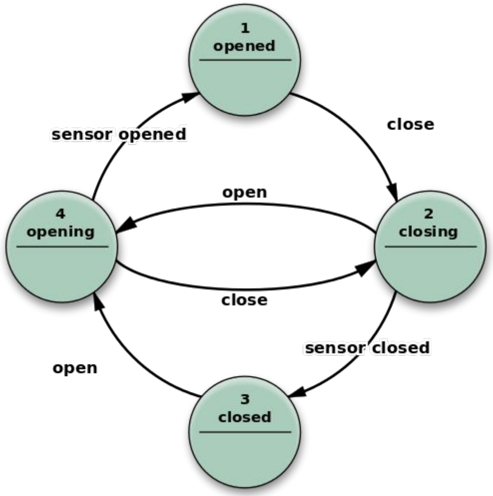
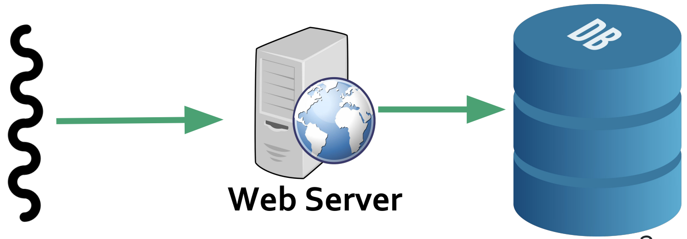
Offline Runtime Verification with Automata

Per Sunde - University of Oslo
Supervised by Joshua Dawes

The Runtime Verification Framework



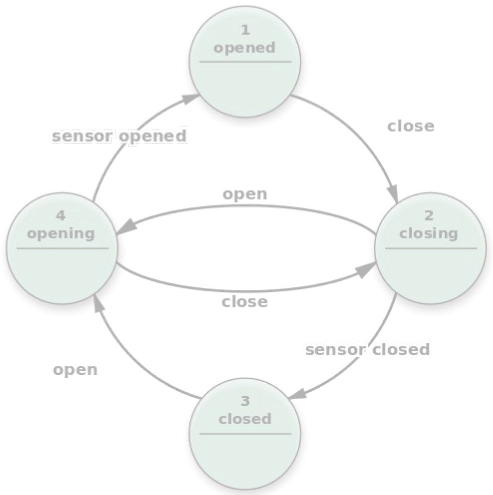
Backend implementation



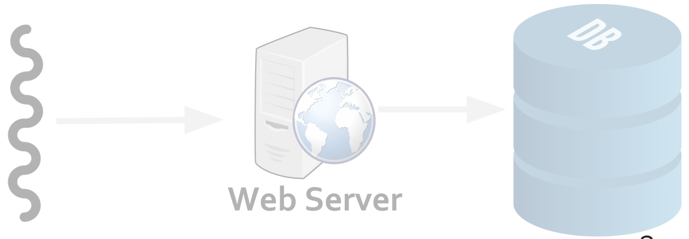
What is

Runtime Verification?

The Runtime Verification Framework

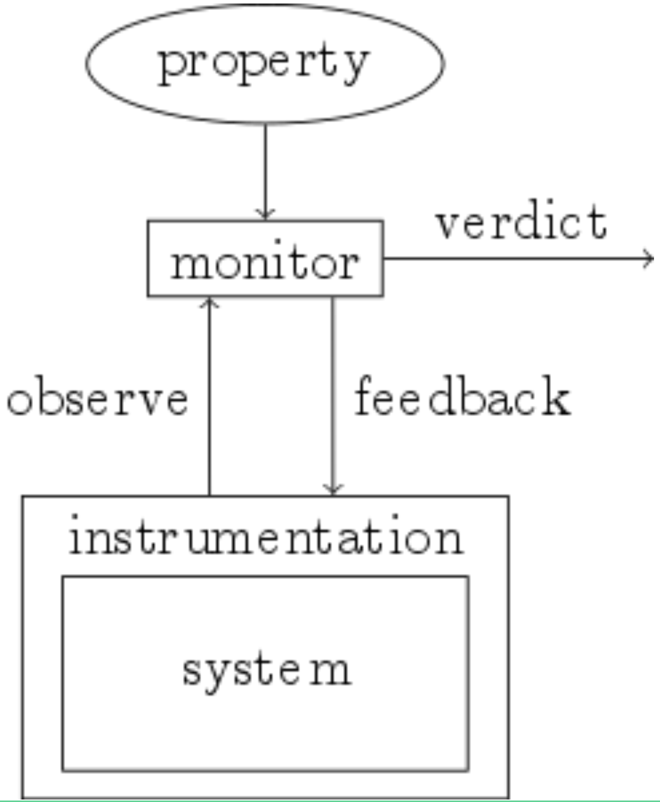


Backend implementation

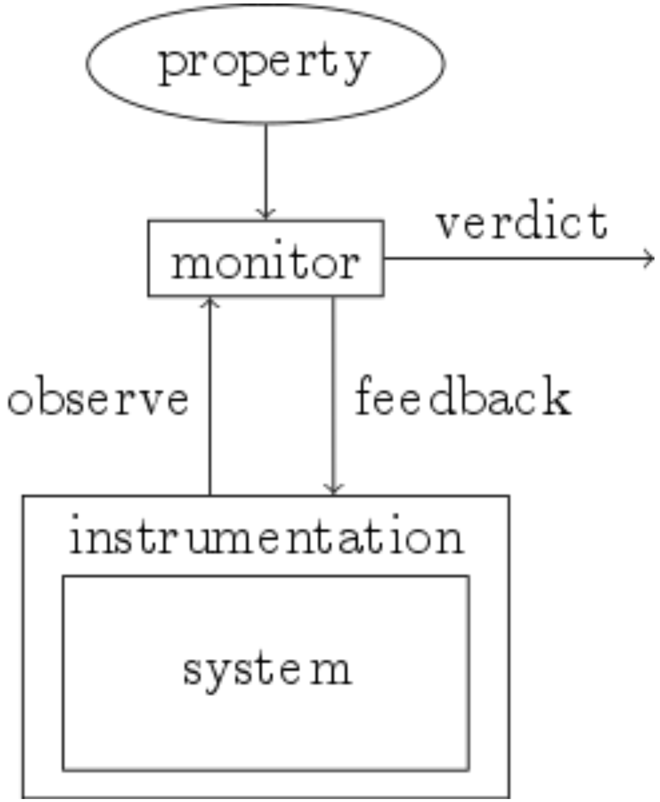


Runtime Verification is the problem to check that the program runs as intended during runtime

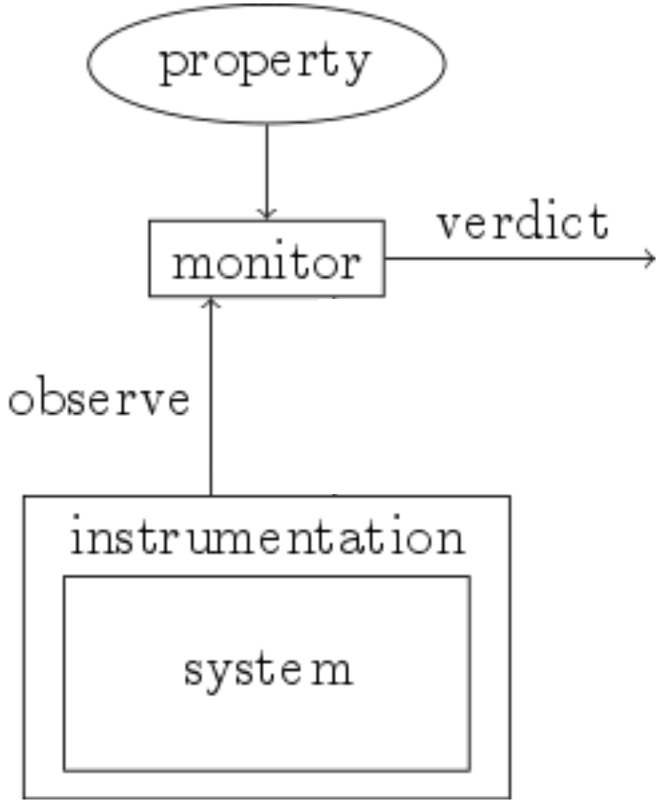
-



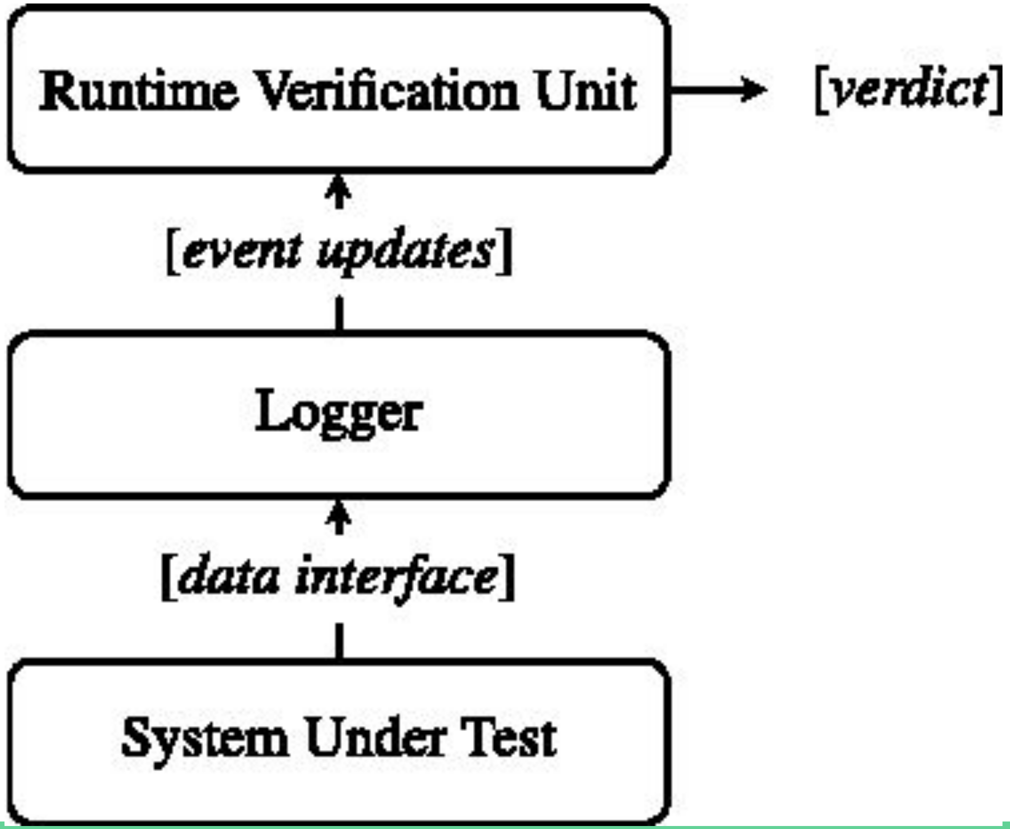
Online



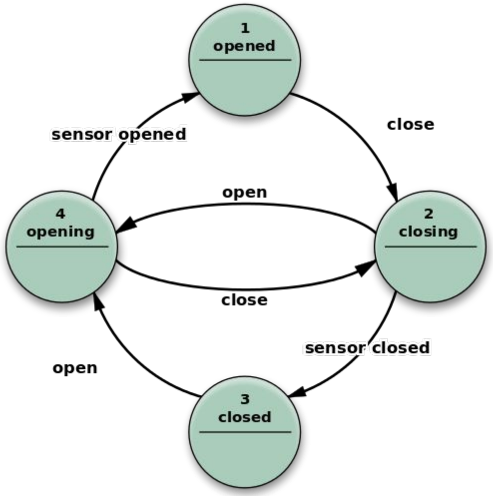
Offline



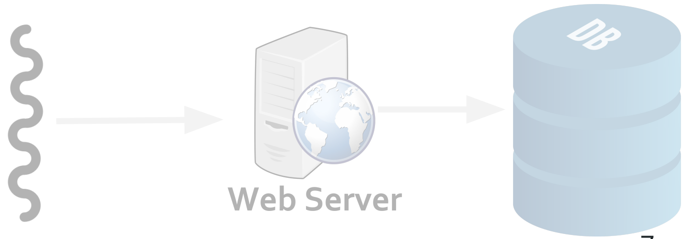
Offline Verification is when you verify the run after it has happened



The Runtime Verification Framework

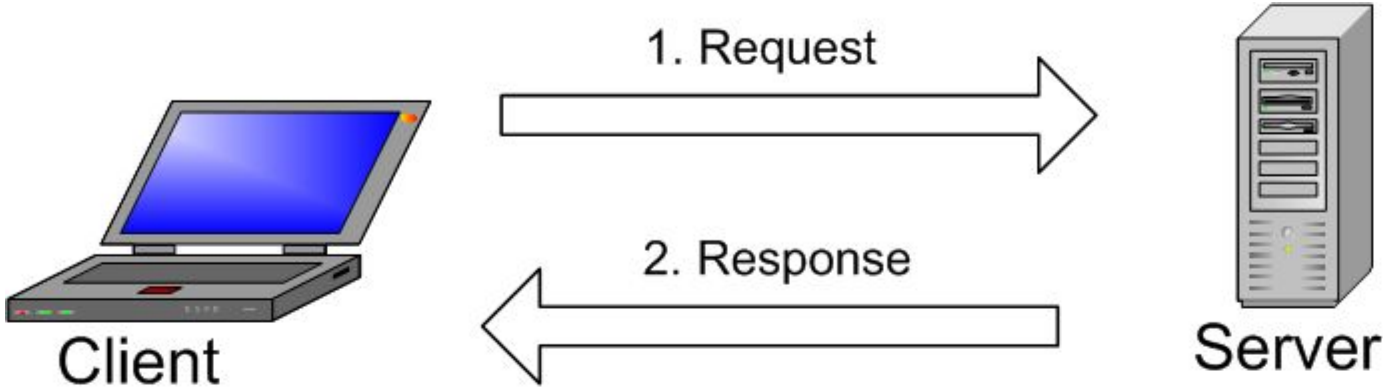


Backend implementation



What is the problem we want to solve?

- Want to check that a program is behaving as intended
  - Using data stored by the monitoring library
  - Written by Josh and myself
- Behaving as intended means:
  - Functions are called in a certain order
  - Functions or a group of functions should complete within a time limit
- How do we do this?
  1. Collect function call metadata from the system during runtime
     a. Name of function
     b. Time of Call
     c. Time Used
     d. etc.
  2. Analyse the metadata
  3. Give a verdict

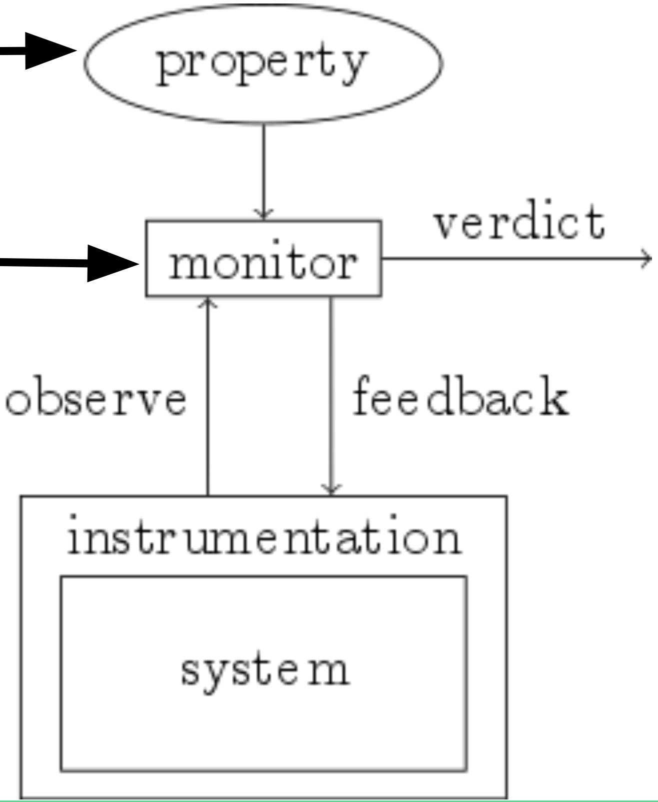

1. Request
2. Response
Client
Server

The Standard way to do Runtime Verification

- Create a logic
  - Based on Lineartime Temporal Logic (LTL) or Timed Lineartime Temporal Logic (TLTL)
  - A logic is a language with an associated semantics
    - Defines truth in logic
- Algorithm that takes properties and creates a Monitor that accepts the “inputs” that hold the properties
  - Usually creates an Automaton from the properties
- Attach the Monitor onto the system
- The Monitor will decide if the system runs are accepted or not

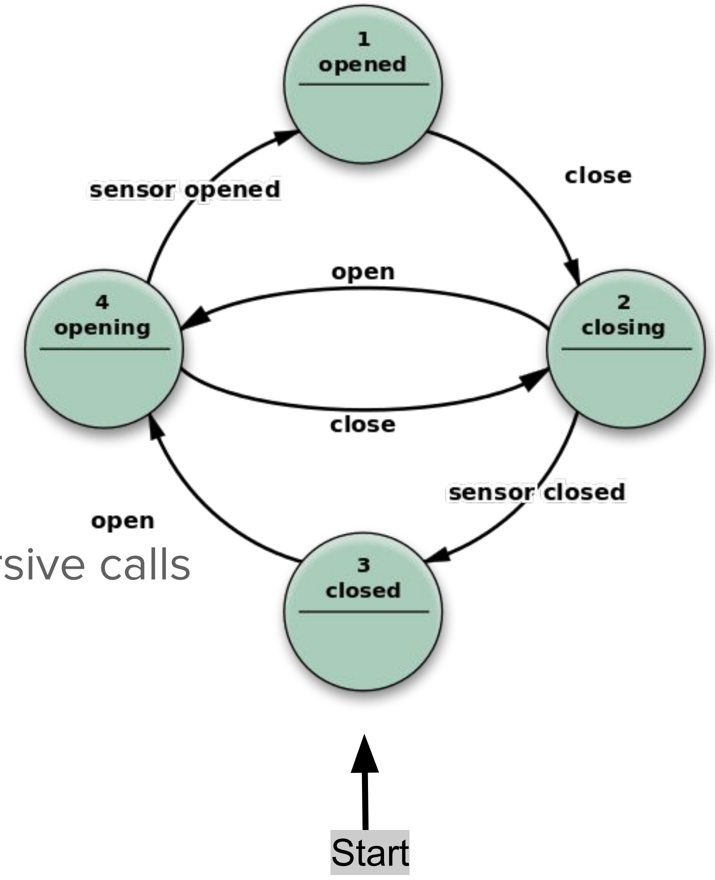
The **Automaton** created

Algorithm that runs the Automaton and based on that gives a verdict



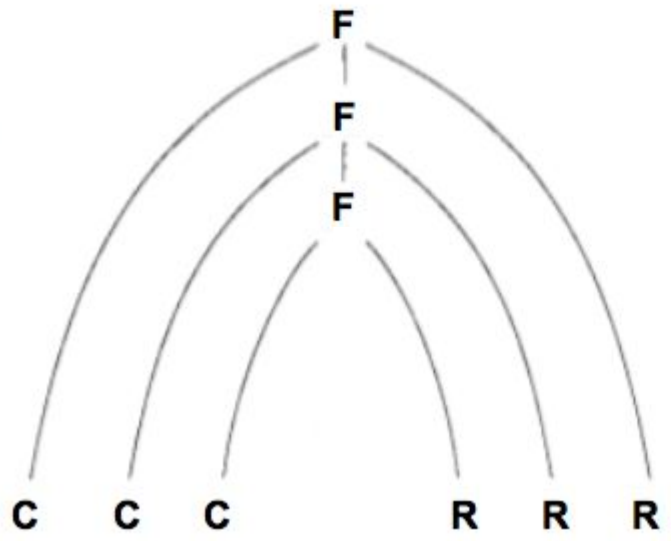
Finite State Automata

- It is a “machine” that will transition between states when it receives input
- It has no memory
  - A state machine can not remember anything
  - Only knows the state it is in at the moment
- Regular Language == Finite State Automaton
- **Problem**: It can not deal with nested and recursive calls



How to model nested function calls?

- For every Call F there has to be a Return F
    - Not Regular, this is a Context Free
- If all functions have NOT returned when there are no more events, then we Reject!



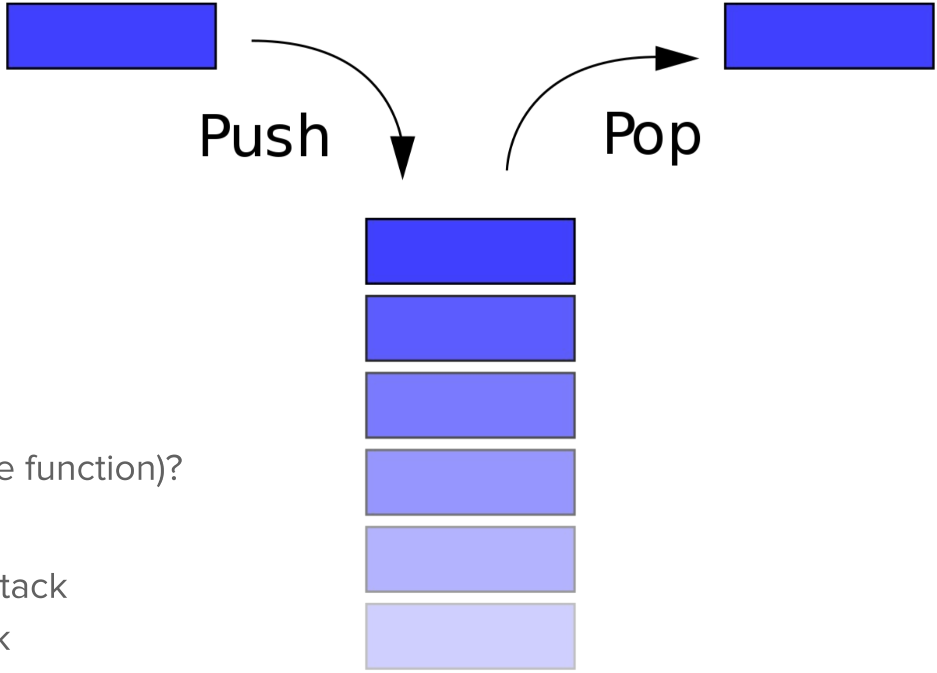
Regular

a^*b^*

Context Free

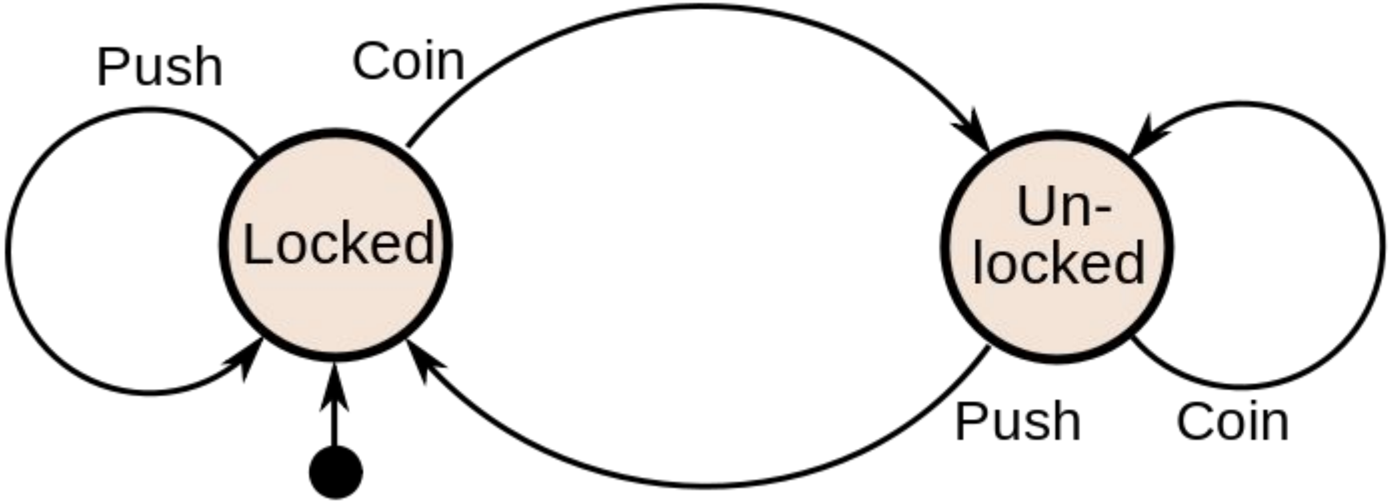
$a^n b^n$

Pushdown Automata

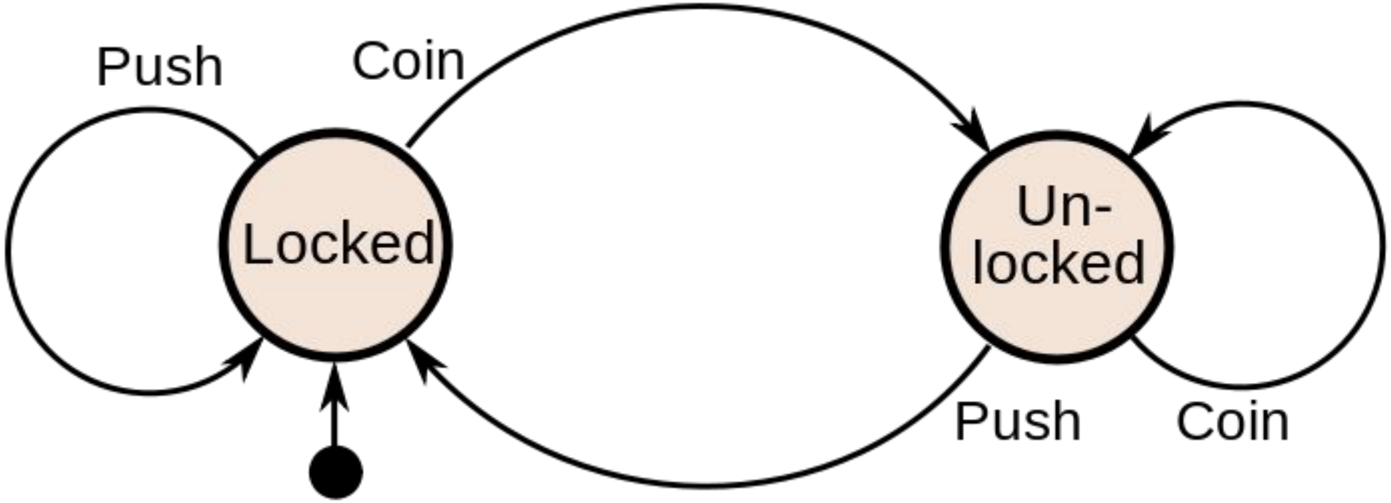


- Finite State Automaton **+ Stack**
  - Now it can remember what have happened
- Can check Context Free Languages
  - For every Call, is there a Return (of the same function)?
- Every time a function is:
  - **Called**: **PUSH** function name onto the stack
  - **Returned**: **POP** function name off the stack
- Now we can check if every call to F has a return of F

Is Push called as many times as Coin?



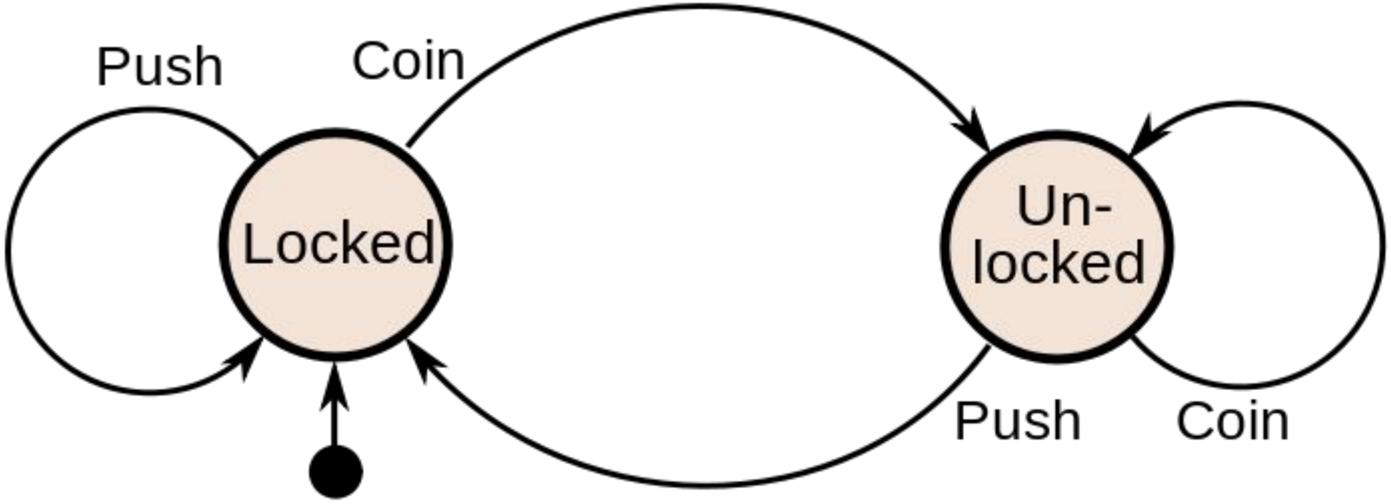
Stack



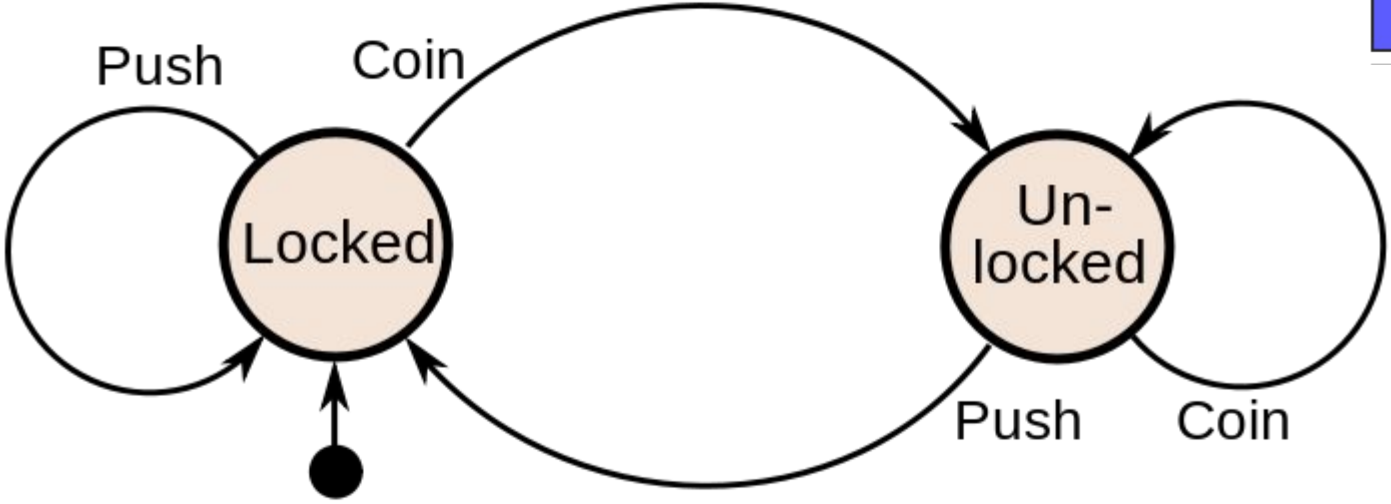
Input: Coin

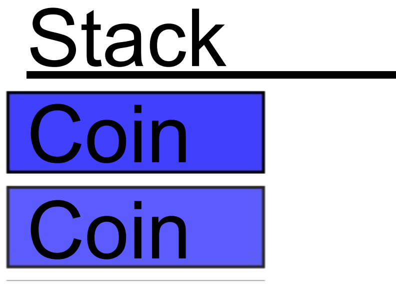
Stack

Coin



Input: Coin

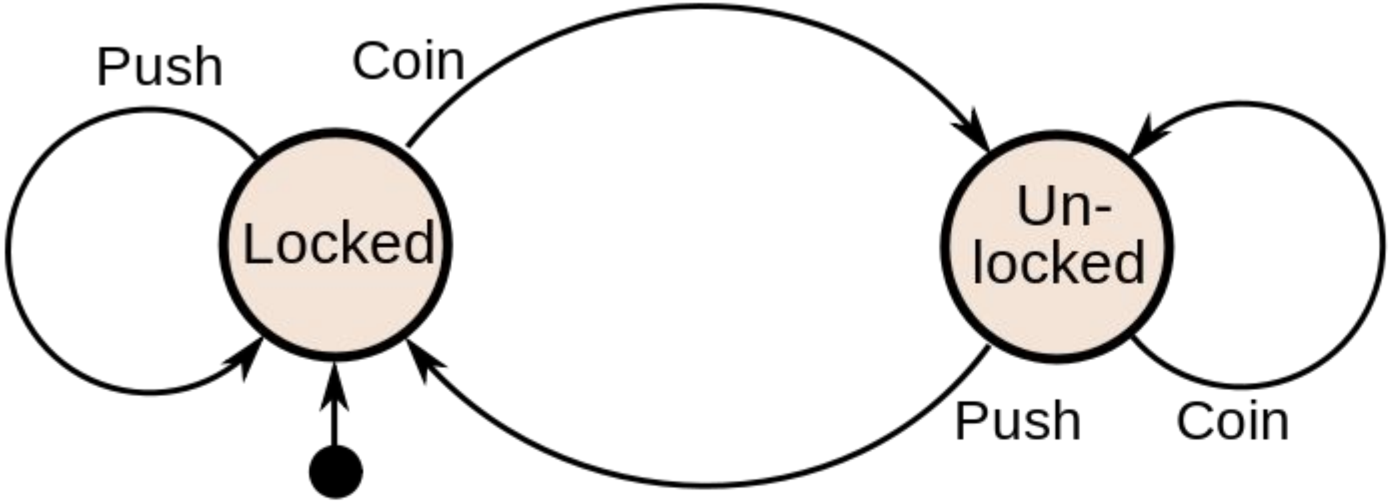




Input: Push

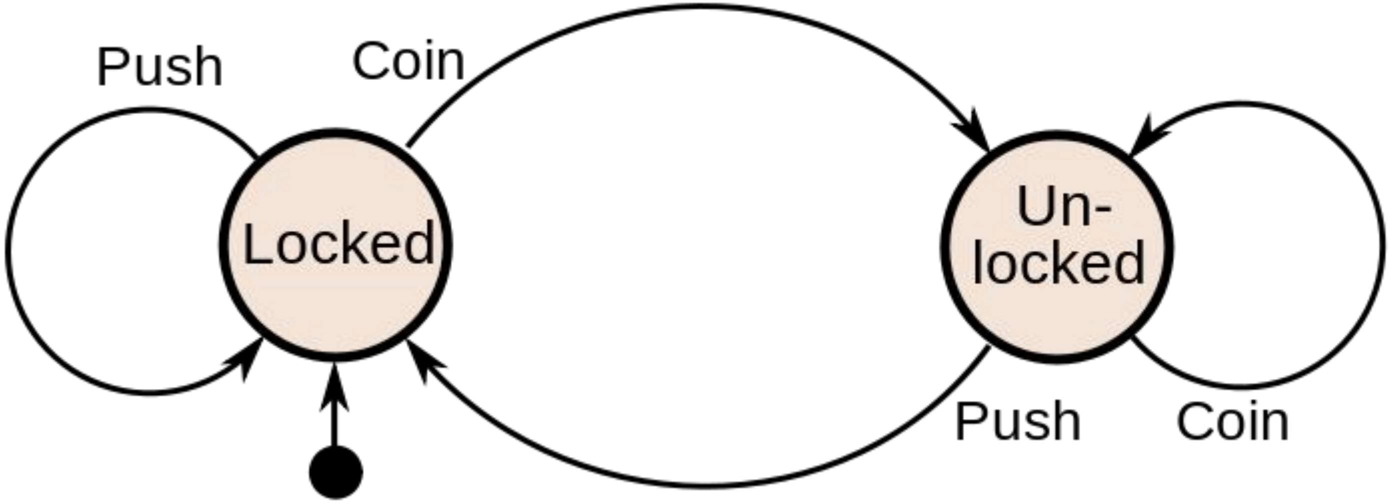
Stack

Coin

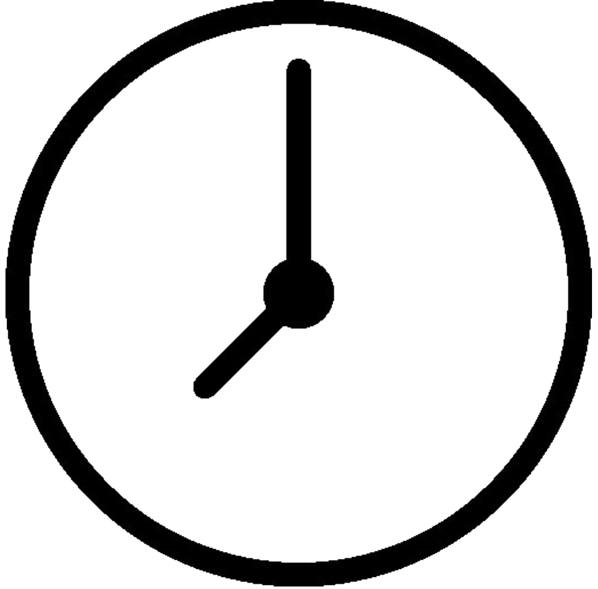


Input: Push

Stack



The Automata has a **Clock**



- An Automaton does not (usually) consider time
- Added a Clock to the Automaton
    - To measure time of a function call or between function calls
- Need a Clock to check the time between two states

In the Implementation:

- Each state can create New Time Constraints
    - A Time constraint is from any state S_x to any state S_y
- Can put a time limit on:
    - Time use by a single Function call
    - TIme between two Functions

What does the Automaton contain?

- States
- Transitions (edges)
- Set of active Time Limits
- Each state has a set of Time Constraints that will be activated once it reaches that state
  - These would be activated when the Automaton enters the state
- Clock
  - Updated for every new Input
  - Checks if each new event is later in time than the previous one
- Stack
  - Remembers which functions have been called and in what order

How does the execution of the Automaton work?

Input is a sequence of tuples: <Function Name, Timestamp, Call / Return>

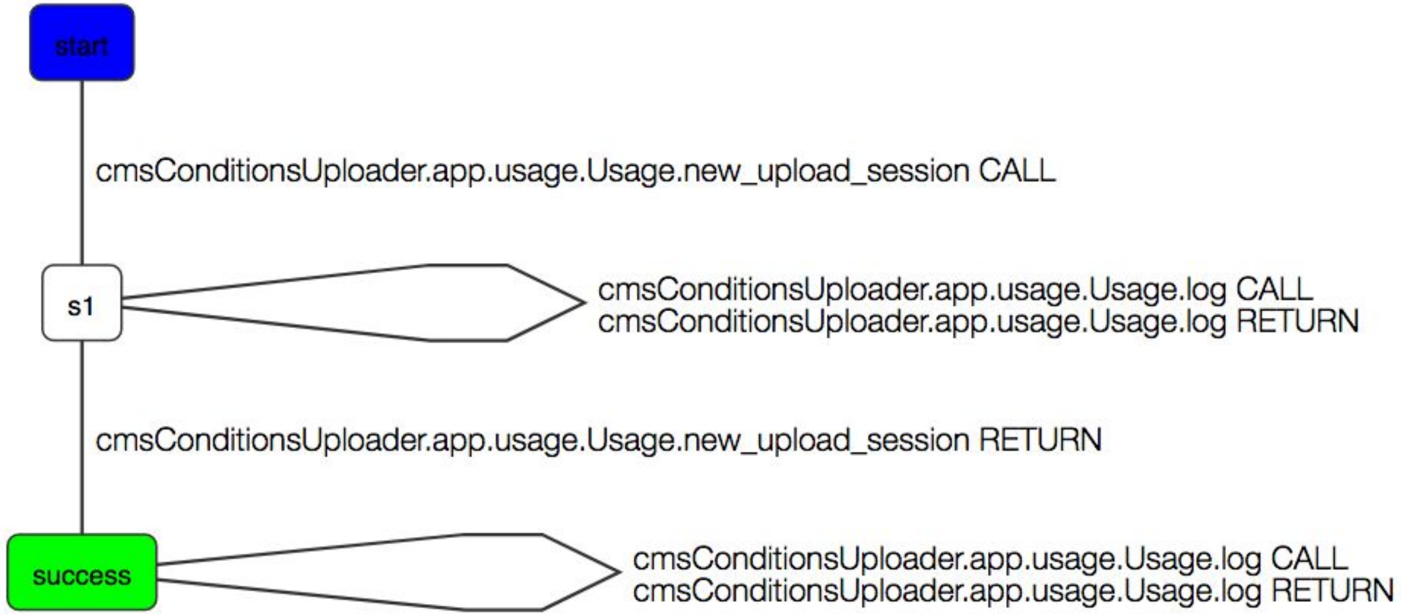
Loop and read from the input until there is nothing left:

1. Check the time limit for reaching this particular state and update the clock
    a. If the clock is past the time limit, Reject
2. Create the new time constraints
3. Read the input and chose the correct transition
    a. If Input event is: Function CALL
        i. PUSH function Name on the stack
    b. If Input event is: Function RETURN
        i. Pop the top of the stack. Compare the input function name to the name on the stack
4. When finished reading the all of the inputs, we Accept if:
    a. The stack is empty
    b. It is in an Accepting state

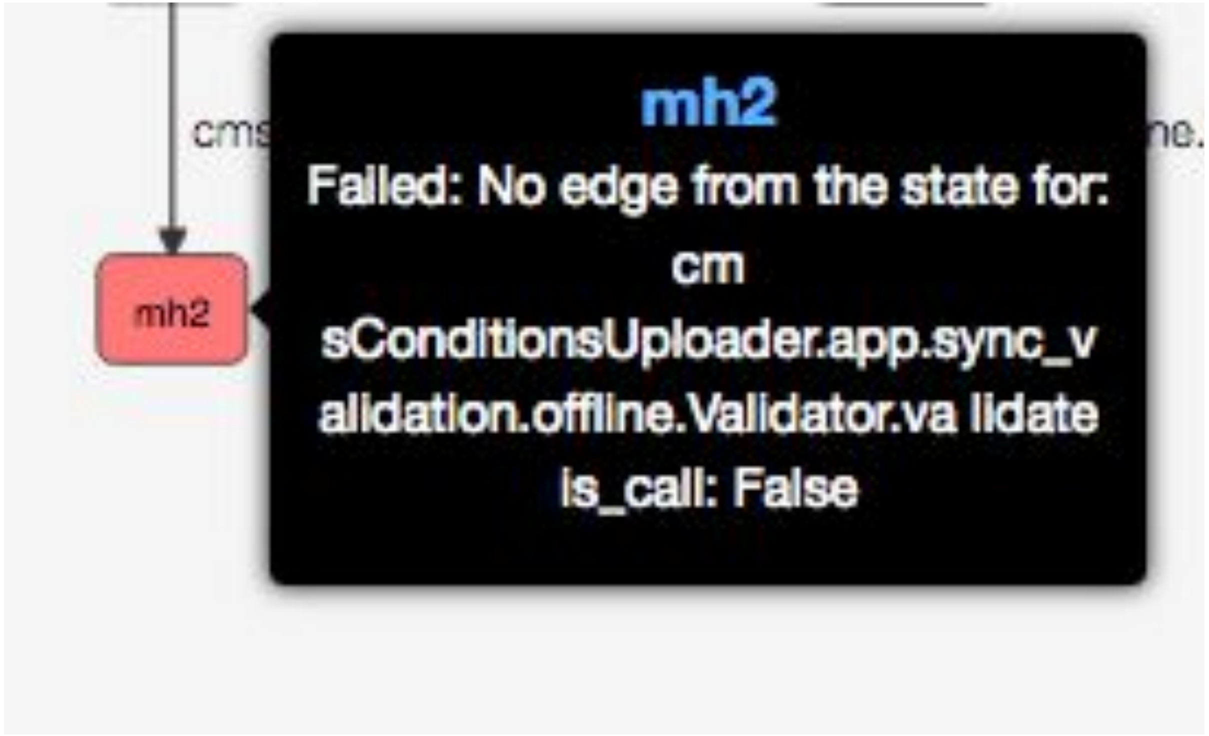
What does it mean to be Accepting or Rejecting?

- Have to finish reading the entire input
- Accept if:
  a. The stack is empty
  b. It is in an Accepting state
- Reject if:
  a. No valid transition for the given input from the current state
  b. The stack is NOT empty
  c. It is in a Rejecting state
     i. All states that are not Accepting

Create your own Automata to use for Runtime Verification


start
cmsConditionsUploader.app.usage.Usage.new_upload_session CALL
s1
cmsConditionsUploader.app.usage.Usage.log CALL
cmsConditionsUploader.app.usage.Usage.log RETURN
cmsConditionsUploader.app.usage.Usage.new_upload_session RETURN
success
cmsConditionsUploader.app.usage.Usage.log CALL
cmsConditionsUploader.app.usage.Usage.log RETURN

- Rejecting



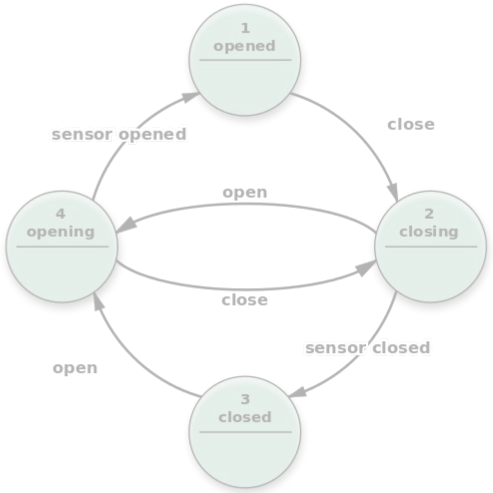
mh2

Failed: No edge from the state for: cm sConditionsUploader.app.sync_v alidation.offline.Validator.va lidate is_call: False

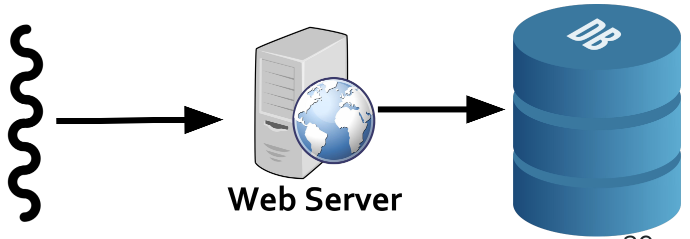
Summary of the system

- Build your own Automata!
- Transitions / Edges are created by Function Calls and Returns
- All function calls/returns must be linearly ordered in time
- Add Time Constraints between any two states
  - Time a function is allowed to use
  - Time between two different function calls
    - new_session() ➔ close_session()
- Run it! Check that your system works as intended
  - If it fails, it will show you in RED the states where it failed and the reason it failed
  - No RED states, it will highlight the SUCCESS STATE it ended at in GREEN

The Runtime Verification Framework

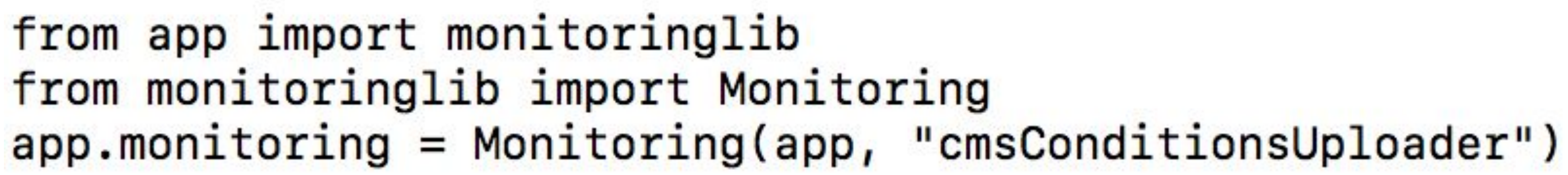


Backend implementation

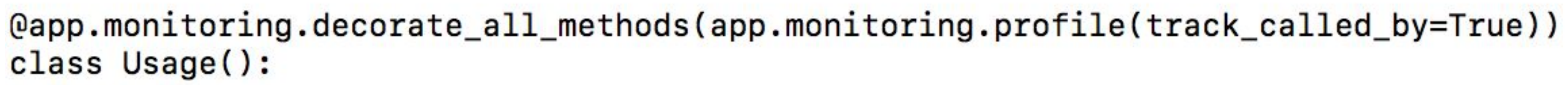


How to apply the monitoring to your system?

1. Import the monitoring library and decorate your classes and functions
   a. Change the server address in the monitoringlib.py file to the server address
      i. I use: http://localhost:<PORTNUMBER>



```
from app import monitoringlib
from monitoringlib import Monitoring
app.monitoring = Monitoring(app, "cmsConditionsUploader")
```



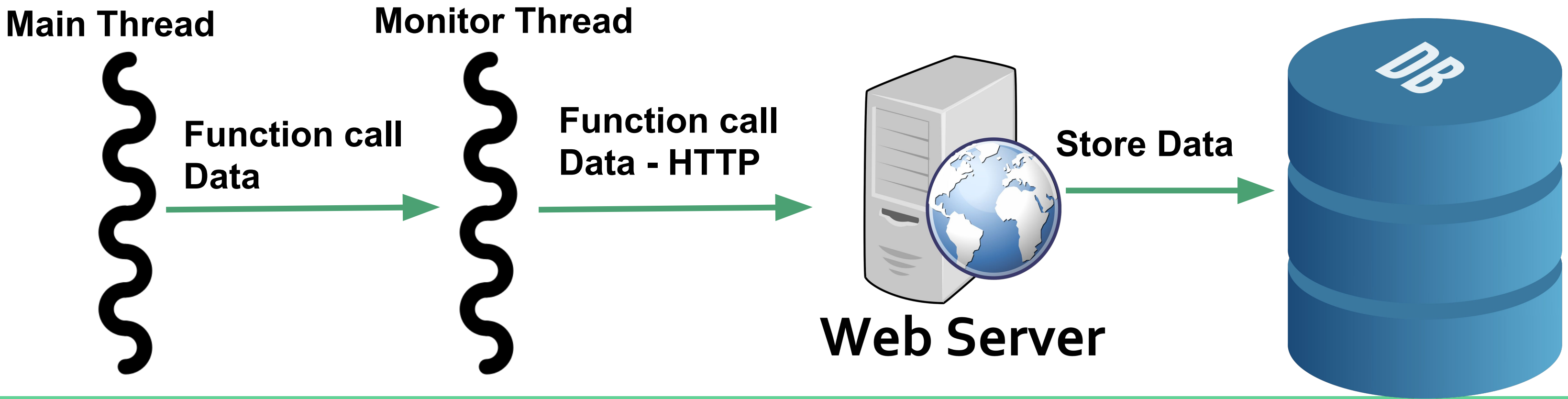
```
@app.monitoring.decorate_all_methods(app.monitoring.profile(track_called_by=True))
class Usage():
```

2. Setup a server to receive the data from the monitored function calls
   a. Download the files from gitlab
   b. Initialize the Database

Instrumentation - Data is sent to a seperate DB process

The collecting and storing of the data works like this:

1. The monitor **creates a new thread** upon initialization
2. The **decorated functions collect data** about the function call and sends this data to the **monitor thread**
3. The **monitor thread** sends this data to a server over HTTP
4. The receiving server stores the data on a SQL Database



How is the data obtained for the Automaton?

1. You create an Automaton
   a. Select **function calls** for the edges
   b. **Time interval** you want to check
2. Filter the data on function names and time interval
   a. Group them by Sessions, a new query for each session
3. The Automaton data pre-processor creates the trace from the fetched data
   a. Has to convert the function call data in the Database into a trace with calls and returns
4. Automaton validates all the traces
   a. One trace for each session
5. Shows on the Graph if the runs are: Success or Reject

Why use it?

- Adds extra test coverage to your system
  - Something between unit-testing and Formal validation
- Verifies that your system is behaving as intended in a production environment
  - It needs more debugging and some refinement, before it is used in production
- Lightweight
  - Little overhead
    - Tested on CERN Open Stack VM - 6% overhead for the Uploader on small uploads
    - On average adds: 15 milliseconds
- Easy to use and setup
  - Import, initialize and you are ready to go
- Use the graph functionality to look deeper into the anomalies and find the cause
  - Maybe your system is slow only at certain times of the day, week, month etc.

Thank you for your time