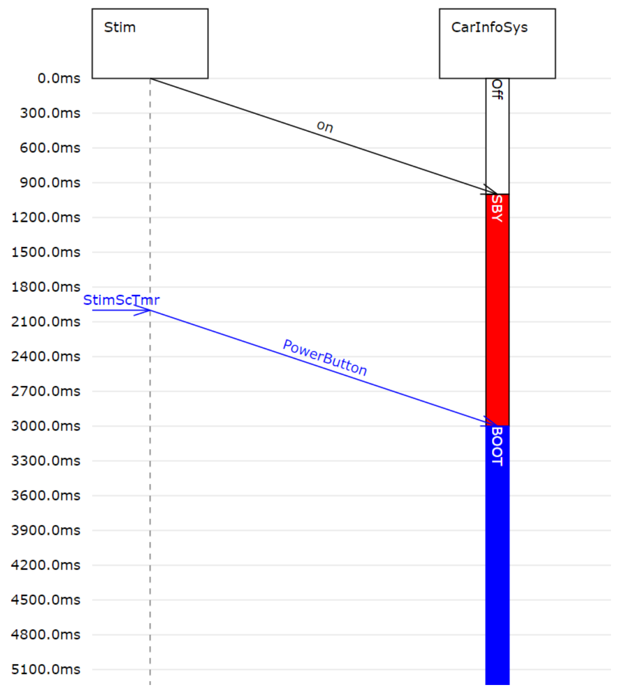
To define how the FSM state are represented on the part's life line, you define for each state the text and the colours:

```
whiteOnGreen = {'boxStrokeColor':'black', 'boxFillColor':'green',
                'textColor':'white'}
whiteOnRed = {'boxStrokeColor':'black', 'boxFillColor':'red', 'textColor':'white'}
whiteOnBlue = {'boxStrokeColor':'blue', 'boxFillColor':'blue', 'textColor':'white'}
blackOnWhite = {'boxStrokeColor':'black', 'boxFillColor':'white',
                'textColor':'black'}

class CarInfoSystem(simFsmPart):

    def __init__(self, sim, objName):


        statusBoxReprMap = {
            'Off':      (None, blackOnWhite),
            'Standby':  ('SBY', whiteOnRed),
            'Booting':  ('BOOT', whiteOnBlue),
            'NormalOp': ('NORM', whiteOnGreen),
            'Shutdown':  ('SD', whiteOnRed)
        }
```

For each FSM state, a tuple (text, appearance) has to be defined.

If you specify the map to be None or if you don't specify a mapping for a specific state, then the text shown at the life line will be the state name and the status box colours will be the default colours.

Using the map shown above, will result in the following sequence diagram:



2.2.5.3 Message Handling

Now, how to connect moddy messages to FSMs?

If a Moddy message is received on an input port owned by a *simFsmPart*, it either

1) generates directly an FSM event or
2) executes a callback function, which can analyse the message and then generate an FSM event

Usually, you want use option 2), i.e. analyse the message first.

In this case, create a callback function for the port that will receive the message. E.g. if your port is called "ingitionPort" and you want to receive the messages in any state:

```
class FSM(Fsm):
    ...
    def State_ANY_ignitionPort_Msg(self, msg):
        if msg == 'on':
            self.event('IgnitionOn')
        elif msg == 'off':
                self.event('IgnitionOff')
```

So the callback function must be named always *State_<state>_<portName>_Msg*. It is passed the message *msg* which was received on the port.

However, if you don't need to analyse the message content, you can also let moddy generate directly an FSM event. For this, specify in your transitions list an event called *<portName>_Msg:*

```
class FSM(Fsm):
    def __init__(self):

        transitions = {
            ...
            'Standby':
                [('PowerButton', 'Booting'),
                 ('ignitionPort_Msg', 'Booting')],
```

If the *simFsmPart* logic recognizes that your FSM knows an event *<portName>_Msg*, it generates this event whenever a message is received on the port and it does NOT call the callback function *State_<state>_<portName>_Msg*.

2.2.5.4 Timer Handling

Timer handling in *simFsmParts* is very similar to message receiption. Also here, you can chose to let moddy directly generate an event when a timer expires, or to execute a callback function. But for timers, you normally let moddy directly generate events:

```
  class FSM(Fsm):
        def __init__(self):

            transitions = {
                ...
                'Booting':
                    [('bootTmr_Expired', 'NormalOp')],
            }
```

Here, the simFsmPart generates the *bootTmr_Expired* event, when the timer named "bootTmr" expires.

If you don't have the *bootTmr_Expired* event in your transitions list, simFsmPart will attempt to execute a callback function *State_<state>_<timerName>_Expired.*

2.3 Sequential Program Simulation

With the message and timer callbacks explained before, you can model already everything.

However, when you want to model a sequential execution, it can be quite cumbersome to do this with the event based mechanism. For example if you want to model the execution of a software program like this

- Get inputs (from Moddy messages), takes 100µs
- Calculate, takes 1ms
- Set outputs, takes 200µs

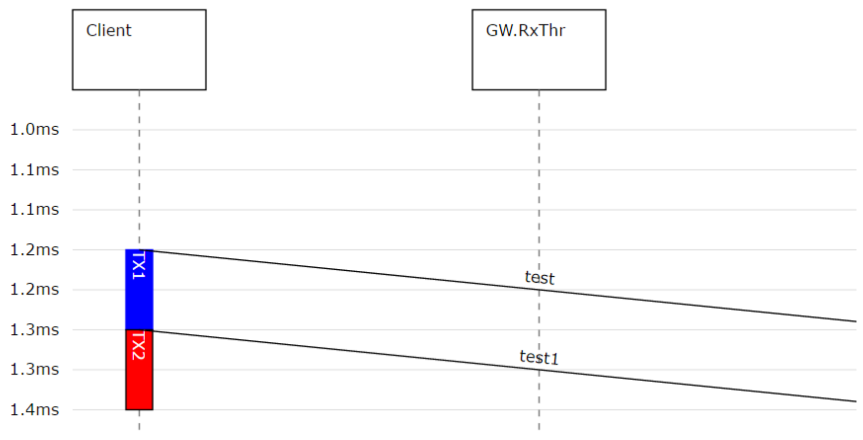
With the event based mechanism, you would need a state machine in the timer's callback routine, which is not easy to understand.

So Moddy allows an alternative way to model those sequential programs, simply called "Programs" in the following.

To create a "program", your Part must be a subclass of either *vThread* or *vSimpleProg*. (More about the difference between them in chapter 2.3.6).

2.3.1 Program Model

```
class Client(vSimpleProg):
    def __init__(self, sim):
        super().__init__(sim=sim, objName="Client", parentObj=None)
        self.createPorts('QueingIO', ['netPort'])

    def runVThread(self):
        while True:
            self.wait(1.2*ms)
            self.netPort.send('test', 100*us)
            self.busy(100*us, 'TX1', whiteOnBlue)
            self.netPort.send('test1', 100*us)
            self.busy(100*us, 'TX2', whiteOnRed)
```



In the above example, you see a very simple sequential program, that is

- Idle (waiting) for 1.2ms
- Performing a send "TX1" operation, which takes 100µs
- Performing a send "TX2" operation, which takes 100µs

Moddy parts derived from vThread or vSimpleProg must implement a method *runVThread*. This method contains your sequential program model.

It is called at the start of the simulation time (simulation time 0).

The *runVThread* method is not supposed to exit/return, that's why it contains an endless loop.

The *runVThread* method can control the timing of the program via "system calls":

- **wait()** – delays the program execution for a specified time or until an event occurred, indicating that the program is idle. No status box is shown on the sequence diagram life line while a program is waiting
- **busy()** – delays the program execution for the specified time. A user defined status box (e.g. 'TX1' in the example above occurs) is shown on the life line while the program is busy.

Note: Moddy executes each *runVThread()* method in a separate python thread. But don't worry: Race conditions resulting from concurrent execution cannot occur in Moddy, because Moddy executes exactly only one thread at each time, either the simulator thread or one of the *vThreads*. There is no need to protect your data via mutexes.

2.3.2 Communication between Moddy Programs and other Moddy Parts

Like normal Moddy parts, Moddy Programs shall communicate with other parts only via Moddy messages.

To send messages, a Moddy program can use standard output ports. There is no difference to other parts

But a standard input port executes a callback method. It cannot tell the program that a message has been received (other than setting a global variable). For this reason, Moddy provides "buffering input ports".

2.3.3 Buffering Input Ports

A buffering input port buffers received messages in the part's local memory and eventually wakes up the program.

There are two types of buffering ports

- **Sampling Port:** A sampling port is used if the receiver is only interested in the most recent message.
  - A sampling port buffers only the last received message.
  - A read from the sampling buffer does *not* consume the buffered message
- **Queing Port:** A queing port is used if the receiver wants to get all messages.
  - A queing port buffers all messages in a fifo queue. The queue depth is infinite.
  - A read from the buffer consumes the first message

Note: The term "Queing" is misspelled in this documentation and in the code. It should be "Queuing". For now, I leave it as it is.

A program reads a message from a buffering ports via the *readMsg()* method. It can check the number of messages in the buffer through the *nMsg()* method.

A program can wait for new message using the *wait()* method. The exact behavior depends on the type of buffer port (Sampling or Queing) and will be explained in the following.

2.3.3.1 Sampling Input Ports

Recall from previous chapter:

- A sampling port buffers only the last received message.
- A read from the sampling buffer does *not* consume the buffered message

A sampling input port is created with the *createPorts()* method, usually from the program's constructor:

```
self.createPorts('SamplingIn', ['inP1'])
```

The *readMsg()* method on a sampling input port returns the most recent message received. If no message at all was received, it returns either the "default" (if provided) or raises a BufferError exception. If you call *readMsg()* and now new message has arrived since the last *readMsg()* call, you get the same message again. Example:

```
msg = self.inP1.readMsg(default=‘123‘)
```

The *nMsg()* method returns 0 if no message was received at 0, or 1 otherwise.

A program can call *wait()* so that is woken up if a message arrives on the port:

```
self.wait(20, [self.inP1])
```

A sampling input port wakes up a waiting program with *every* message that arrives.

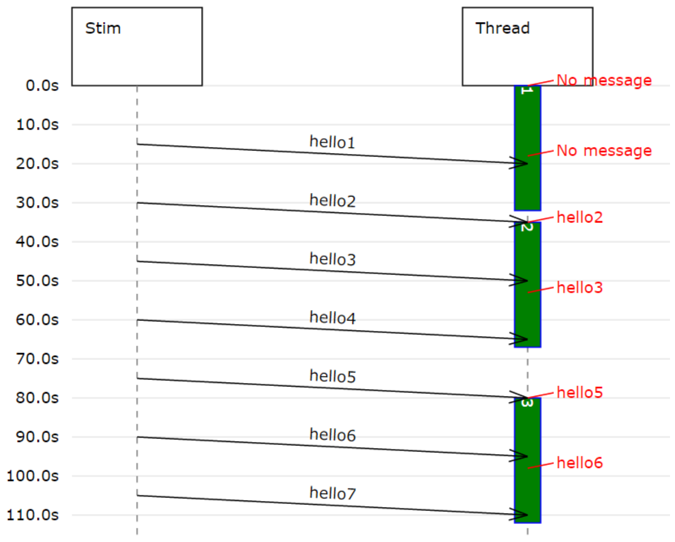
The following snippet demonstrates the use of a sampling port.

myThread1 is a program that has a sampling input port "inP1", while *stimThread* is firing messages to that port.

```
class myThread1(vSimpleProg):
    def __init__(self, sim ):
        super().__init__(sim=sim, objName='Thread', parentObj=None)
        self.createPorts('SamplingIn', ['inP1'])

    def showMsg(self):
        msg = self.inP1.readMsg(default='No message')
        self.addAnnotation(msg)

    def runVThread(self):
        cycle=0
        while True:
            cycle += 1
            self.showMsg()
            self.busy(18,cycle, busyAppearance)
            self.showMsg()
            self.busy(14,cycle, busyAppearance)
            self.wait(20,[self.inP1])


class stimThread(vSimpleProg):
    def __init__(self, sim ):
        super().__init__(sim=sim, objName='Stim', parentObj=None)
        self.createPorts('out', ['toT1Port'])

    def runVThread(self):
        count=0
        while True:
            count+=1
```

```
                    self.wait(15)
                    self.toT1Port.send('hello%d' % count,5)
```

This would result in the following sequence diagram:



2.3.3.2 Queing Input Ports

Recall from previous chapter:

- A queing port buffers all messages in a fifo queue. The queue depth is infinite.
- A read from the buffer consumes the first message

A queing input port is created with the *createPorts()* method, usually from the program's constructor:

```
    self.createPorts('QueingIn', ['inP1'])
```

The *readMsg()* method on a sampling input port returns the first message of the queue. If no message is in the queue, it raises a BufferError exception.

The *nMsg()* method returns the number of messages in the queue (0 if none).

A program can call *wait()* so that is woken up when a the *first message an empty queue* arrives on the port:

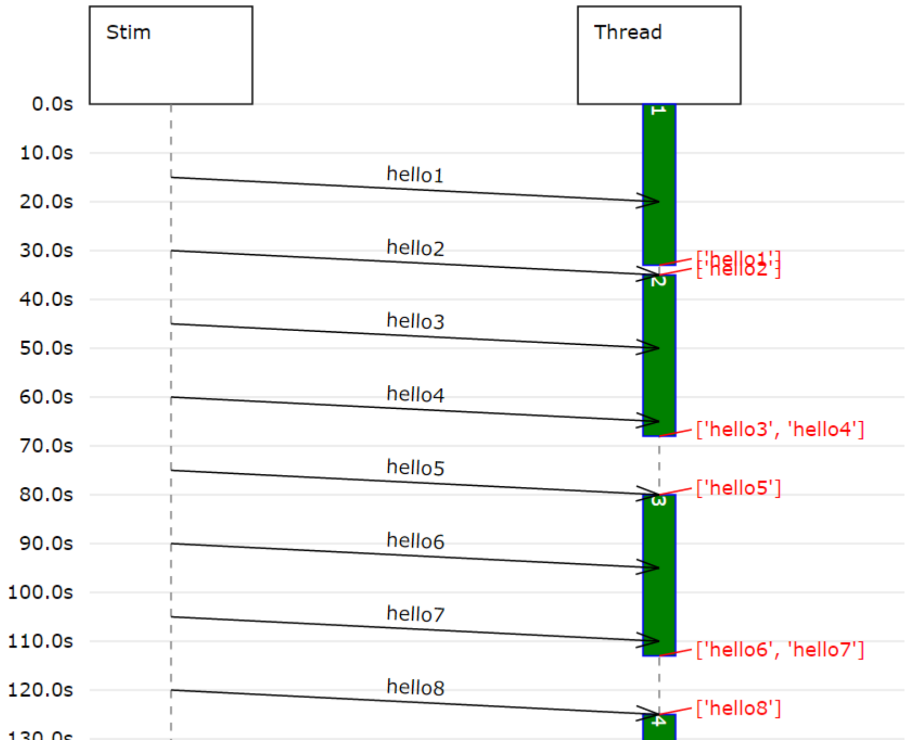
```
    self.wait(20, [self.inP1])
```

Note: Call *wait()* only on empty queing ports, otherwise the program will not be woken up (because the wakeup happens only at empty->non-empty transitions)!

The following snippet demonstrates the use of a queing port.

myThread1 is a program that has a queing input port "inP1", while *stimThread* is firing messages to that port.

```
class myThread1(vSimpleProg):
    def __init__(self, sim ):
        super().__init__(sim=sim, objName='Thread', parentObj=None)
        self.createPorts( 'QueingIn', ['inP1'])

    def getAllMsg(self):
        lstMsg = []
        while True:
            try:
                msg = self.inP1.readMsg()
                lstMsg.append(msg)
            except BufferError:
                break

        self.addAnnotation(lstMsg)


    def runVThread(self):
        cycle=0
        while True:
            cycle += 1
            self.busy(33, cycle, busyAppearance)
            self.getAllMsg()
            self.wait(20, [self.inP1])
            self.getAllMsg()


class stimThread(vSimpleProg):
    def __init__(self, sim ):
        super().__init__(sim=sim, objName='Stim', parentObj=None)
        self.createPorts( 'out', ['toT1Port'])

    def runVThread(self):
        count=0
        while True:
            count+=1
            self.wait(15,[])
            self.toT1Port.send('hello%d' % count,5)
```



2.3.4 System Calls for Sequential Programs

In the previous chapters, the system calls *wait()* and *busy()* were already briefly introduced. Now in more detail:

wait() delays the program execution for a specified time or until an event occurred.

The model is indicating with *wait()* that the program is idle. Therefore no status box is shown on the sequence diagram life line while a program is waiting.

```
wait(timeout, evList=[])
```

Where

- *timeout* is the timeout. If it is *None*, wait forever. A timeout value of 0 is invalid.
- *evList* is the list of events to wait for. Events must be instances from one of the classes *vtSamplingInPort, vtQueuingInPort,* or *vtTimer*. If evList is empty (or omitted), wait for timeout unconditionally.

wait() returns a string:

- 'ok' if one of the events has been triggered (it does not tell you which one)
- 'timeout' if timeout

busy() – delays the program execution for the specified time.

The model is indicating with *busy()* that the program is performing some operation. Therefore, a user defined status box appears on the life line while the program is busy.

```
busy(time, status, statusAppearance={})
```

Where:

- *time* is the busy time.
- *status* is the status text shown on the sequence diagram life line
- *statusAppearance* defines the decoration of the status box in the sequence diagram. See chapter 2.1.7

busy() returns always the string 'ok'.

2.3.5 Concurrent Program Execution/RTOS Simulation

Moddy comes with a simulation of a simple RTOS (real time operating system) scheduler. With this feature, you can model SW threads that run concurrently on a single CPU core.

The scheduler has the following features:

- 16 thread priorities - 0 is highest priority
- Priority based scheduling. Low priority threads run only if no higher thread ready.
- Threads with same priority will be scheduled round robin (when one of the same priority threads releases the processor, the next same priority thread which is ready is selected). Note that the round robin occurs not periodically. It happens only when one task executes *wait()*.

First, you create the scheduler object:

```
sched= vtSchedRtos(sim=simu, objName="sched", parentObj=None)
```

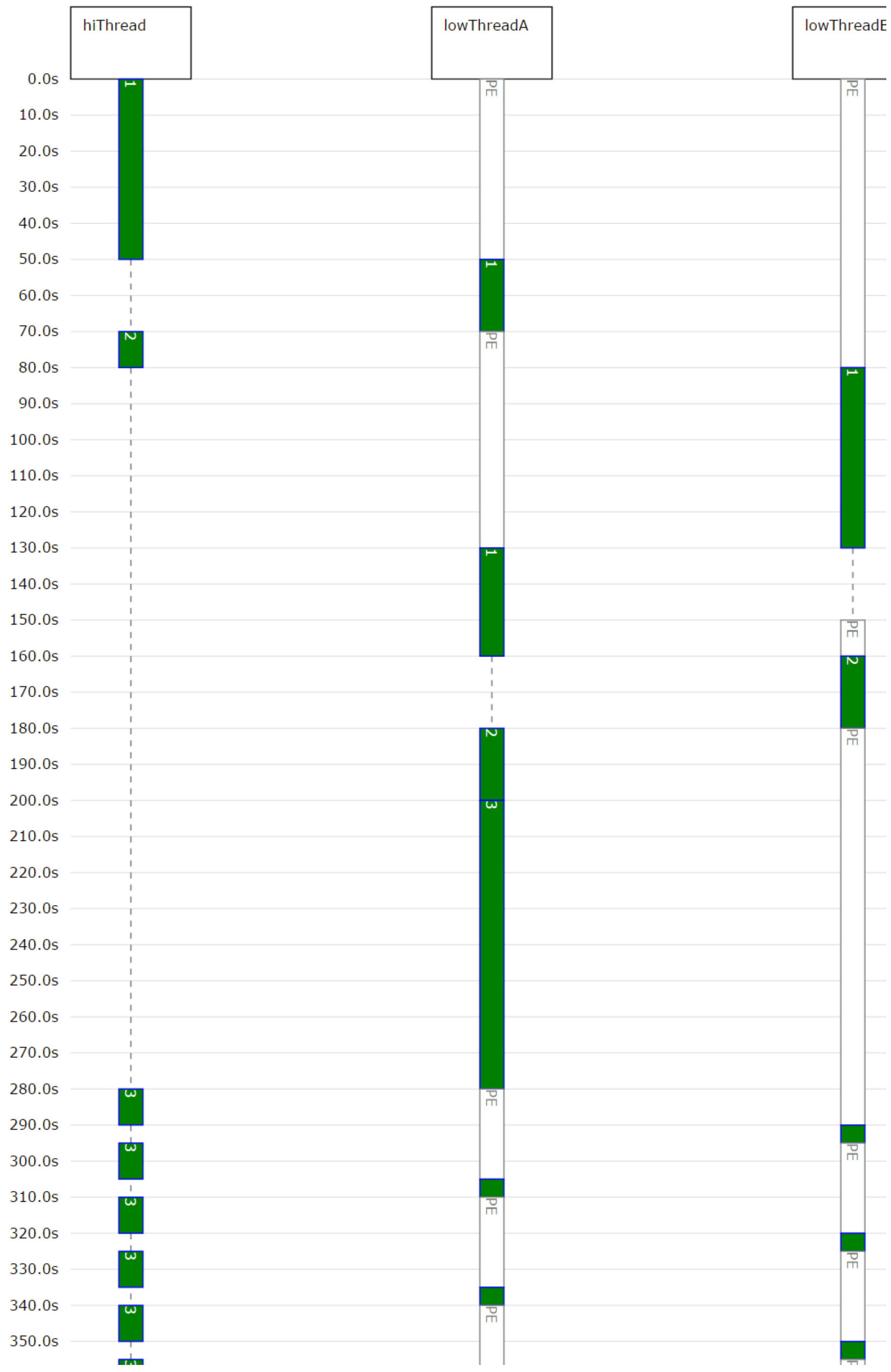
Then you add the threads (subclasses of vThread) that shall be scheduled by the scheduler:

```
t1 = myThread1(simu)
t2 = myThread2(simu)
t3 = myThread3(simu)
sched.addVThread(t1, prio=0)
sched.addVThread(t2, prio=1)
sched.addVThread(t3, prio=1)
```

Example snippet:

```
class myThread1(vThread):
    def __init__(self, sim ):
        super().__init__(sim=sim, objName='hiThread', parentObj=None)
    def runVThread(self):
        print("   VtHi1")
        self.busy(50,'1',busyAppearance)
        print("   VtHi2")
        self.wait(20,[])
        print("   VtHi3")
        self.busy(10,'2',busyAppearance)
        print("   VtHi4")
        self.wait(100,[])
        print("   VtHi5")
        self.wait(100,[])
        while True:
            print("   VtHi5")
            self.busy(10,'3',busyAppearance)
            self.wait(5,[])

class myThread2(vThread):
    def __init__(self, sim ):
        super().__init__(sim=sim, objName='LowThreadA', parentObj=None)
    def runVThread(self):
        print("   VtLoA1")
        self.busy(50,'1',busyAppearance)
        print("   VtLoA2")
        self.wait(20,[])
        print("   VtLoA3")
        self.busy(20,'2',busyAppearance)
        print("   VtLoA4")
        self.busy(250,'3',busyAppearance)

class myThread3(vThread):
    def __init__(self, sim ):
        super().__init__(sim=sim, objName='LowThreadB', parentObj=None)
    def runVThread(self):
        print("   VtLoB1")
        self.busy(50,'1',busyAppearance)
        print("   VtLoB2")
        self.wait(20,[])
        print("   VtLoB3")
        self.busy(100,'2',busyAppearance)
        print("   VtLoB4")
        self.busy(250,'3',busyAppearance)

simu = sim()
sched= vtSchedRtos(sim=simu, objName="sched", parentObj=None)

t1 = myThread1(simu)
t2 = myThread2(simu)
t3 = myThread3(simu)
sched.addVThread(t1, 0)
sched.addVThread(t2, 1)
sched.addVThread(t3, 1)
```

Resulting in the following sequence diagram:



Note: The "PE" status indicator tells you that the thread is "preempted", i.e. it would be ready to run, but it has to wait for the CPU resource, because a higher priority thread is busy.

2.3.6 vSimpleProg and vThread

vSimpleProg is a specialization of vThread. vSimpleProg uses an exclusive scheduler for the program, so there are no concurrent threads.

vThreads must be explicitly assigned to a scheduler.

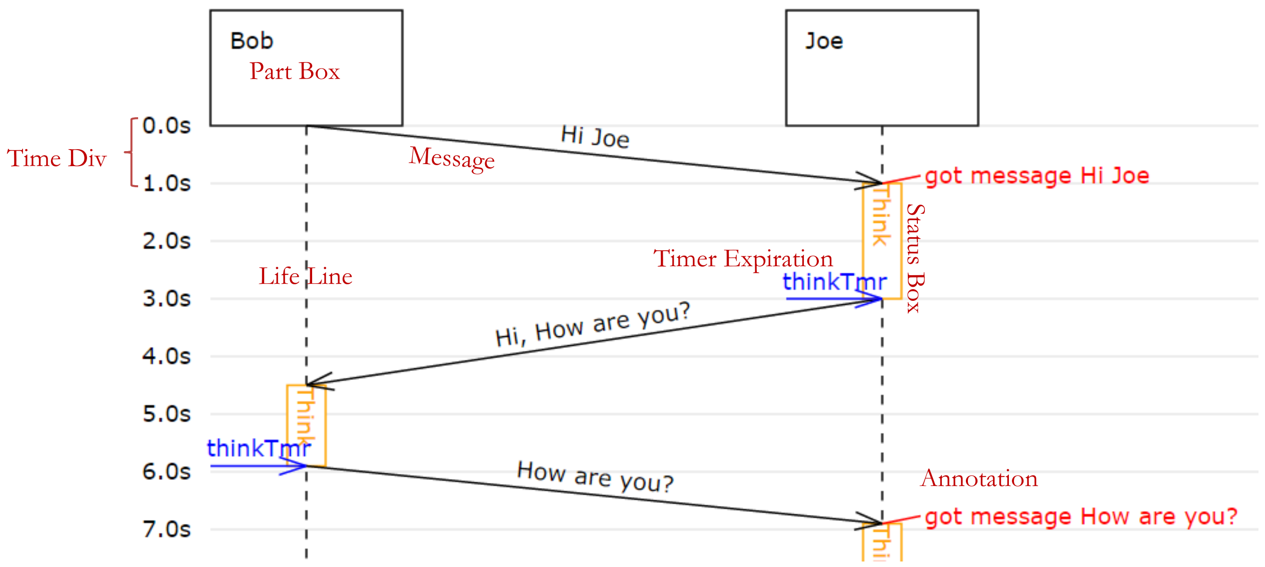
2.4 Visualisation

2.4.1 Sequence Diagram Generation

Moddy can generate sequence diagrams to show the communication between Moddy parts, including their timing information.

To generate a sequence diagrams, your code must call *moddyGenerateSequenceDiagram()* after the simulation has finished. The parameters provided to *moddyGenerateSequenceDiagram()* define the appearance of the sequence diagram.

The elements of a sequence diagram:



Parameters to *moddyGenerateSequenceDiagram()*:

- *sim:* the simulator object. Mandatory.
- *filename*: output filename (including .svg or .html). Mandatory.
- *fmt*: either 'svg' for pure SVG or 'svgInHtml' for SVG embedded in HTML. Optional, Default:'svg'
- *showPartsList*: Optional.
  If given, show only the listed parts in that order in sequence diagram. Each element can be either a reference to the part or a string with the hierarchy name of the part. If omitted, show all parts known by simulator, in the order of their creation.
- *showVarList*: Optional
  List of watched variables to include in the sequence diagram. Each element must be a string with the variable hierarchy name. e.g. "VC.var1". If omitted, no variables are included.
- *excludedElementList*: Optional.

    - parts or timers that should be excluded from drawing
    - Each list element can be the object to exclude or one of the following:
        - 'allTimers' - exclude all timers
- *timeRange*: tuple with start and end time. Everything before start and after end is not drawn. You can specify *None* for end time, in this case end time is the end of the simulation.
- *timePerDiv:* time per "Time Div". Mandatory.
- *title:* Title text to be displayed above the sequence diagram. Optional, default=None.
- *pixPerDiv*: pixels per time grid division. Optional, default:25
- *partSpacing:* horizontal spacing in pixels between parts. Optional, default:300
- *partBoxSize:* Tupel with x,y pixel size of part box. Optional, default:(100,60)
- *statusBoxWidth*: pixel width of status box on life line. Optional, default:20

The sequence diagrams are generated in SVG (scalable vector graphics) format. You can choose between two different flavors:

- **Pure SVG** (fmt='svg', filename ends with .svg): Use this if you want to edit the sequence diagram. SVG files can be edited with SVG capable editors such as MS Visio or Inkskape.
- **SVG embedded in HTML** (fmt='svgInHtml', filename ends with .html): Use this if you want to show the sequence diagram in a web browser. In this case, the SVG is embedded into a HTML page and a <div> element so that the browser can scroll through the diagram.

In simple cases, you call:

```
moddyGenerateSequenceDiagram( sim=simu,
                  fileName="myExample.html",
                  fmt="svgInHtml",
                  showPartsList=["Bob", "Joe"],
                  timePerDiv = 1.0)
```

A more complex case:

```
        moddyGenerateSequenceDiagram( sim=simu,
                                      fileName="2_sergw.html",
                                      fmt="svgInHtml",
                                      timeRange=(1.0*ms,None),
                                      title="Serial Gateway Demo",
                                      showPartsList=[client, gateway.rxThread,
                                       gateway.txThread, serDev],
                                      excludedElementList=['allTimers'],
                                      timePerDiv = 50*us,
                                      pixPerDiv = 30)
```

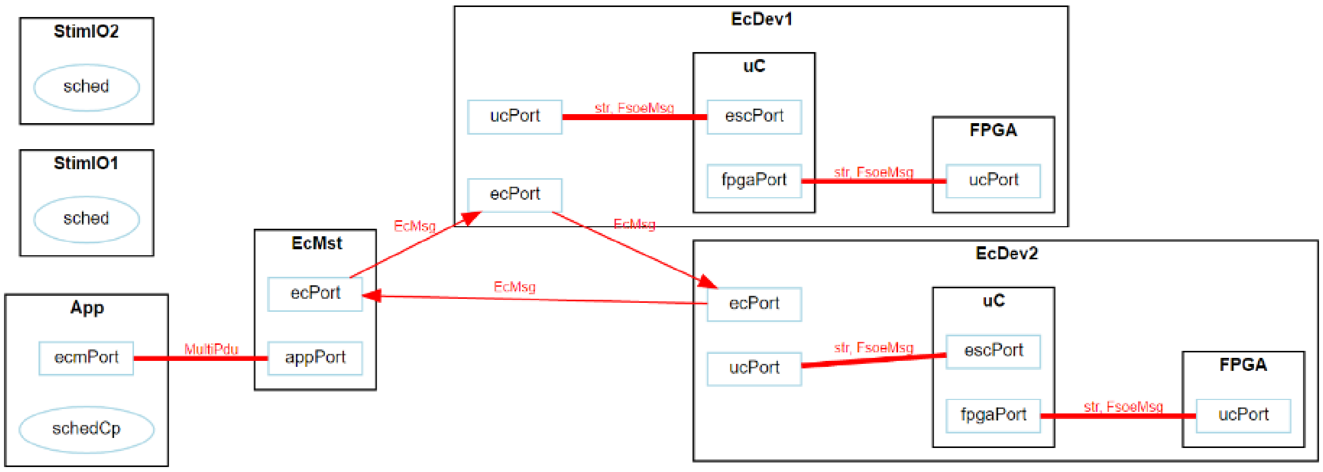
Here, the sequence diagram begins at time 1.0ms, no timer expiration events are shown, the time division is set to 50µs and each time division occupies 30 pixels.

You can call *moddyGenerateSequenceDiagram()* multiple times to show different views of the simulation.

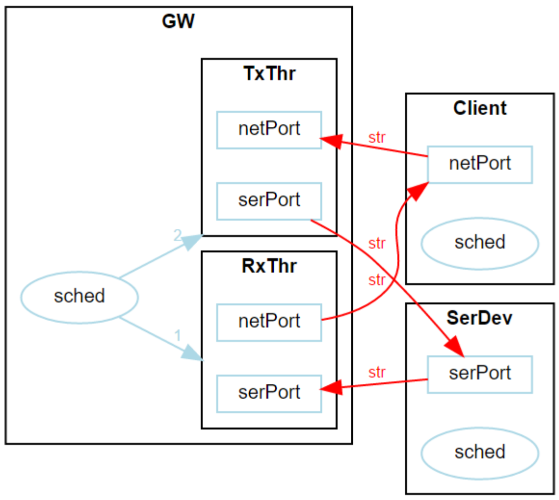
2.4.2 Structure Graph Generation

Moddy can also generate files to show the static structure of the model. Very useful to document the model and to explain the model to others.

An example structure graph:



- The parts hierarchy is shown: For example, EcDev1 contains two sub-parts: “uC” and “FPGA”.
- The ports of a part are shown: For example, EcDev1 has a “ucPort” and an “ecPort”.
- The port bindings are shown:
  - The thin, directed arrows show unidirectional bindings, from an output port to an input port.
  - The thick, undirected arrows show bidirectional bindings between IO-Ports.
- The message type transferred between ports are shown (e.g. EcMsg): Moddy learns during the simulation which message types (python classes) are transferred.
- The relation between schedulers and parts are shown. In the example above, “schedCp” controls “App”. In the example below, if the scheduler “sched” controls multiple threads, the relation is shown, including the thread priorities. (WARNING: This feature does not work under all circumstances)



Moddy uses the “GraphViz” command line tool “dot” to generate the structure graphs, so “GraphViz” has to be installed. The output format is fixed to SVG.

To generate a structure, just call *moddyGenerateStructureGraph()* after running the simulation. (You can also run it before the simulation, but then no message types will be shown)

```
moddyGenerateStructureGraph(simu, '2_sergw_structure.svg')
```

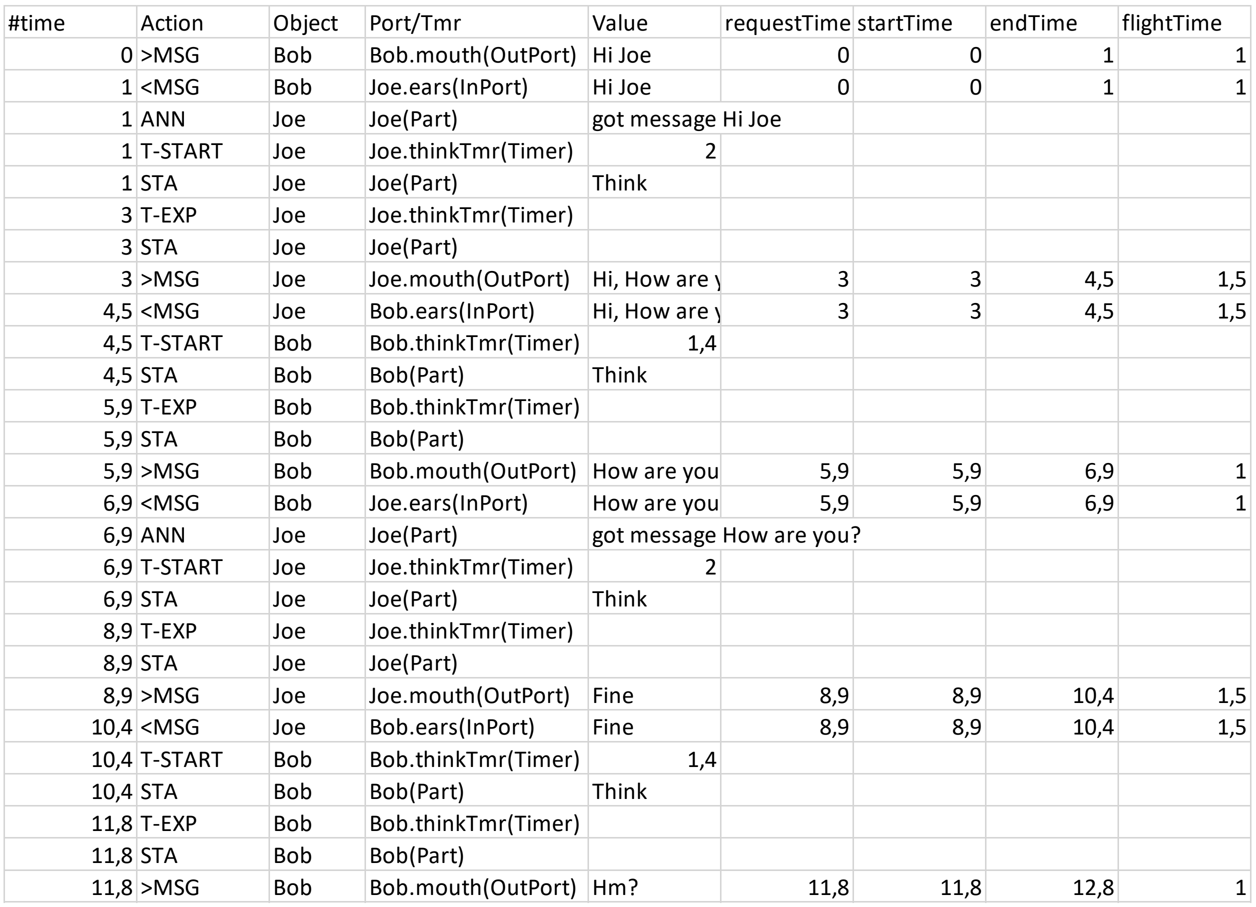
There is currently no way to influence the appearance of the structure graph.

The only optional parameter to *moddyGenerateStructureGraph()* is *keepGvFile*. If True, don't delete GraphViz input file.

2.4.3 Trace Table Generation

Moddy can generate a CSV (comma separated value) file containing all simulator events. This file can be viewed with a table calculation program such as MS Excel.

Example:



#time	Action	Object	Port/Tmr	Value	requestTime	startTime	endTime	flightTime
0	>MSG	Bob	Bob.mouth(OutPort)	Hi Joe	0	0	1	1
1	<MSG	Bob	Joe.ears(InPort)	Hi Joe	0	0	1	1
1	ANN	Joe	Joe(Part)	got message Hi Joe				
1	T-START	Joe	Joe.thinkTmr(Timer)	2				
1	STA	Joe	Joe(Part)	Think				
3	T-EXP	Joe	Joe.thinkTmr(Timer)					
3	STA	Joe	Joe(Part)					
3	>MSG	Joe	Joe.mouth(OutPort)	Hi, How are y	3	3	4,5	1,5
4,5	<MSG	Joe	Bob.ears(InPort)	Hi, How are y	3	3	4,5	1,5
4,5	T-START	Bob	Bob.thinkTmr(Timer)	1,4				
4,5	STA	Bob	Bob(Part)	Think				
5,9	T-EXP	Bob	Bob.thinkTmr(Timer)					
5,9	STA	Bob	Bob(Part)					
5,9	>MSG	Bob	Bob.mouth(OutPort)	How are you	5,9	5,9	6,9	1
6,9	<MSG	Bob	Joe.ears(InPort)	How are you	5,9	5,9	6,9	1
6,9	ANN	Joe	Joe(Part)	got message How are you?				
6,9	T-START	Joe	Joe.thinkTmr(Timer)	2				
6,9	STA	Joe	Joe(Part)	Think				
8,9	T-EXP	Joe	Joe.thinkTmr(Timer)					
8,9	STA	Joe	Joe(Part)					
8,9	>MSG	Joe	Joe.mouth(OutPort)	Fine	8,9	8,9	10,4	1,5
10,4	<MSG	Joe	Bob.ears(InPort)	Fine	8,9	8,9	10,4	1,5
10,4	T-START	Bob	Bob.thinkTmr(Timer)	1,4				
10,4	STA	Bob	Bob(Part)	Think				
11,8	T-EXP	Bob	Bob.thinkTmr(Timer)					
11,8	STA	Bob	Bob(Part)					
11,8	>MSG	Bob	Bob.mouth(OutPort)	Hm?	11,8	11,8	12,8	1

To generate a trace table, call *moddyGenerateTraceTable()* after running the simulation:

```
moddyGenerateTraceTable(simu, fileName='1_hello.csv', timeUnit=1.0)
```

Where *timeUnit* specifies the time unit for all time stamps in table ('s', 'ms', 'us', 'ns').

The optional parameter *floatComma* specified the character used in floating point numbers (default ',', you can set it to '.')

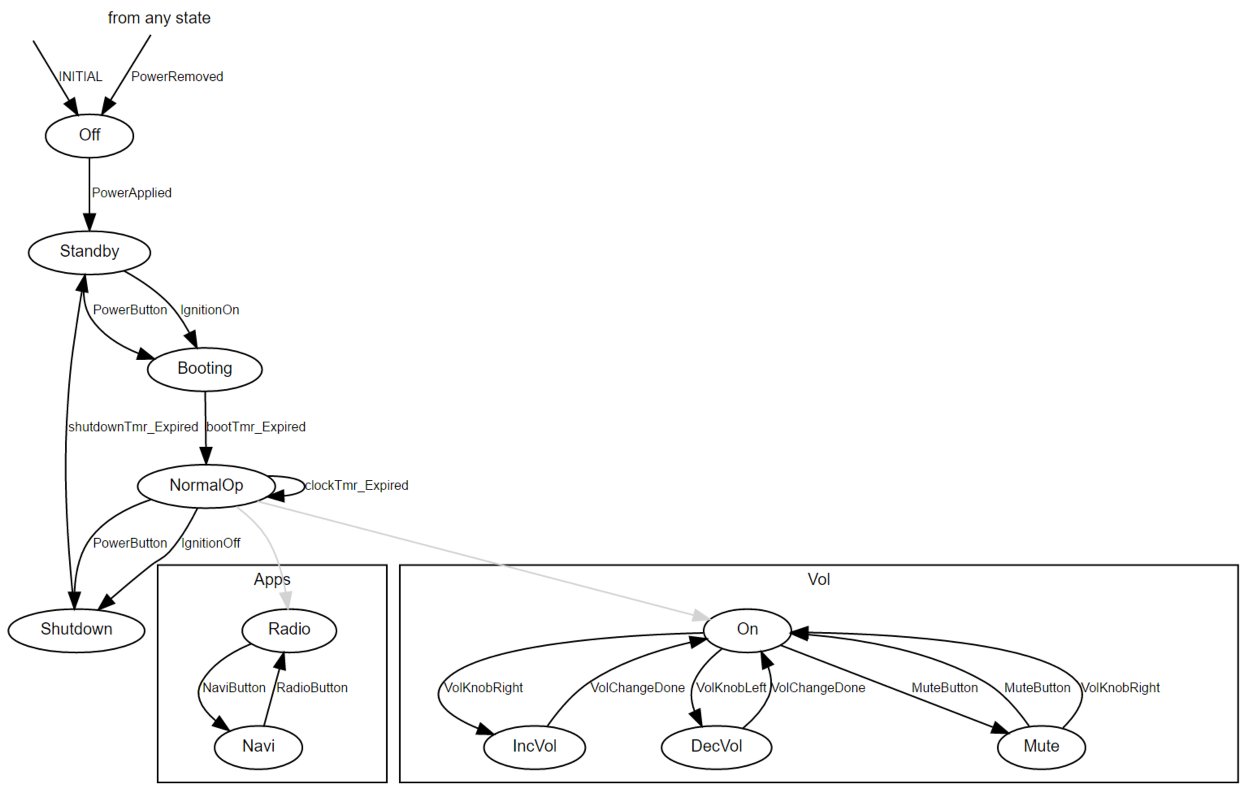
Explanation to the events:

- >MSG: A message transfer starts.
- >MSG(Q): A message transfer starts. The message was delayed because another message was blocking the output port. The requestTime tells you when the model requested the message send
- <MSG: Message arrived at input port: Note that "Object" specifies the part that contains the output port
- T-START: A timer has been started
- T-STOP: A timer has been stopped
- T-RESTA: A timer was restarted
- T-EXP: A timer was expired
- ANN: Model added an annotation
- STA: Part changed its status (via setStatusIndicator)

2.4.4 State Machine Graph Generation

Moddy can also generate graphs to show the finite state machines of the model.

An example state machine graph (from tutorial "3_carinfo.py"):



The "Apps" and "Vol" boxes are sub-states of the "NormalOp" state.

To generate a state machine graph, just call *moddyGenerateFsmGraph()* after state machine instantiation.

```
    moddyGenerateFsmGraph( fsm=cis.fsm, fileName='3_carinfo_fsm.svg')
```

There is currently no way to influence the appearance of the graph.

The only optional parameter to *moddyGenerateFs,Graph()* is *keepGvFile*. If True, don't delete GraphViz input file.

3 Installation

Tested with python 3.5.2 and 3.6 under Windows 10, Eclipse Neon/pyDev

Required libraries: svgwrite https://pypi.python.org/pypi/svgwrite , tested with svgwrite 1.1.9

Required tools: GraphViz, tested with Version 2.38

3.1 Installing Eclipse

Download & Install Eclipse from https://eclipse.org/downloads/

3.2 Install Python

Download & Install Python from https://www.python.org/downloads/

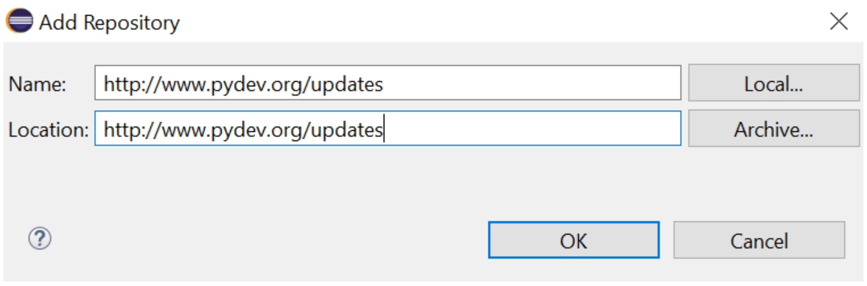
Please install python Version >=3.5.

Moddy does not support Python 2.x.

During install, click checkbox “add python to PATH”

3.3 Installing PyDev

In Eclipse, select Help->Install new Software and enter http://www.pydev.org/updates for both Name and location


Add Repository

Name: http://www.pydev.org/updates

Location: http://www.pydev.org/updates

Select “pyDev” and follow the installation instructions

3.4 Installing svgwrite

Open Windows command line.

From the command line, call

```
pip install svgwrite
```

3.5 Installing GraphViz

Install GraphViz from

http://www.graphviz.org/Download..php

Then ensure that the bin/ subdirectory of the GraphViz installation is in your PATH.

3.6 Installing moddy

Clone moddy from GitHub.

Let's assume you want to clone into an eclipse project called "mymoddy".

Change into the eclipse workspace path:

```
# D:
# cd D:\owncloud\documents\Klaus\eclipse
```

Clone moddy into "mymoddy" subdirectory

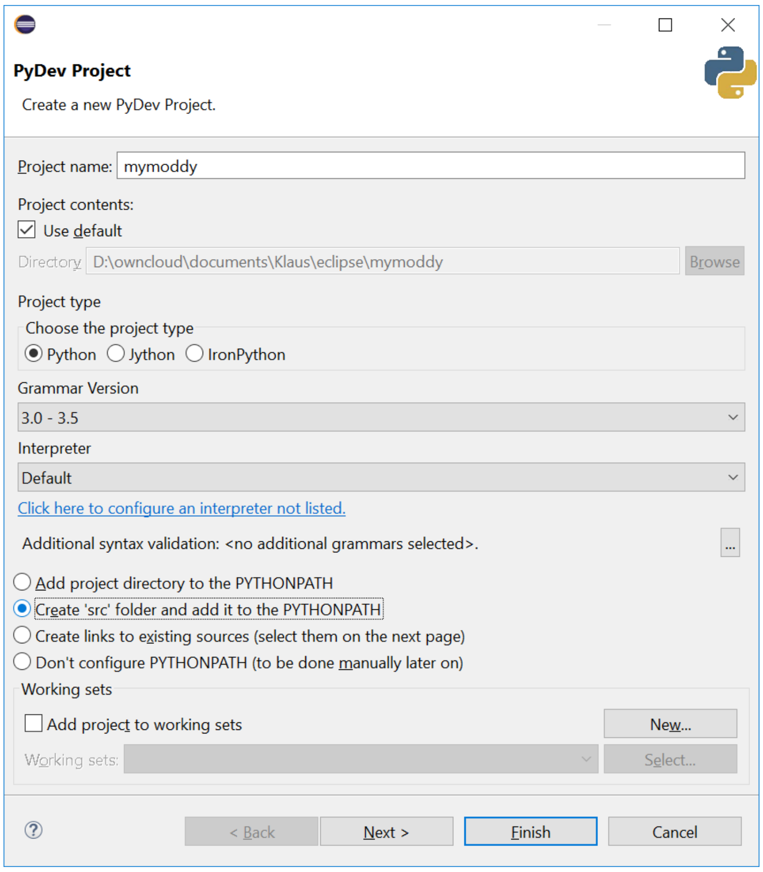
```
# git clone https://github.com/KlausPopp/Moddy.git mymoddy
```

From eclipse, select File->New->PyDev Project

Choose a the project name according to the directory where you cloned moddy to (here:mymoddy).

Select PyDev – Interpreter = 3.x (according to your phyton version)

Select "Create 'src' folder and add it to the PYTHONPATH



PyDev Project

Create a new PyDev Project.

Project name: mymoddy

Project contents:

☑ Use default

Directory D:\owncloud\documents\Klaus\eclipse\mymoddy

Project type

Choose the project type

● Python ○ Jython ○ IronPython

Grammar Version: 3.0 - 3.5

Interpreter: Default

Click here to configure an interpreter not listed.

Additional syntax validation: <no additional grammars selected>.

○ Add project directory to the PYTHONPATH

● Create 'src' folder and add it to the PYTHONPATH

○ Create links to existing sources (select them on the next page)

○ Don't configure PYTHONPATH (to be done manually later on)

Working sets

☐ Add project to working sets

Working sets:

In eclipse, try to run the tutorial:

Right click in Package Explorer to "moddy/src/tutorial/1_hello.py".

Select "Run As"->"Python Run"

The console log should end with

```
…
Drawing ANN events
saved 1_hello.html as svgInHtml
Saved structure graph to 1_hello_structure.svg
saved 1_hello.csv as CSV
```

If you want to update Moddy, go into the directory where you have Moddy cloned to and execute:

```
# git pull
```