

A Python Finite State Machine Library for EPICS

Dr. Marcato Davide

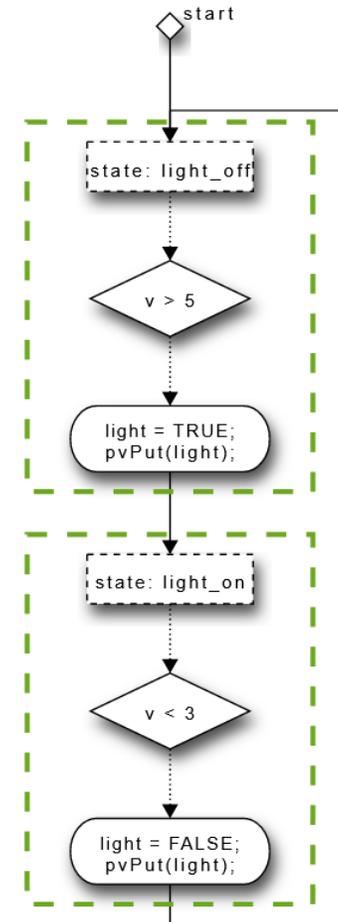
INFN – Legnaro National Laboratories

davide.marcato@lnl.infn.it

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

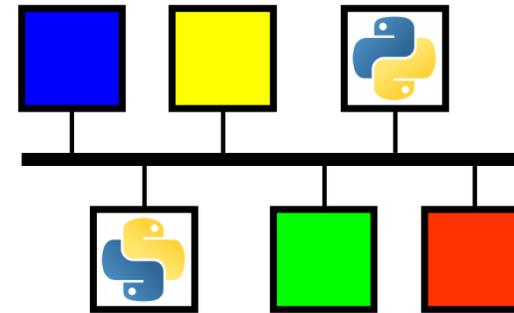
The EPICS Sequencer

- 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



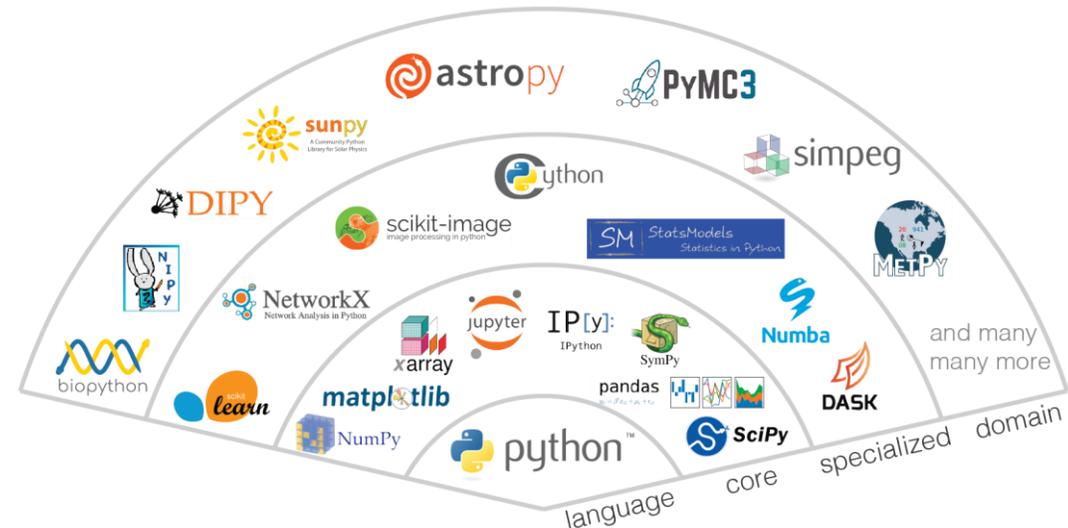
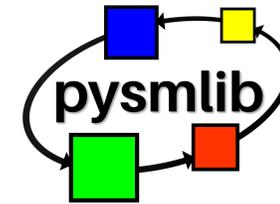
Alternatives

- 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



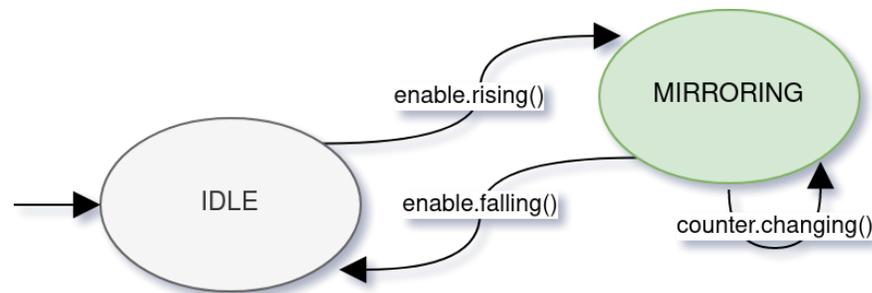
Pysmlib

- 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

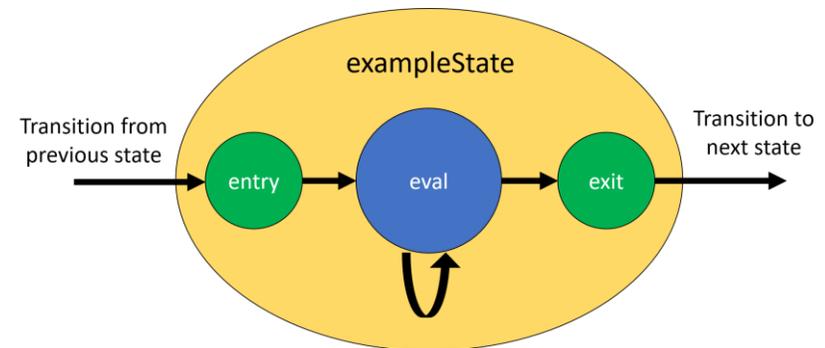
- Subclass fsmBase
 - Connect to the PVs on the constructor
- Idle state
 - Wait for enable
- Mirroring state
 - Copy the value of the *counter* PV to the *mirror* PV



```
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()
```

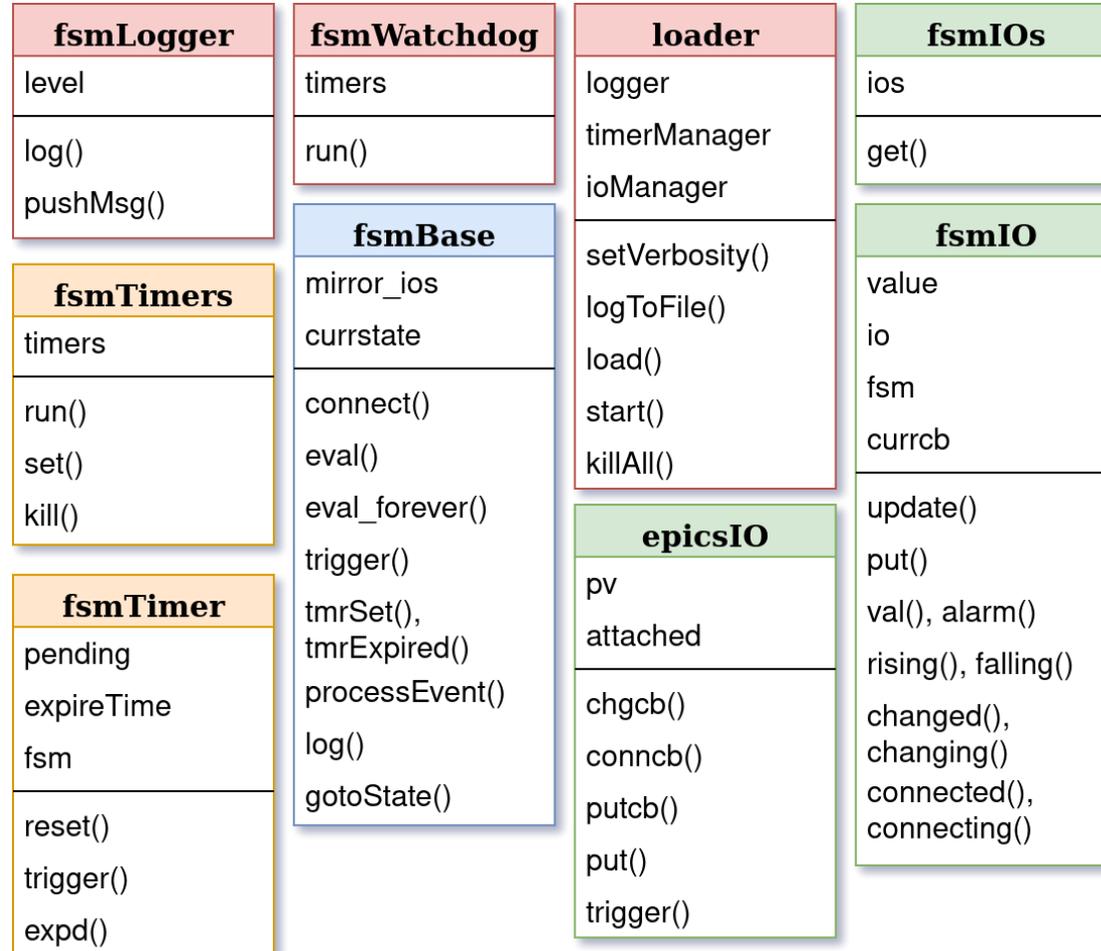
Design

- 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()

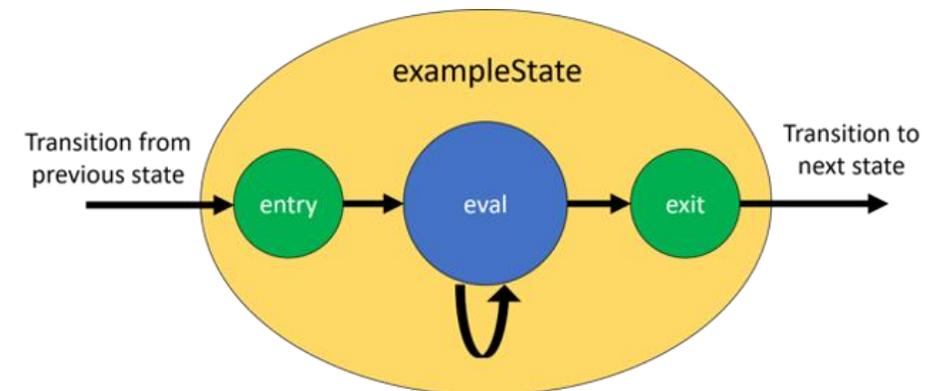
fsmIO
value io fsm currcb
update() put() val(), alarm() rising(), falling() changed(), changing() connected(), connecting()

Execution Flow

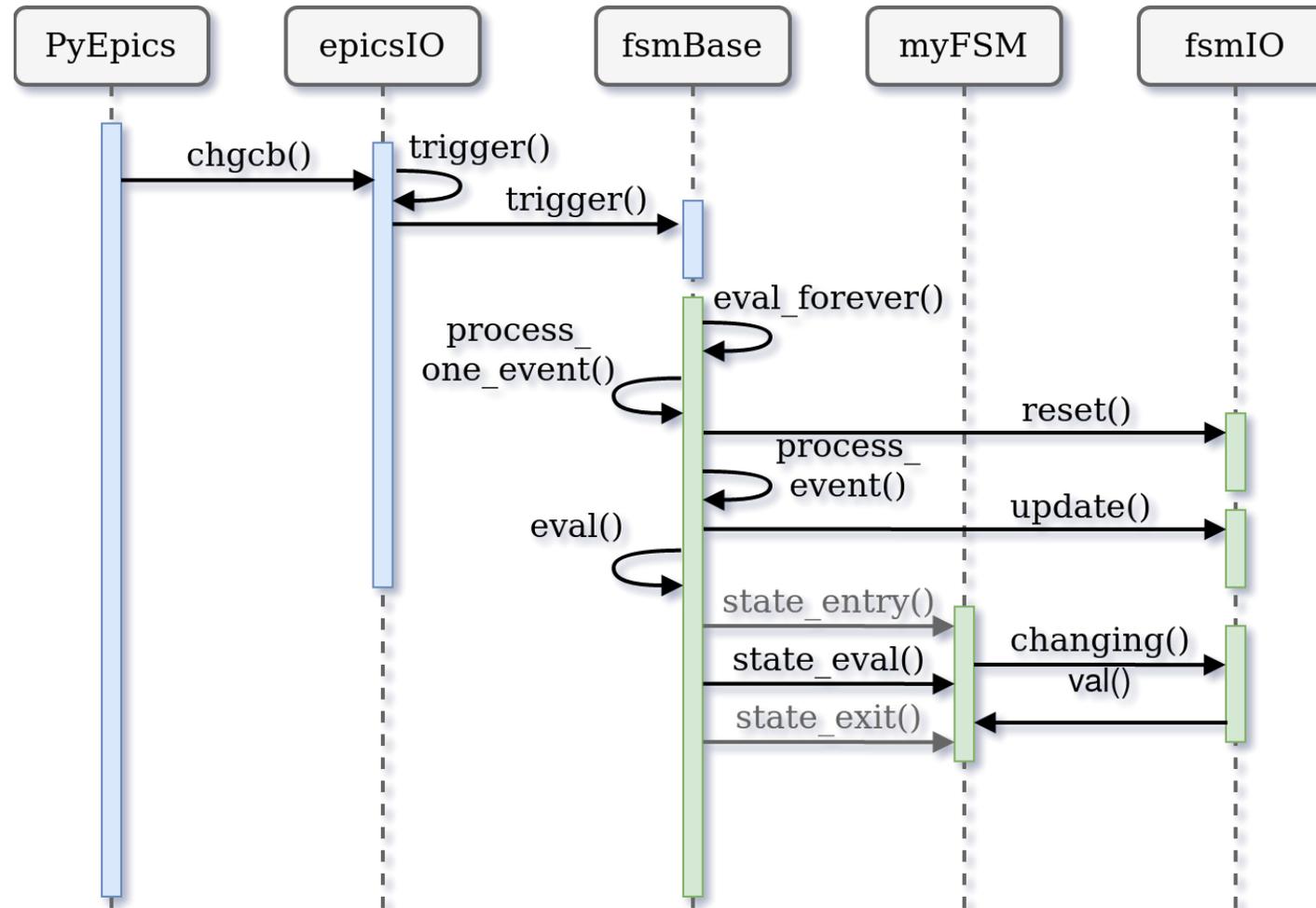
1. Perform a state transition if required. In this case it also executes the `_entry()` method of the new state, if it's defined.
2. Execute the `_eval()` method of the current state.
3. If the user requested a state transition, the `_exit()` method of the current state is executed. In this case go back 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(), 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

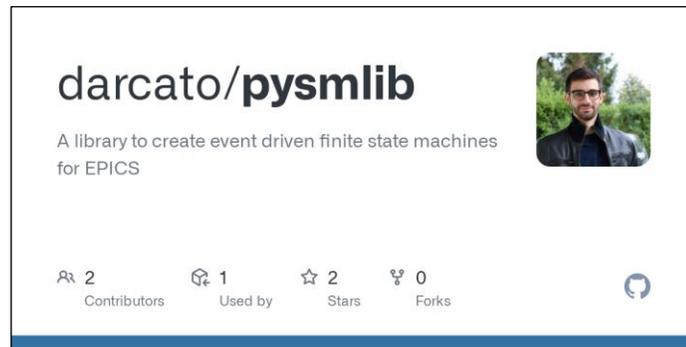
- Specify PV as watchdog
- A thread periodically writes a value
- The PV goes into alarm if no writes occur after a delay

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/pysmlib>



The image shows the GitHub repository page for `darcato/pysmlib`. It features the repository name, a description: "A library to create event driven finite state machines for EPICS", and a profile picture of the author. Below the description, there are statistics: 2 contributors, 1 used by, 2 stars, and 0 forks. At the bottom right, there is a GitHub logo.

pipeline passed coverage 63.00%

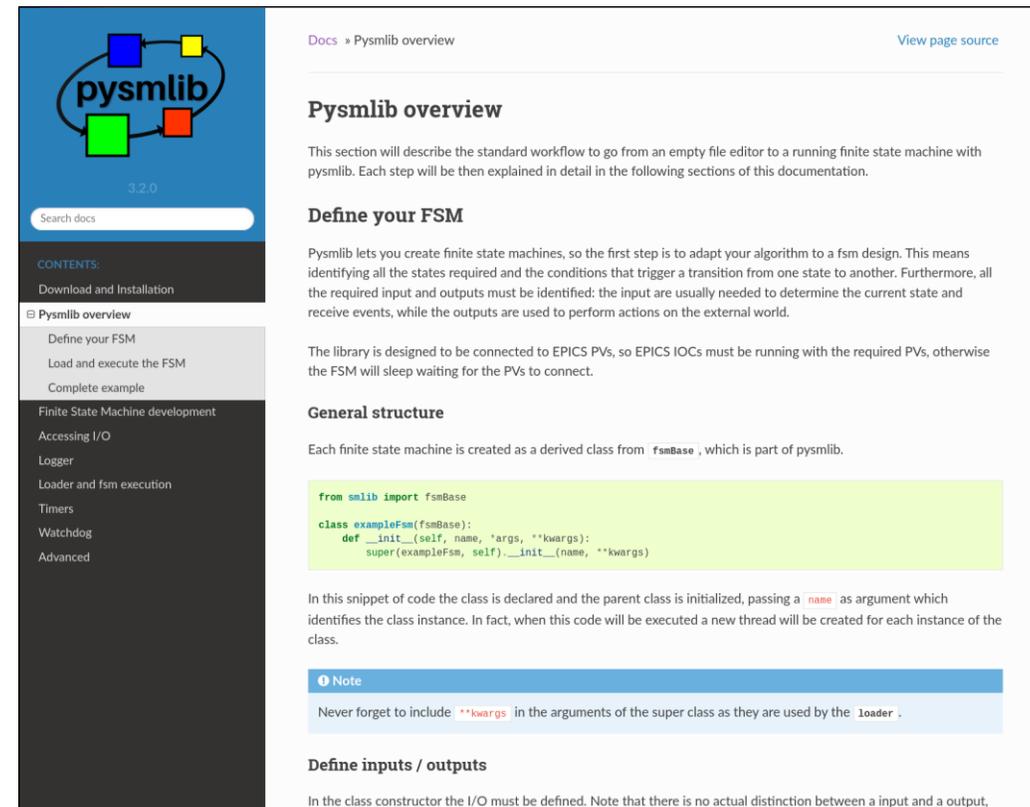
 GPL-3.0 License

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



The image shows the PyPI package page for `pysmlib 3.2.0`. It features the package name and version in large white text on a blue background. Below it, there is a code block with the command `pip install pysmlib` and a small icon of a terminal window.

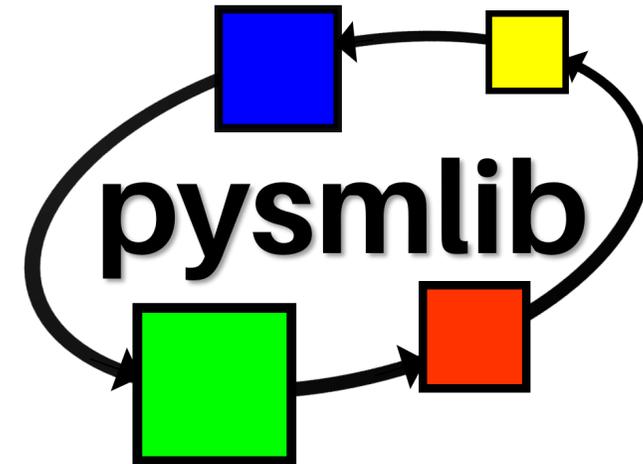
<https://darcato.github.io/pysmlib>



The image shows the documentation page for `pysmlib` overview. It features a blue header with the `pysmlib` logo and version 3.2.0. Below the header, there is a search bar and a table of contents. The main content area is titled "Pysmlib overview" and contains text describing the standard workflow to go from an empty file editor to a running finite state machine with `pysmlib`. It also includes a section "Define your FSM" with a code snippet and a "General structure" section. A note at the bottom states: "Never forget to include `**kwargs` in the arguments of the super class as they are used by the `loader`."

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

