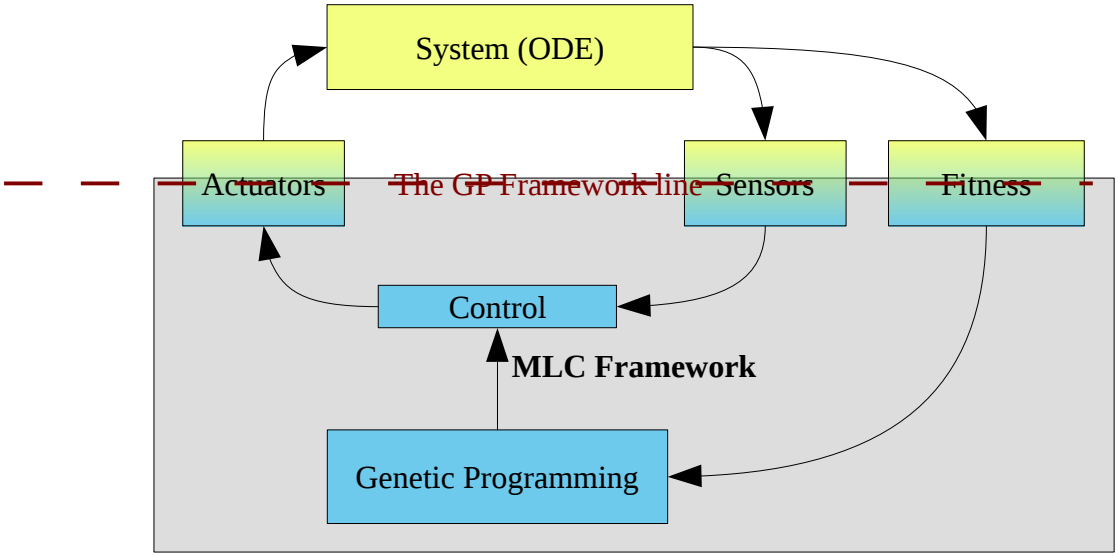
MLC (Machine Learning Control)

Description of the generalized framework

Principle:



- The Genetic Programming module will generate candidate control functions that will be tested on the System via the actuators and sensors.
- Each candidate control function will be given a quality value: the fitness value (or cost value).
- This process is iterated until the maximum number of trials has been reached or until the optimal control function has been found.

Technical description:

Interface between the MLC framework and the System:

The MLC Framework is a Java application. To make it able to communicate with a wide variety of systems, we use a modular architecture based on the “observers” design pattern.

On the MLC side, the sensors, actuators and fitness are objects that can be notified or notify about a change. The basic learning scheme is the following:

1. The Genetic Programming generates a new candidate control function.
2. The System changes state → it notifies the sensors that there are new state values.
3. The sensors react and pass the new values to the candidate control function
4. The actuation is computed using this candidate control function.
5. The candidate control function notifies the Actuators that there are new actuation values
6. The Actuator reacts and passes the new actuation values to the System.

7. Loop back to point 2.
8. When the system reached a certain point (max running time, max iterations...), it returns the quality of the candidate control function, with respect to predefined criteria.
9. The quality of the candidate control function is updated and loop to point 1.

Global scheme of actuators, sensors and fitness:

The Actuators, Sensors and Fitness are Java objects that use the Jana Native Interface (JNI) library to communicate with the system. This makes us ables to interact with any system that can provide a C/C++ type library.

The MLC framework only needs to know the prototypes of the functions it will need to call from the System, and the only thing the system needs to provide is a header file with the prototypes of the functions it provides to interact.

For example, a very simple system could provide the following methods:

- void actuate(double[] values); // takes the value and actuate on the system.
- void reset(); // resets the system when a new candidate control function is generated

Ideally, the sensors should follow the same principle, but as the communication between the two parts (MLC framework and the System) is unidirectional, the Sensors cannot be notified by he system of a change in state. Therefore, the sensors will “check” the state when it needs to know it. This adds a method that the the system has to provide, which could be:

- double[] getSensors(); // returns a vector of sensor values representing the state of the system

The fitness would work the same way: the Fitness object would get the fitness value of the currently used candidate control function using:

- double getFitness();

The two loops of the System:

In general, in the case of numerical simulations of dynamical systems, we don't want to actuate at every integration step “i”. We want to actuate at every time step “t” with t>i. This is why th System had to be built with two loops: one loops which iterations will be triggered by the call to the “actuate” method, and another loop that will perform “n” integration steps during one iteration of the first loop. Note that “n” can vary, and the length of this second loop can be different for each iteration of the first loop, depending on design choices. An example could be the following:

```
when actuate() is called and max is not reached do
      update_actuation();
      while(change of state < threshold)
            integrate_step();
            update_state();
      end while
      update_sensors():
end
update_fitness();
```

We clearly see in this example that, **for the learning process**, the MLC Framework is the master, as it triggers the integration of the system.