ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-TUBL05

Pysmlib: A PYTHON FINITE STATE MACHINE LIBRARY FOR EPICS

D. Marcato[1,3,*], G. Arena[1], M. Bellato[2], D. Bortolato[1], F. Gelain[1], G. Lilli[1], V. Martinelli[1], E. Munaron[1], M. Roetta[1], G. Savarese[1],
[1]INFN Legnaro National Laboratories, 35020 Legnaro, Italy
[2]INFN Padova Division, 35131 Padova, Italy
[3]Department of Information Engineering, University of Padova, 35131 Padua, Italy

Abstract

In the field of Experimental Physics and Industrial Control Systems (EPICS) [1], the traditional tool to implement high level procedures is the Sequencer [1]. While this is a mature, fast, and well-proven software, it comes with some drawbacks. For example, it's based on a custom C-like programming language which may be unfamiliar to new users and it often results in complex, hard to read code. This paper presents pysmlib, a free and open source Python library developed as a simpler alternative to the EPICS Sequencer. The library exposes a simple interface to develop event-driven Finite State Machines (FSM), where the inputs are connected to Channel Access Process Variables (PV) thanks to the PyEpics [2] integration. Other features include parallel FSM with multi-threading support and input sharing, timers, and an integrated watchdog logic. The library offers a lower barrier to enter and greater extensibility thanks to the large ecosystem of scientific and engineering python libraries, making it a perfect fit for modern control system requirements. Pysmlib has been deployed in multiple projects at INFN Legnaro National Laboratories (LNL), proving its robustness and flexibility.

INTRODUCTION

The Experimental Physics and Industrial Control Systems (EPICS) [1] is one of the most successful frameworks to develop control systems for physics facilities, being used at major laboratories and experiments all around the world. Its main feature is the implementation of the Channel Access (CA) (and the PV Access in newer versions), a standard protocol where the different parts of the control system can communicate. This provides a standard interface to access all the Process Variables (PV) and works as a hardware abstraction layer. Using this protocol, many components have been developed by the community to provide the core functionalities of a modern control system, like Phoebus [3] and React Automation Studio [4] for Graphical User Interfaces (GUI) or the Archiver Appliance [5] for historical data storage.

The sequencer is the tool proposed by the EPICS core developers to implement high level procedures and Finite State Machines (FSM) for process automation. This is an extension of the core software and was first proposed in 1991 in the EPICS paper [1] and originally developed at the Los Alamos National Laboratory. It defines a C-like language called State Notation Language to develop finite state machines which is transcompiled to C code and then compiled to machine code. The user can define states and transitions, while the sequencer takes care of low-level details like the connection with the Channel Access, the event handling and concurrency. Finally, the code is usually run as part of a EPICS input output controller (IOC), which is the piece of software which defines and publishes the PVs on the Channel Access.

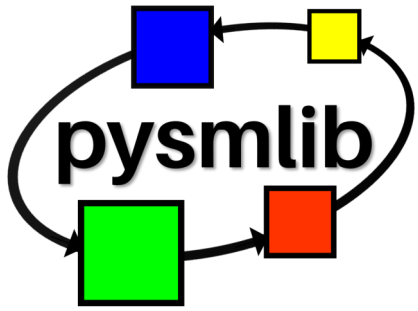

Figure 1: Pysmlib logo.

This software has proved valuable and has been widely adopted thanks to its good performance, seamless integration with the Channel Access and the IOC, and its programming model. Even so, some of its limitations have emerged over time. For example, being C-based was a great advantage at the beginning since it means that one could extend it with any C/C++ library. Today, higher level languages are preferred for this kind of high level tasks, and performance is no longer a limiting factor in most cases. For this reason Python has emerged as one of the most prominent languages for modern scientific and engineering computing, thanks to a large number of dedicated libraries. Also, Python appeals to a broader audience of less-technical programmers.

The PyEpics [2] python library, which wraps the original `libca` C library, became thus a popular alternative to communicate with the Channel Access. This can be used to write both simple scripts and full blown programs. Large experiments or collaborations used this, or similar wrappers, to build tightly integrated high level suites which handle automation and much more at facility level, such as ophyd and bluesky [6]. These are great solutions, but require a big investment into their design model, which could not be ideal for simple tasks or small independent laboratories. Also, at this level there is a lot of fragmentation in the community, with no default go-to solution but many different approaches tailored to the needs of specific laboratories.

* davide.marcato@lnl.infn.it, www.davide.marcato.dev

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-TUBL05

In the middle between vanilla PyEpics scripts and full experiment-wide software, there is still the need for a generic standalone solution to develop finite state machines, where the execution flow, event handling and connection with the channel access is provided. Pysmlib aims to recreate the sequencer programming model and its strengths, while taking advantage of the python language and focusing on ease of use.

This paper formally presents the library to the scientific community, describing its design and going into details on the implementation and execution flow. Then an overview of the user interface is presented and finally the current status of the project and some plans for the future are outlined.

DESIGN PRINCIPLES

A Finite State Machine is a computational abstraction where the program behaviour depends on its current state and the input it receives. The machine can perform a transition to a different state in response to inputs. For example the schema in Fig. 2 represents a simple FSM with 3 states S_1, S_2, S_3 and 2 inputs x_0, x_1. The execution starts from S_1 and while x_0 is 0 the current state does not change. When x_0 becomes 1, the FSM executes a transition to S_1. Now the FSM follows the value of x_1 by staying in S_1 when its value is 0 and moving to S_2 when it becomes 1. In S_2 the FSM can go back to S_0 if x_0 becomes 0 or to S_1 when x_1 is 0.

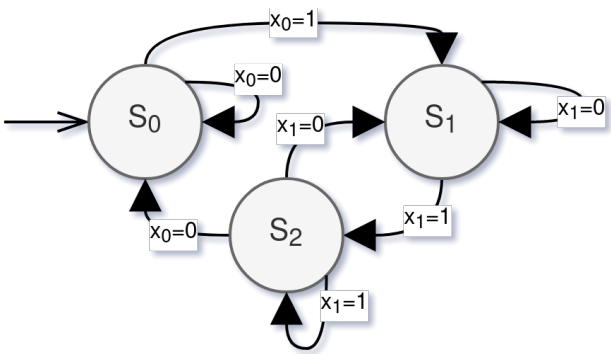

Figure 2: Schematic representation of a FSM with 3 states and 2 inputs.

This kind of abstraction is extremely useful to design control system algorithms, where the controlled system can be described with states and transitions, and the control algorithm can be tuned for the current state.

Pysmlib aims to provide a simple way to describe Finite State Machine states and transitions, while handling all common tasks and minimizing the custom code required. Thus, it keeps track of the current state and executes it in response to events and takes care of connecting the FSM to the inputs, while providing the user some methods to access them. The term *input* is here used both for input (readable) and output (writable) signals. In fact, in pysmlib they are an abstraction of Channel Access PVs, even though the library is designed so that a future expansion to different types of inputs is relatively easy and requires as little updates to the user interface as possible. On top of the methods to read and write PVs provided by PyEpics, the methods to detect rising and falling edges or changes are provided, which are particularly useful for the design of a FSM. Then, the user can implement the FSM states as methods of a class, which represents the FSM, with a specific naming convention and the library is able to automatically discover them. When performing a transition from one state to another, the user can decide to implement *entry* and *exit* methods for each state, which are executed only once during the transitions, as can be seen in Fig. 5.

Pysmlib is designed to maximize network efficiency and system responsiveness, in order to be able to react with minimum latency to input events. Thus, all the inputs are connected at startup time, and the connection is kept in memory during the whole execution. The current state is executed every time a channel access event is received, in a event driven way, without relying on a periodic loop. For example, when an input changes its value or its connection status, the current state is re-evaluated so that the FSM can react to the new value, while the state is never executed if no event is received. To optimize network usage, multiple FSMs can be loaded and run in multi-threading while sharing the connection to common PVs and a dispatcher is used to broadcast channel access events to each FSM. The user is expected to develop multiple FSMs, load them together on a single *daemon*, using the provided loader helper, and let it run continuously. This approach comes natural for always-on operations (eg: a feedback correction) but it is also the ideal execution flow for procedures that start from a user input or some external conditions: in these cases the FSM will wait on a *idle* state where no action is performed until the enable signal is received, execute its core procedure, and then come back to idle. This means that when the enable arrives, all the inputs are already connected and the user doesn't have to wait for the FSM startup and all the connection times.

To ensure consistency, the inputs should not change value or status during the execution of a state. For this reason a queue of events is used and they are evaluated in order one by one, where each event triggers one execution of the current state of all the FSMs connected to such input. Along with input events, the user can also set *timer* events, so that the current state is executed after a custom amount of time. This is useful to set timeouts or to execute periodic actions. Other features include logging methods to write FSM logs in a unified way, and a watchdog implementation. In fact, contrary to the sequencer where the code is usually executed on the same IOC that defines the PVs, here the FSM daemons can run everywhere on the channel access network, and most probably will run independently from the IOC and from the user interface. A method to assert if the FSM is online and running is thus needed, since the control system may rely on the FSM to work correctly. For this reason a watchdog is implemented, where the FSM will periodically write to an external PV. The PV is then configured to raise an alarm or an error when the write operation does not happen regularly.

Finally, pysmlib is developed as a generic library with no dependency on the task to perform, and aims to be useful

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-TUBL05

for a large audience of control system developers. It offers an online documentation and follows open source standards, where the users are encouraged to contribute.

SOFTWARE ARCHITECTURE

In this section the internal architecture of the library is detailed, along with a description of the execution flow of a FSM. Figure 3 shows a summary of the main classes with their attributes and methods. These can be grouped into 4 modules, as highlighted by the color of the tables. Each module concerns with one of the main features of the library: access to the inputs (green), FSM execution (blue), timers (orange) and other smaller utilities (red). A complete description of each module follows.

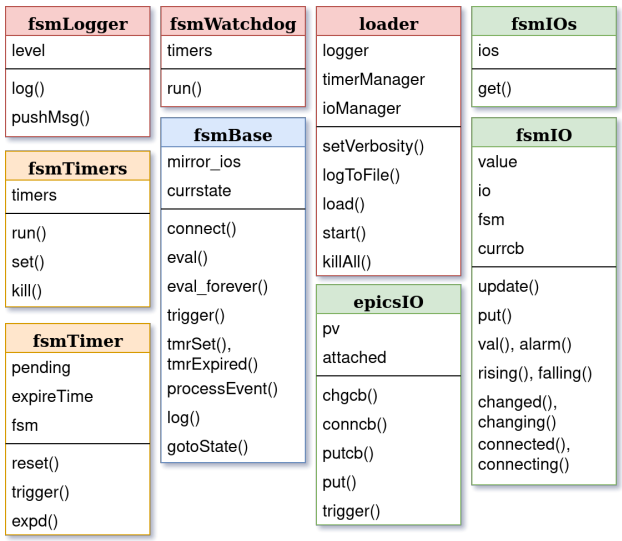

Figure 3: Diagram of the main classes and their most important attributes and methods.

Input Management

The Channel Access defines some methods to access the Process Variables: it's possible to perform a `get` (read) or a `put` (write) operation, but also to `monitor` a PV, that is register to the IOC server to be notified about PV events. There are 3 types of events: *change* events are emitted when the PV has a new value, *connection* events when the connection status changes and *put complete* events when a put operation completes. With PyEpics it's possible to register a callback to be executed when one of these events occurs, so that the event can be processed.

Pysmlib defines the `epicsIO` class which is responsible to keep the connection with a PyEpics `PV` object and registers the callbacks on the Channel Access events. When an event arrives a `trigger` method is executed, which will pass a copy of the event data to all the attached FSMs. Each FSM has a queue of events where this data is temporarily placed, while waiting to process it. All of this is done on the callback thread, while the queue consumption is done on the FSM execution thread, so the queues must be thread safe. Furthermore, this class is also responsible to perform the `put` operations, which are directly executed by the FSM, since the Channel Access already implements a queuing system. The `epicsIO` class is instantiated when a FSM connects to a new PV. To avoid duplicating connections to the CA when different FSMs connect to the same PV, a helper class (`fsmIOs`) is used. This class keeps track of the available inputs and creates a new `epicsIO` instance only when required.

One important requirement is that an input does not change during the execution of a FSM state. To achieve this, every time a FSM connects to an input it creates a corresponding instance of `fsmIO`, a class which represents a local copy of an input to each FSM. The execution follows this pattern: first an event is removed from the queue, then the local copy of the corresponding input is updated using the event data and finally the current state is executed. This pattern is repeated until there are no more events in the queue and the thread waits idle for new ones. So each event brings the update of a single input and triggers a single execution of the current state. If the original IO changes during the state execution, the local copy is not affected.

Figure 4 shows an example of the FSM execution flow in response to a *change* event from an input. First, the registered callback is executed by the PyEpics library and all the FSMs connected to the corresponding input are triggered. Then, the event data is queued on the event queue and the thread of the callback (blue in the figure) completes its execution. Now, the FSM thread (green in the figure) awakes and consumes the available events one by one. In order to keep the instance of `fsmIO` synchronized with the original input, first the `reset()` method is called, which removes old data, and then the `update()` one which feed the new information. Finally, the current FSM state is executed by calling the user-defined methods.

In the code, the user can inspect the new received value thanks to the `fsmIO` class, which implements all the methods to read and write the value and metadata of the input. For example, it's possible to read the current value with `val()`,

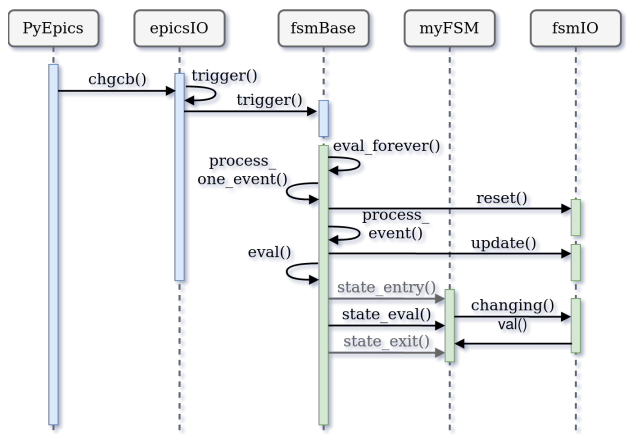

Figure 4: Execution flow of the FSM current state in response to an event from an input.

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-TUBL05

connection status with `connected()` and the alarm severity with `alarm()`, and to write the value with `put()`. Other methods allow to detect rising or falling edges, or other transient properties of the input. For example, we want that the `connecting()` method returns `True` only during the state execution that was triggered by the connection event, and not on the following ones. Conversely, the `connected()` method will continue to return `True` until the input disconnects. This is especially useful for the design of a FSM to perform some actions just once when a condition is met. To implement this, the type of the current event is used to understand what kind of event triggered the state execution. For example, an input is `changing()` when the current event is of *change* type, while it is `rising()` when the current event is of *change* type and the new value is greater than the preceding one.

Finite State Machine Execution

The FSM execution flow is managed by the `fsmBase` class. The user is expected to derive from this class to implement his own specific FSM. In the constructor of the derived class the user will connect to all the inputs using the `connect()` method, and will specify the first state to execute. This can be done using the `gotoState()` method which accepts a string with the state name as argument. After that the user will implement the states as methods of the class. As illustrated in Fig. 5, for each state the user will implement a *eval* method and optionally the *entry* and *exit* methods which are executed only once during the state transitions. To be correctly recognized, the methods must be called by concatenating the state name to `_eval()`, `_entry()` or `_exit()`. For example, for a state called *idle*, the user must define the method `idle_eval()` and can optionally define the methods `idle_entry()` and `idle_exit()`.

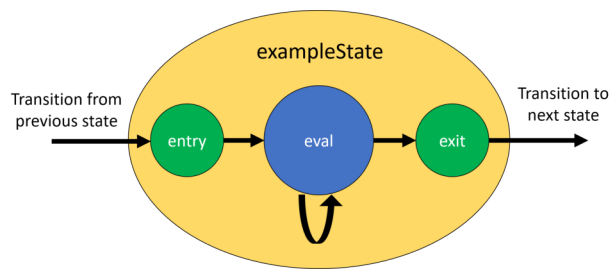

Figure 5: Entry, eval and exit methods can be implemented for each FSM state.

The `fsmBase` defines a thread which runs the FSM states. This thread initially waits on the event queue for some events to populate it. When an event arrives, it is removed from the queue and processed in order to update the `fsmIO` instance of the corresponding input. Then the `eval()` method is called, which is responsible for the actual state execution. This method performs the following steps:

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.

The user can call `gotoState(new_state_name)` inside the state `_eval()` method to perform a transition. In this case, the execution loop is restarted from step 1 without processing a new event, so that the new state is evaluated once without waiting. The `gotoState()` function simply saves the name of the next state and uses the `getattr` python function to find the methods with the corresponding names. The actual transition is executed in step 1 by updating the pointer to the current state.

Other methods of the `fsmBase` class can be used to check conditions on multiple inputs together or to retrieve the input responsible for the current execution of the state. Furthermore, the class is used as a single interface for timers and the logger, as explained in the following paragraphs.

Timers

Timers are used to schedule internal events, where the current state is executed after a fixed amount of time. They can be used to perform asynchronous operations such as periodic writes or to enforce a timeout when waiting for an external condition to become true. The user can set a timer with the `tmrSet()` method of `fsmBase`, selecting an identifying name and an expiration time in seconds, while the methods `tmrExpired()` and `tmrExpiring()` can be used to verify if a timer has expired in the past or in the current event.

A single timer is represented by the class `fsmTimer`, which keeps track of the expiration status. The class `fsmTimers`, instead, is used to manage all the timers of all the FSMs. This is a thread which keeps a list of timers ordered by their expire time. When the FSM sets a timer, this is added to the list in the correct place. The thread schedules a waiting time equivalent to the remaining time of the first timer in the list, and then triggers the corresponding FSM by appending a *timer expired* event to its event queue. After that it removes the timer from the list and schedules a new waiting time. When the list is empty the thread waits idle until a new timer is set.

All the timers, inputs, and each FSM live on different threads, and thus create a complex multi-threading execution flow. With `pysmlib` this complexity is completely hidden from the user, which gets a well-tested core to build upon, without worrying about low level details.

Utilities

On top of the core functionalities, the library offers useful utilities to simplify the user code on some common tasks. For example, the `loader` class provides methods to create a launcher file which executes many FSMs in parallel. In fact, after the development of the FSM classes, it's often useful to run different instances of the same FSM with different

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-TUBL05

parameters, or simply to launch together correlated FSMs to share common resources. The `load` method of the `loader` class takes care of instantiating a FSM. It lets the user specify custom parameters to be passed to the constructor of the FSM, but it adds some system parameters used to share common resources. For example, a single instance of `fsmIOs` is created and passed to all loaded FSM in order to share the inputs. The same mechanism is used to share the timer manager thread.

The `loader` class is also used to configure log messages. In fact, the library provides an abstraction of a generic log facility which can be configured to send messages to different backends with a unified interface. At load time the user can specify to log to file (`logToFile()`) or standard output (default) and can set the verbosity level with the `setVerbosity()` method, choosing between *debug, info, warning* and *error* levels. Likewise, the `fsmBase` class provides the `logD()`, `logI()`, `logW()`, `logE()` methods that can be used to write log messages with the corresponding level. The logging facility is provided by the `fsmLogger` class, which is currently a custom class with few options, but can easily be expanded or replaced by the `logging` module of python without any changes to the user interface, thus minimizing the impact on the existing code.

Another shared resource is the `fsmWatchdog` class, which provides the ability to periodically write to an external PV while a FSM is running. This class is derived from `fsmBase` and implements a specific FSM which periodically writes the watchdog of all running, user-defined FSMs. The user can define an input of an FSM to be used as the watchdog PV by calling the `setWatchdog()` method of `fsmBase` on the constructor of each FSM, and this will be automatically passed to an instance of `fsmWatchdog` at load time. The user can also specify the interval and the value to be written for each watchdog, choosing between `0`, `1` or an intermittent value. The external PV should be configured so that when a write operation does not occur after a specified time interval, it raises an alarm. This can easily be achieved with the `HIGH` field of a binary output PV.

DEVELOPMENT TOOLS

The development of the library adheres to modern standards for python projects. The code is well formatted and documented, respecting the `pylint` indications. To ensure easier updates to the core library automated tests have been developed. These simulate the most common operations and check if the expected result is achieved, so that the developer can easily notice when an update or a new functionality introduces bugs. The tests were developed using the `pytest` [7] framework, and can be run directly with pytest or using `nox` [8], a tool to automate test executions in different environments. In fact, given a list of python versions, it takes care of creating a different conda environment for each version, install the dependencies and run the tests. This way, it's easy to verify that the code continues to run correctly on all supported versions of python. To simulate Channel Access connectivity the `pcaspy` module is used, which helps running a PV server from python, without firing up a complete EPICS IOC. Finally, the nox tests are run automatically on the Gitlab CI [9] every time the code is pushed to the server, so that the developer can be alerted of failures.

Since the library was developed with a broad and generic audience in mind, it has also been published as free and open source code on Github [10] with a GPLv3 license, which is compatible with the EPICS open license. The library is packaged as a standard python module using `setuptools` and is available on the PyPI [11] python software repository, so that it can be installed with just one command. Version strings are managed using `versioneer` [12], a tool that helps defining automatically the current version based on the underlying git tag.

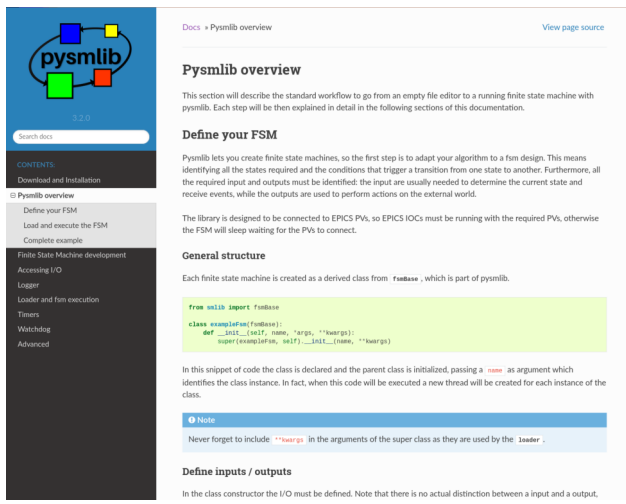

Figure 6: Pysmlib online documentation.

Web-based documentation was built using `sphinx`, which generates beautiful HTML pages from reStructuredText files. The resulting website can be seen in Fig. 6 and includes some examples, a tutorial, and the detailed API description. Finally, the HTML pages are hosted using Github Pages [13], so that they are always available online.

USER EXPERIENCE

Pysmlib was first envisioned and developed for the radio frequency (RF) control system [14] of the ALPI linear accelerator at the INFN Legnaro National Laboratories (LNL). This system requires a lot of procedures to power up, tune and lock each cavity which were previously performed manually. With this library it was possible to automate lots of tasks, thus reducing the time and effort required to setup the accelerator for a run.

Given the success in its first use case, the original code was decoupled from the RF control system to create a general purpose library that could be easily used on different projects. Thus, version `v2.0` introduced the *pysmlib* name and the library was published online, complete with documentation. Version 3 dropped the support for Python 2, which was being deprecated, and further consolidated the user interface,

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-TUBL05

adding methods to access the alarm state of an input and improving the automated tests.

The library has been integrated into multiple projects at Legnaro to automate different tasks, from simple ones like acquiring the spectre of the beam using a magnetic dipole to more sophisticate ones like online beam emittance measurement or BOLINA [15] beam trajectory optimization algorithm. Another common use case is to build simulators, where the FSM reacts to control PVs simulating the effect of a real object, which is useful to test the control system software. For example, a FSM was developed to simulate a wire scanner diagnostic complete with its handling motor. A FSM can also be used to build alarm handlers, like `telegram-epics-bot` [16] which sends a Telegram message to selected users when a PV enters the alarm state. In general pysmlib is useful whenever it is necessary to develop programs which are not executed one-shot, but are always running waiting for the correct condition to execute their task. In fact, the ability to detect edges on the PV values is extremely useful for this scenario and with this library it is provided by default.

In listing 1 an example of a minimal complete FSM is provided, where the FSM is expected to copy the value of a *counter* PV to a *mirror* one, when the *enable* PV is non-zero.

```
#! /usr/bin/python
from smlib import fsmBase, loader

# FSM definition
class exampleFsm(fsmBase):
  def __init__(self, name, *args, **kwargs):
    super().__init__(name, **kwargs)
    self.counter = self.connect("counter_pv_name")
    self.mirror = self.connect("mirror_pv_name")
    self.enable = self.connect("enable_pv_name")
    self.gotoState('idle')

  # idle state
  def idle_eval(self):
    if self.enable.rising():
      self.gotoState("mirroring")

  # mirroring state
  def mirroring_eval(self):
    if self.enable.falling():
      self.gotoState("idle")
    elif self.counter.changing():
      readValue = self.counter.val()
      self.mirror.put(readValue)

# Main
if __name__ == '__main__':
  # load the fsm
  l = loader()
  l.load(exampleFsm, "myFirstFsm")

  # start execution
  l.start()
```

Listing 1: Example of a FSM implementation.

The FSM is implemented by defining a class which derives from `fsmBase`. In the constructor the user can connect to any amount of PVs, addressing them with their name and then specify the first state to be executed. After that, the example shows the definition of two states, called *idle* and *mirroring*. These are implemented as methods of the class, with the `_eval` suffix in their name. Initially the FSM evaluates the idle state, until the enable PV rises from 0 to 1, which causes the FSM to execute a transition to the mirroring state. In this state the enable PV is continuously checked, so that when an external user revert it back to 0 the FSM goes back to the idle state. Until then, the mirroring state listens to change events to the counter PV and copies the new value to the mirror one. This is done with the `changing()` method to avoid repeating continuously the put operation with the same value, which could slow down the Channel Access, and instead execute a put only once when the counter value changes.

At the end of the example, the loader usage is demonstrated. In this simple case a single FSM is loaded, with just its name as parameter, and the execution is started with the `start()` method. This is by default a blocking call and the execution can be halted using a keyboard interrupt.

CONCLUSIONS AND FUTURE IMPROVEMENTS

Pysmlib is now a stable project and aims to provide a robust solution to the EPICS community to develop modern Finite State Machines. The focus on simplicity and the choice of the Python language make it a good choice for both new and advanced users, who may take advantage of the vast amount of Python scientific libraries.

While the user interface can be considered stable, future updates will focus on improving the internal components to improve the quality and expandability of the code. For example, the current logger facility should be deprecated in favor of the Python logging module to gain all of its functionalities. A more object-oriented approach to events could help to introduce more easily different kind of external events, and thus different kinds of inputs. Currently only the Channel Access inputs are supported, but an expansion to the PV Access could be helpful to users of this newer EPICS protocol. The library could also be generalized to work as a generic Finite State Machine engine, where the user is able to define its custom inputs and connect them to any kind of external system. For this to work, a rigorous I/O interface should be defined and documented.

Other improvements could be evaluated following user suggestions, based on real-world use cases. Contributions and fixes are welcome.

REFERENCES

[1] L. R. Dalesio, M. R. Kraimer, and A. J. Kozubal. “EPICS architecture”, in *Proc. ICALEPCS’91*, 1991.

[2] M. Newville, *et al.*, pyepics/pyepics Zenodo. `doi:10.5281/zenodo.592027`

[3] CS-Studio (Phoebus), `https://controlssoftware.sns.ornl.gov/css_phoebus/`.

[4] W. Duckitt, J.K. Abraham, “React Automation Studio: A New Face to Control Large Scientific Equipment”, in *Proc.*

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-TUBL05

Cyclotrons'19, Cape Town, South Africa, Sep. 2019, pp. 285-288. doi:10.18429/JACoW-Cyclotrons2019-THA03

[5] M. Shankar, M. Davidsaver, M. Konrad and L. Li, "The EPICS Archiver Appliance", in *Proc. 15th International Conference on Accelerator and Large Experimental Physics Control Systems (ICALEPCS 2015)*, Melbourne, Australia, October 17-23, 2015. doi:10.18429/JACoW-ICALEPCS2015-WEPGF030

[6] Bluesky Project and Ophyd, https://blueskyproject.io/.

[7] Krekel *et al.*, pytest 6.0.1, 2004, https://github.com/pytest-dev/pytest

[8] Nox, flexible test automation for Python, https://nox.thea.codes/en/stable/.

[9] Gitlab CI, https://about.gitlab.com/stages-devops-lifecycle/continuous-integration/.

[10] : D. Marcato *et al.*, Pysmlib Github repository, https://github.com/darcato/pysmlib

[11] : D.Marcato *et al.*, Pysmlib on PyPI software repository, https://pypi.org/project/pysmlib/.

[12] B. Warner *et al.*, The Versioneer, https://github.com/python-versioneer/python-versioneer

[13] : D. Marcato *et al.*, Pysmlib online documentation, https://darcato.github.io/pysmlib/docs/html/,

[14] D. Bortolato *et al.*, "New LLRF control system at LNL", in *2016 IEEE-NPSS Real Time Conference (RT)*, 2016, pp. 1-8. doi:10.1109/RTC.2016.7543105

[15] V. Martinelli, *et al.*, "High Level Software for Beam 6D Phase Space Characterization", in *Proceedings of the 10th International Particle Accelerator Conference (IPAC2019)*, Melbourne, Australia, 2019. doi:10.18429/JACoW-IPAC2019-WEPGW025

[16] D. Marcato, Telegram EPICS bot, https://github.com/darcato/telegram-epics-bot