

# A Python Finite State Machine Library for EPICS

Dr. Marcato Davide

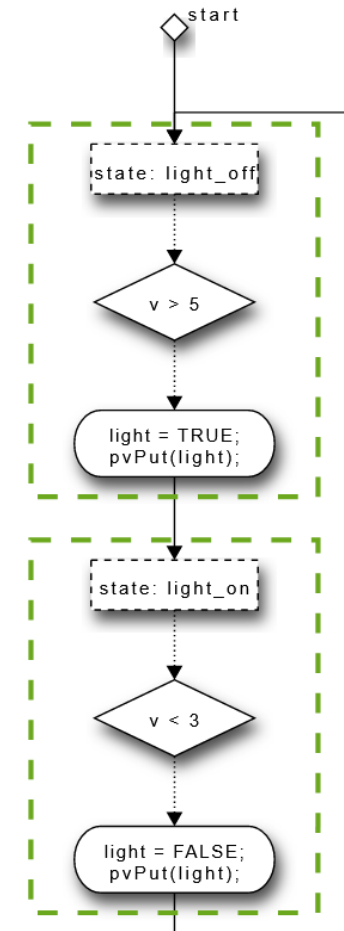
INFN – Legnaro National Laboratories

[davide.marcato@lnl.infn.it](mailto:davide.marcato@lnl.infn.it)

18th International Conference on Accelerator and Large Experimental Physics Control Systems

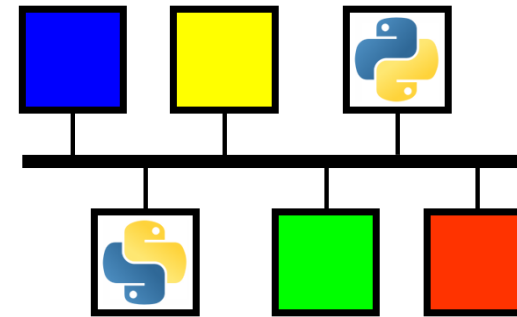
# The EPICS Sequencer

- A tool to define procedures and sequences of operations in EPICS
- State Notation Language
  - To describe Finite State Machines (FSM) states and transitions
  - C-like language, transcompiled to C
- Standard tool in the EPICS community
  - First proposed on the original EPICS paper
  - Good performance and reliability
  - Flexible programming model
- Low level language
  - Unfamiliar to new users
  - Limited expandability



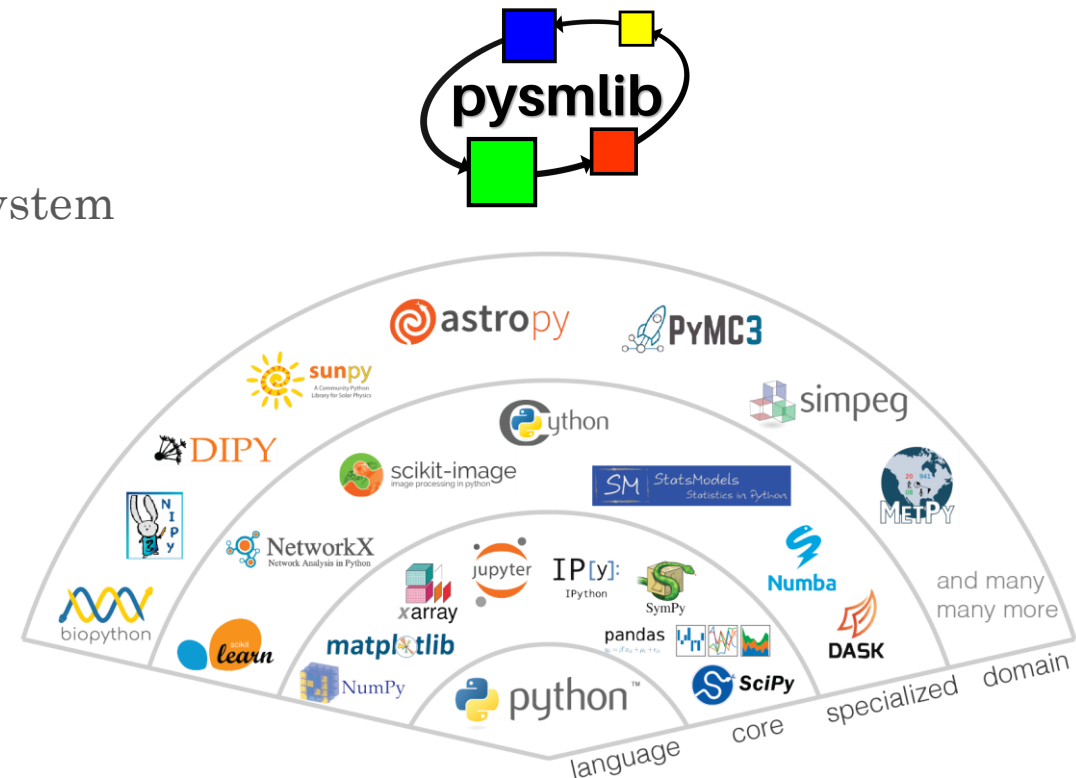
# Alternatives

- Vanilla PyEpics scripts
  - Easy, fast to prototype
  - Basic functionality
- Bluesky project
  - Complete suite of tools for data acquisition, experiment specification and orchestration.
  - Advanced functionalities
  - Requires a big investment into their design model
- Facility or experiment-specific tools
  - Not available to smaller labs/experiments



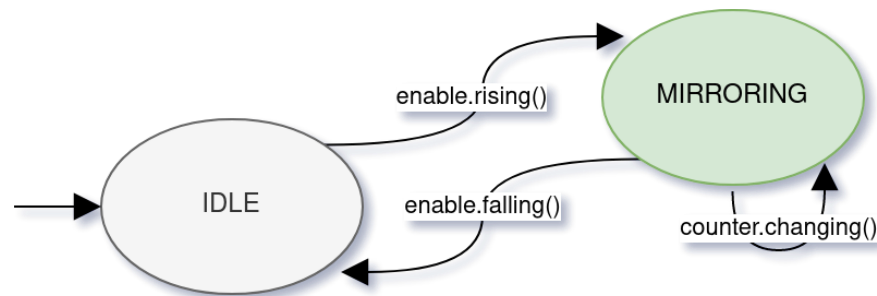
# Pysmlib

- A simpler alternative to the EPICS sequencer
  - High level description of FSMs
  - Leave implementation details to the library
- Python language
  - High level language
  - Rich scientific and engineering ecosystem
  - Familiar to many new users



# Example FSM

- Subclass fsmBase
  - Connect to the PVs on the constructor
- Idle state
  - Wait for enable
- Mirroring state
  - Copy the value of the *counter* PV to the *mirror* PV

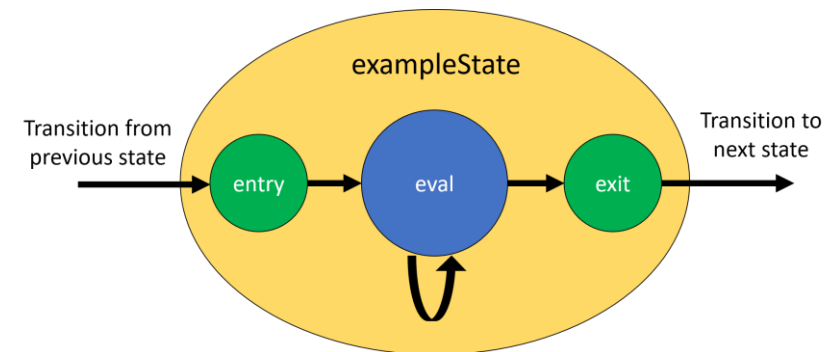


```
1  #!/usr/bin/python
2  from smlib import fsmBase, loader
3
4  # FSM definition
5  class exampleFsm(fsmBase):
6      def __init__(self, name, *args, **kwargs):
7          super(exampleFsm, self).__init__(name, **kwargs)
8
9          self.counter = self.connect("testcounter")
10         self.mirror = self.connect("testmirror")
11         self.enable = self.connect("testenable")
12
13         self.gotoState('idle')
14
15     # idle state
16     def idle_eval(self):
17         if self.enable.rising():
18             self.gotoState("mirroring")
19
20     # mirroring state
21     def mirroring_eval(self):
22         if self.enable.falling():
23             self.gotoState("idle")
24         elif self.counter.changing():
25             readValue = self.counter.val()
26             self.mirror.put(readValue)
27
28 # Main
29 if __name__ == '__main__':
30     # load the fsm
31     l = loader()
32     l.load(exampleFsm, "myFirstFsm")
33
34     # start execution
35     l.start()
```



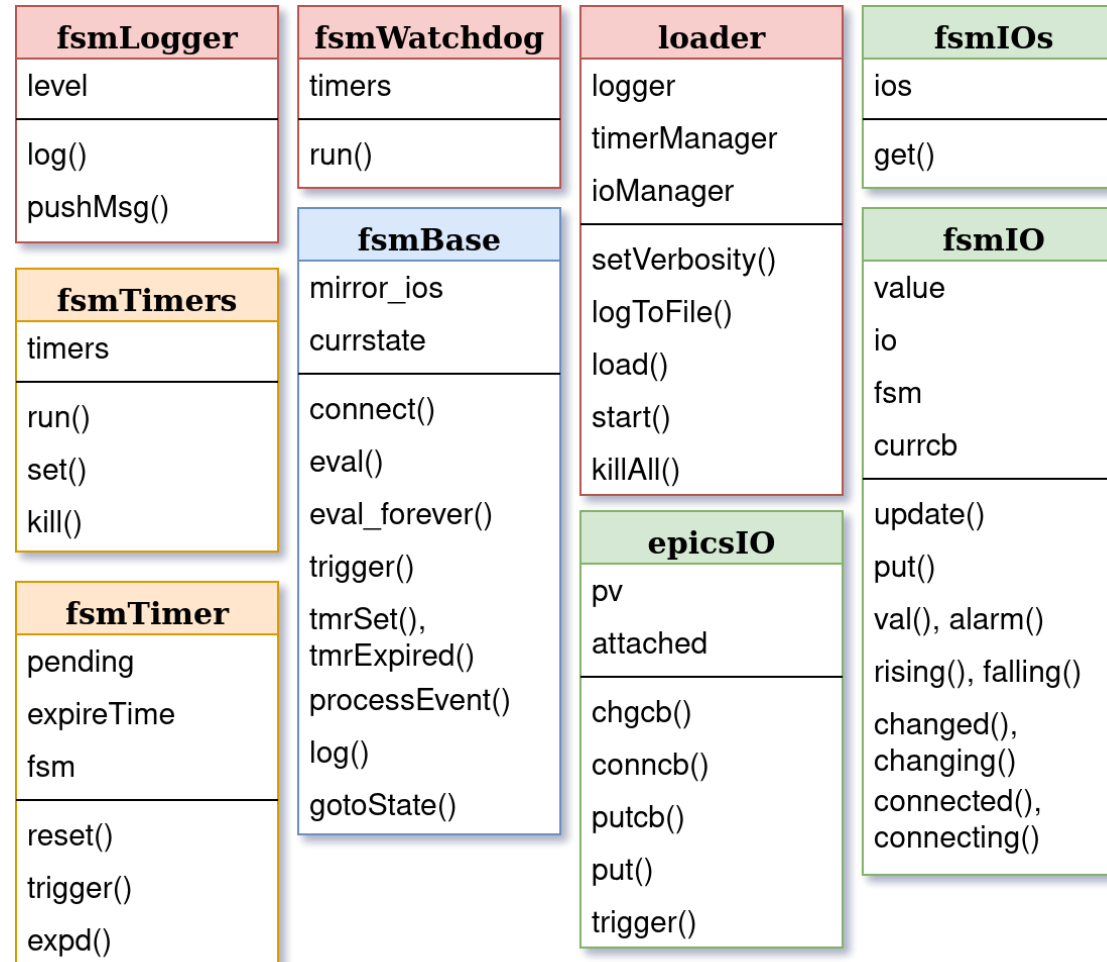
# Design

- Event driven FSM
- Daemon-like execution flow
  - Concurrent execution of multiple FSMs
- Network efficiency
  - Share the Channel Access PV connections across FSMs
- Inputs should not change during the state execution
  - Each input event triggers one state execution
- Execute actions on state transitions
  - *entry*, *eval*, *exit* methods



# Architecture

- 4 main subsystems
  - Input management
  - FSM execution
  - Timers
  - Utilities



# Input Management

- 3 event types from Channel Access
  - *change, connection, put\_complete*
- One PV emits an event
  - The event data is placed on thread-safe queues
- All the FSM connected to the corresponding input are executed
  - Each one is a different thread
- Each FSM keeps a local proxy of all its inputs
  - fsmIO class
  - Updated with the data retrieved from the queue
- The current state is executed
  - The triggering event type is used to check edge conditions

| epicsIO   |
|-----------|
| pv        |
| attached  |
| chgcb()   |
| conncb()  |
| putcb()   |
| put()     |
| trigger() |

| fsmIOs |
|--------|
| ios    |
| get()  |

| fsmIO                        |
|------------------------------|
| value                        |
| io                           |
| fsm                          |
| currCb                       |
| update()                     |
| put()                        |
| val(), alarm()               |
| rising(), falling()          |
| changed(),<br>changing()     |
| connected(),<br>connecting() |

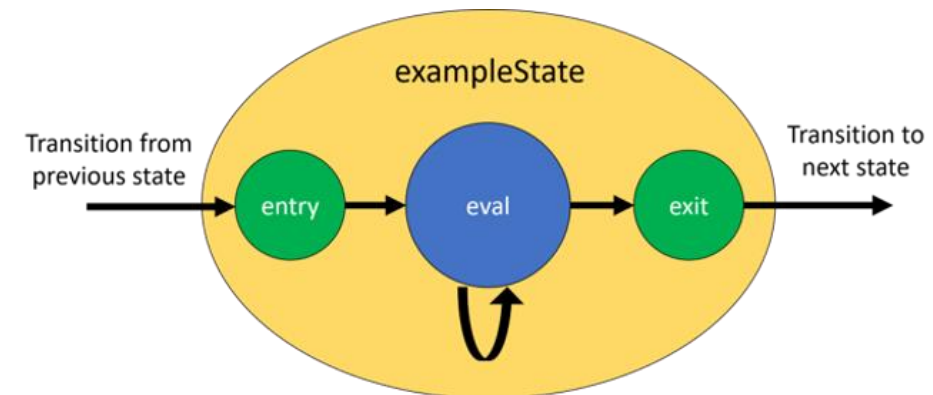


# Execution Flow

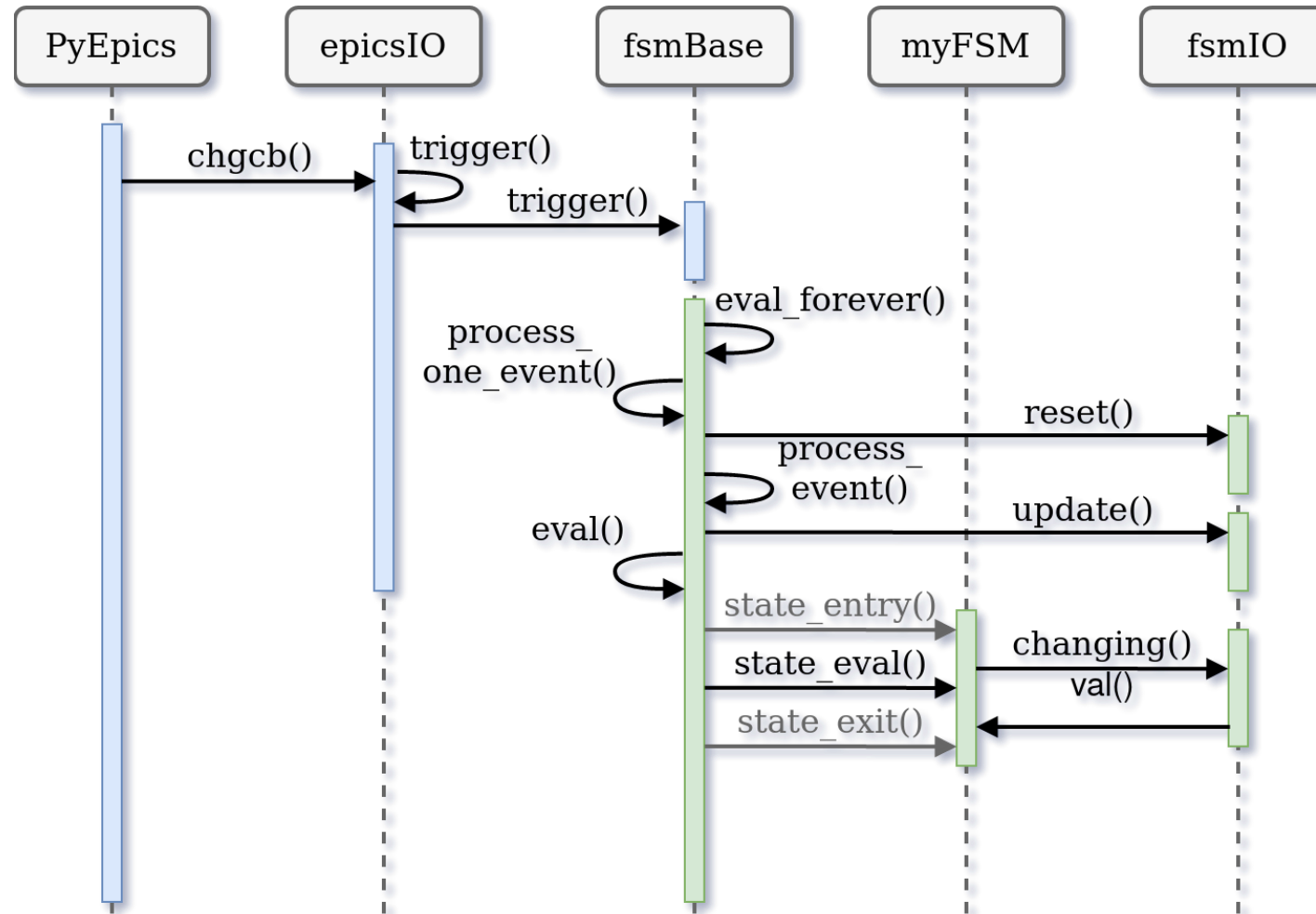
1. Perform a state transition if required. In this case it also executes the **\_entry()** method of the new state, if it's defined.
2. Execute the **\_eval()** method of the current state.
3. If the user requested a state transition, the **\_exit()** method of the current state is executed. In this case go back to step 1 without processing a new event.

**gotoState()** automatically finds the right methods based on the state name

| fsmBase                   |
|---------------------------|
| mirror_ios                |
| currstate                 |
| connect()                 |
| eval()                    |
| eval_forever()            |
| trigger()                 |
| tmrSet(),<br>tmrExpired() |
| processEvent()            |
| log()                     |
| gotoState()               |



# Change event example



# Timers

- Trigger FSM execution after a fixed time delay
  - To check timeouts, perform periodic actions, wait before an action...
- Internal event of type *timer\_expired*
  - A thread manages all the timers and queues events

```
def move_entry(self):  
    self.motor.put(100)                # move the motor  
    self.tmrSet('moveTimeout', 10)     # Set a timer of 10s  
  
def move_eval(self):  
    if self.doneMoving.rising():        # If the motor movement completed  
        self.gotoState("nextState")    # continue to next state  
    elif self.tmrExpiring("moveTimeout"): # Timer expired event  
        self.gotoState("error")        # go to an error state
```

# Utilities

## Logger

- Unified interface to log to different backends

## Loader

- Load multiple FSM on a single executable
- Share resources

## Watchdog

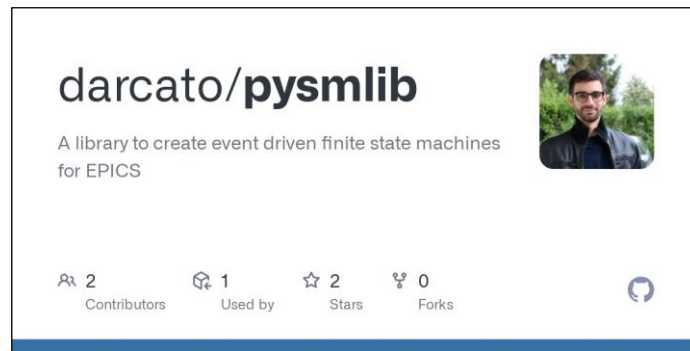
- Specify PV as watchdog
- A thread periodically writes a value
- The PV goes into alarm if no writes occur after a delay

# User Experience

- First concept in 2016 for RF control system @ LNL
- Used for many other subsystems
  - Diagnostic, ion beam sources, vacuum
- Simulators
  - Replace real devices by simulating their actions on PVs
- Alarm handling
  - Example: send notification via Telegram
- Beam Optimization Procedures
  - BOLINA
- Useful when asynchronous interaction is expected
  - Eg: user input, non-constant delays
  - Trigger on the rising or falling edges of conditions

# Publishing

<https://github.com/darcato/pysmilib>



pipeline **passed** coverage **63.00%**

GPL-3.0 License

<https://pypi.org/project/pysmilib/>



<https://darcato.github.io/pysmilib>

Docs » Pysmilib overview [View page source](#)

## Pysmilib overview

This section will describe the standard workflow to go from an empty file editor to a running finite state machine with pysmilib. Each step will be then explained in detail in the following sections of this documentation.

### Define your FSM

Pysmilib lets you create finite state machines, so the first step is to adapt your algorithm to a fsm design. This means identifying all the states required and the conditions that trigger a transition from one state to another. Furthermore, all the required input and outputs must be identified: the input are usually needed to determine the current state and receive events, while the outputs are used to perform actions on the external world.

The library is designed to be connected to EPICS PVs, so EPICS IOCs must be running with the required PVs, otherwise the FSM will sleep waiting for the PVs to connect.

### General structure

Each finite state machine is created as a derived class from `fsmBase`, which is part of pysmilib.

```
from smlib import fsmBase

class exampleFsm(fsmBase):
    def __init__(self, name, *args, **kwargs):
        super(exampleFsm, self).__init__(name, **kwargs)
```

In this snippet of code the class is declared and the parent class is initialized, passing a `name` as argument which identifies the class instance. In fact, when this code will be executed a new thread will be created for each instance of the class.

**Note**

Never forget to include `**kwargs` in the arguments of the super class as they are used by the `loader`.

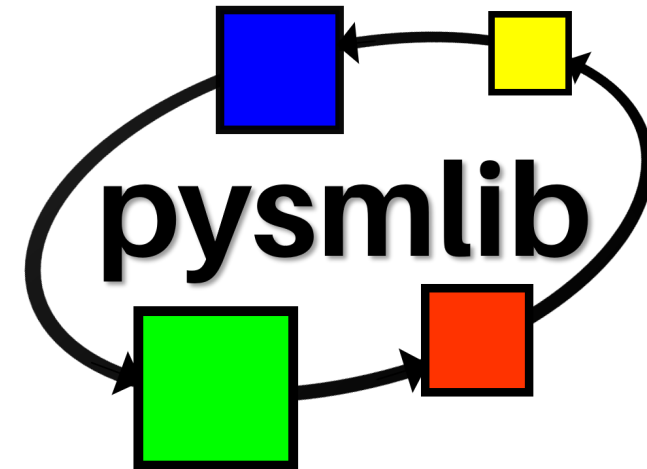
### Define inputs / outputs

In the class constructor the I/O must be defined. Note that there is no actual distinction between a input and a output,



# Conclusion

- Pysmlib: A library to develop EPICS Finite State Machines
  - Focus on simplicity
  - Great expandability with Python libraries
  - Useful features for common use-cases
- Available to the whole EPICS community
  - Makes no assumption
  - Tested and running in production
- Future improvements
  - Add support for different input types (pvAccess?)
  - Contributions are welcome



# Thank you

Davide Marcato

