Decision Making & Planning for Cyber-physical Systems

Raimund Kirner

University of Hertfordshire

acknowledgement: includes slides by Mick Walters

Overview

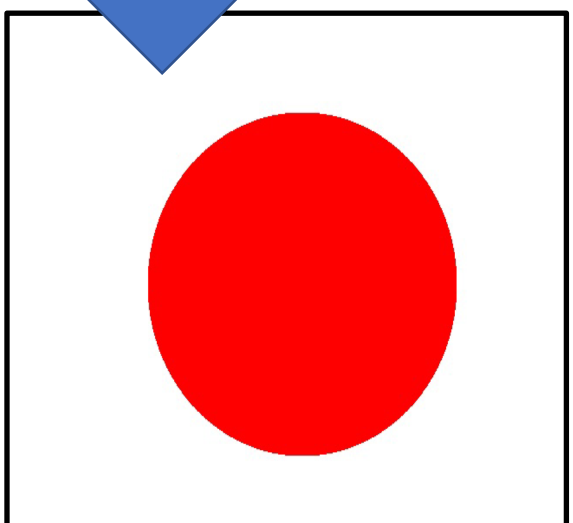
- CW2 Overview
- Parallel Programming
- Finite State Machines
- Nested Finite State Machines

CW2: Extended Autonomous Reliable Car

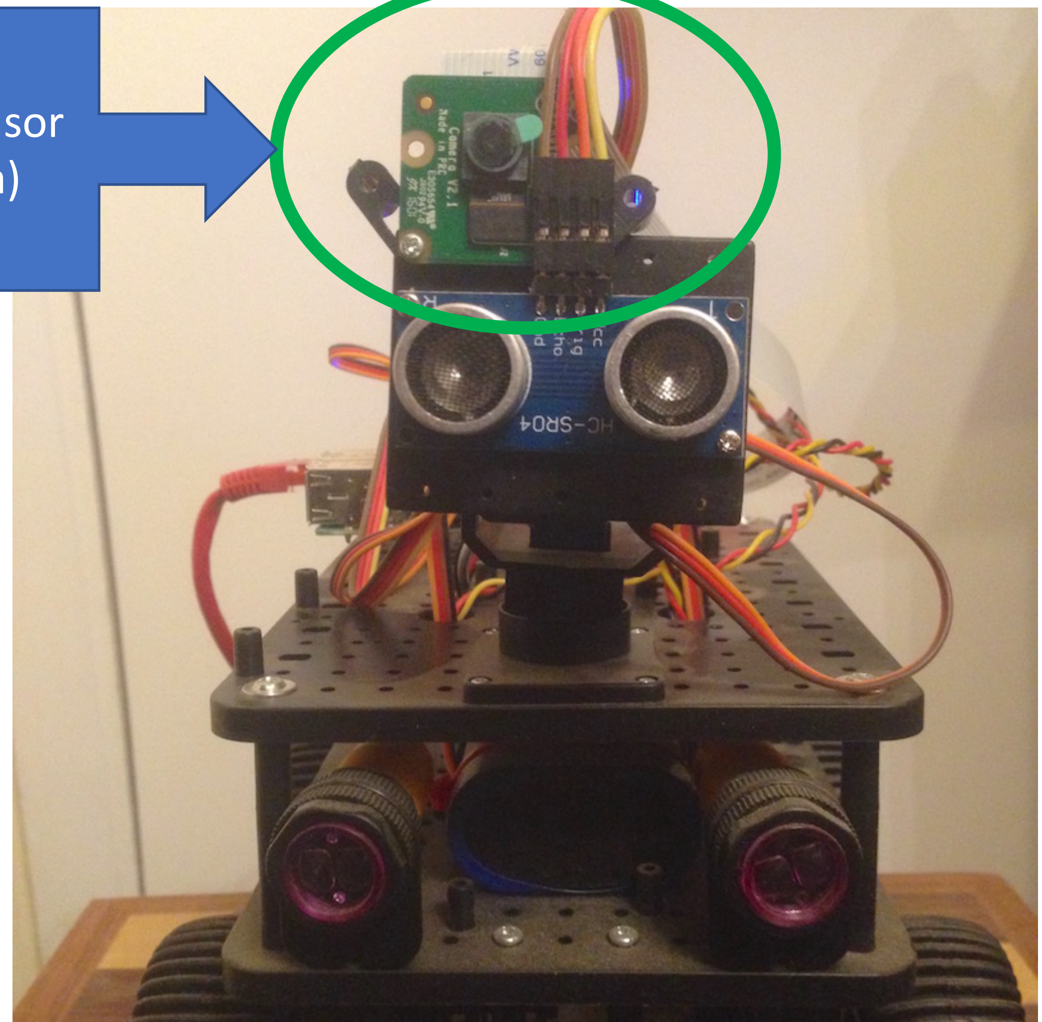
- EARC: Extended autonomous reliable car
- features:
  1. Stop if obstacle ahead (IR sensors)
  2. Search for binary large object (blob) using camera
  3. Align to found blob
  4. Keep distance to found blob (US sensors)

CW2: Extended Autonomous Reliable Car

Object to align to (red blob)



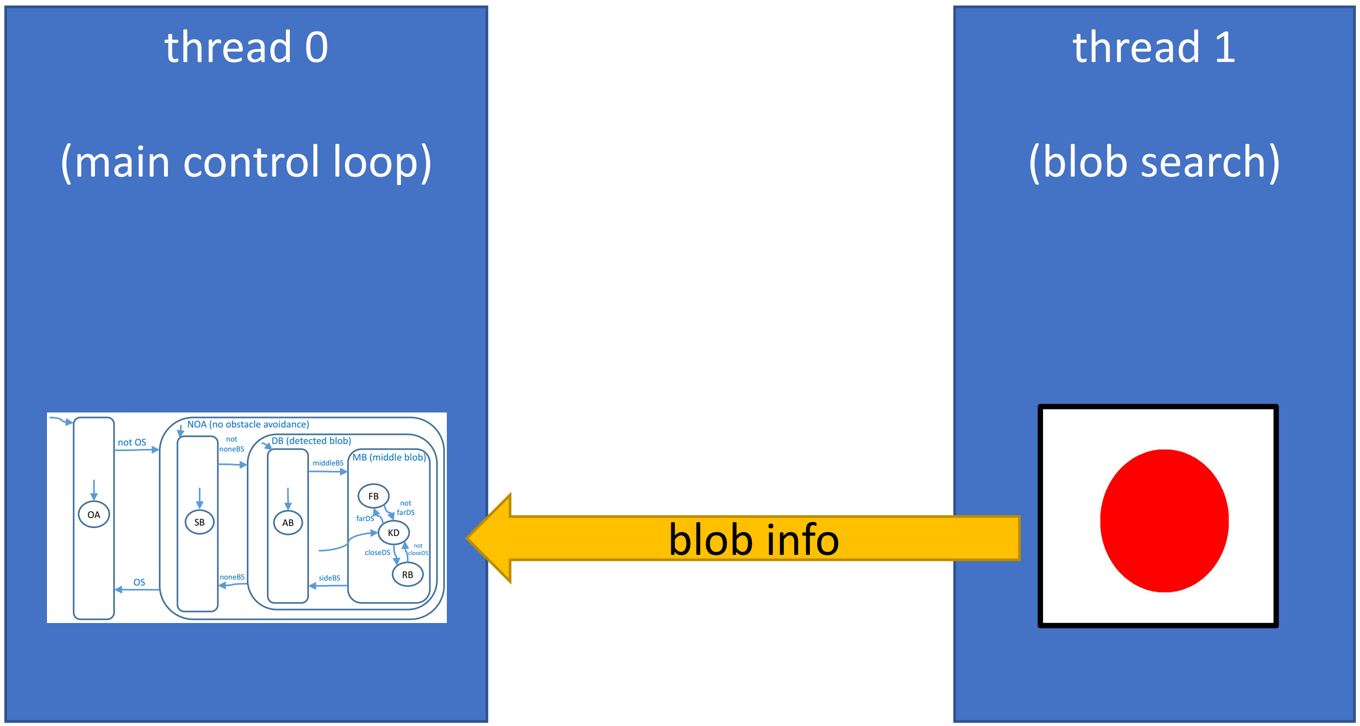
Visual Sensor (camera)



Parallel Programming - CW2 Requirements

- Activity of visual sensing takes relatively long (~ 1 second)
- Visual sensing takes too long to be included within main control loop
- Use of separate blob thread which does
  - visual sensing
  - blob search
- Whenever one blob search done, update the result to main control loop

Parallel Programming - CW2 Requirements



Parallel Programming - Foundations

- Sequential Computing:
  complete one execution before next one starts

- Parallel Computing:
  involves the concurrent or parallel execution

Parallel Programming - Foundations

- Definition: Parallel Computing:

Two or more computations are executed simultaneously

Parallel Programming - Foundations

- Definition: Concurrent Computing:

The interval between start and stop of two or more computations overlaps

Parallel Programming - Example of Parallelism

- Task: bees need to kill visiting scout of Japanese Giant Hornet before it leaves and returns with reinforcement to kill the whole bee hive.
- Algorithm: Using the fact that bees can withstand higher temperatures than hornets, the bees form a ball around the hornet and vibrate in order to produce a temperature increase inside the ball that kills the hornet.
- This only works if the bees work in parallel, i.e., simultaneously (working concurrently is not sufficient).

Parallel Programming - Example of Concurrency without Parallelism

- John works in a customer service, where he occasionally has to answer the phone. In the pauses between two calls he reads a nice book.
- The work in the customer service and the book reading are two concurrent processes with overlapping start-end intervals.
- However, both processes cannot be executed at the same time (no reading while talking to a customer, so no parallelism)

Parallel Programming - Foundations

- Difference between processes and threads:
- processes:
  - have their own address space
  - communication only via inter-process communication mechanisms
- threads:
  - all threads of same process share the address space
  - communication directly via objects in shared memory
  - synchronisation needed to ensure consistent communication

Parallel Programming - Creating concurrent programs with pthread.h

```
#include <pthread.h>
#include <assert.h>
void *worker(void *p_thread_dat);

int main (int argc, char **argv) {
  int balance = 1000;
  pthread_t rt_thread;   // thread management data
  pthread_attr_t pt_attr;   // thread attributes
  assert (pthread_create(&(rt_thread), &pt_attr, worker, &balance)==0 );
  // do something concurrently to second thread:
  balance = balance – 300;
  // wait for thread to finish
  assert ( pthread_join(rt_thread, NULL) == 0 );
  pthread_attr_destroy(&pt_attr);  // destroy thread attribute
  return EXIT_SUCCESS;
}
```

Parallel Programming -
Creating concurrent programs with pthread.h

```
void *worker(void *p_thread_dat) {
  int *balance = (int *) p_thread_dat;
  // do some concurrent update of balance:
  *balance = *balance + 100;
  return NULL;
}
```

Parallel Programming - Creating concurrent programs with pthread.h

```
void *worker(void *p_thread_dat) {
  int *balance = (int *) p_thread_dat;
  // do some concurrent update of balance:
  *balance = *balance + 100;
  return NULL;
}
```

extracting parameter inside thread function

Parallel Programming - Race Conditions

- A race condition is a phenomenon where the computed result of two or more concurrent programs depends on the timing of the individual programs
- The execution time of the programs or scheduling decisions of the operating system, for example, can influence the execution time.
- Due to race conditions the final result can become non-deterministic.

Parallel Programming - Race Conditions

```
balance = 1000;
void book_in  (int amount) { balance = balance + amount; }
void book_out (int amount) { balance = balance - amount; }
```

Thread 0:

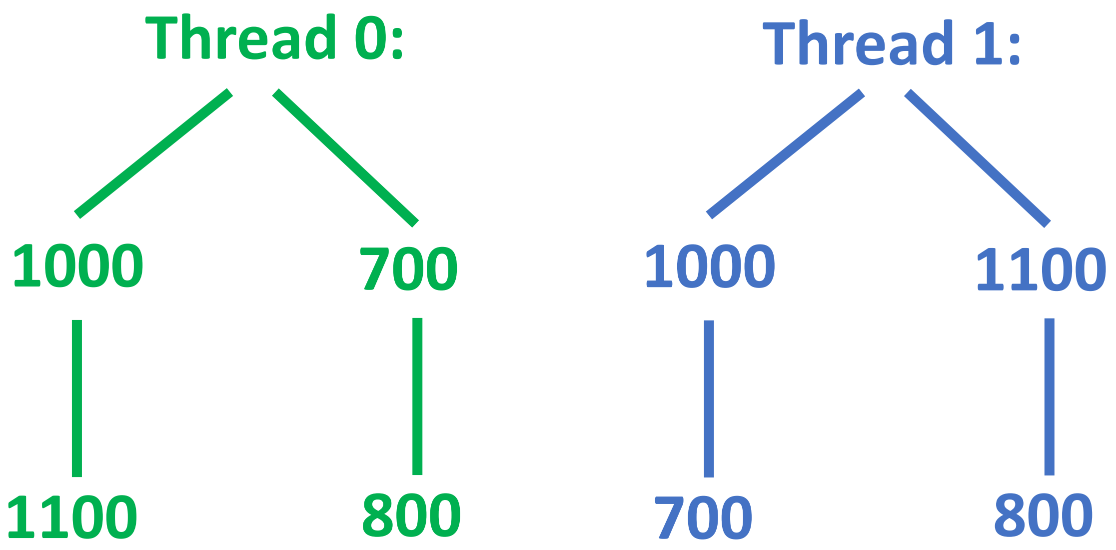
```
book_in(100);
```

Thread 1:

```
book_out(300);
```

Q: what will be the final value of balance?

Parallel Programming - Race Conditions



Parallel Programming - Race Conditions

- The basic problem of race conditions in the example is non-atomic access of shared data.
- The program parts where concurrent access to shared data happens is called "critical section"

- To fix this, we have to make sure that "critical section" is accessed by each program in an atomic way (no in-between access of the shared data by any other program)

Parallel Programming - Semaphore

- One way to make access to "critical sections" atomic, is the use of semaphores
- A semaphore S is a variable that represents the access state, being used via two functions:
  - **wait(S):** "allocate resource": if S>0 then decrement S and program continues, if S=0 then thread blocks and is linked to the waiting list of S.
  - **signal(S)**: "deallocate resource": if S has waiting threads, then awake first blocked thread to continue, else increments value of S.

Parallel Programming - Race Condition Eliminated

semaphore
pseudo instructions

Extending the code with pseudoinstructions (wait/signal):

```
balance = 1000;
semaphore S=1;
void book_in    (int amount) { wait(S); balance = balance + amount; signal(S); }
void book_out (int amount) { wait(S); balance = balance - amount; signal(S); }
```

Thread 0:

book_in(100);

Thread 1:

book_out(300);

balance can only be 800

Parallel Programming - Implementing semaphores with pthread.h

```
#include <pthread.h>
int balance;
pthread_mutex_t count_mutex;

void book_in    (int amount) {
  pthread_mutex_lock(&count_mutex);
  balance = balance + amount;
  pthread_mutex_unlock(&count_mutex);
}
```

```
void book_out    (int amount) {
  pthread_mutex_lock(&count_mutex);
  balance = balance - amount;
  pthread_mutex_unlock(&count_mutex);
}
```

Thread 0:

book_in(100);

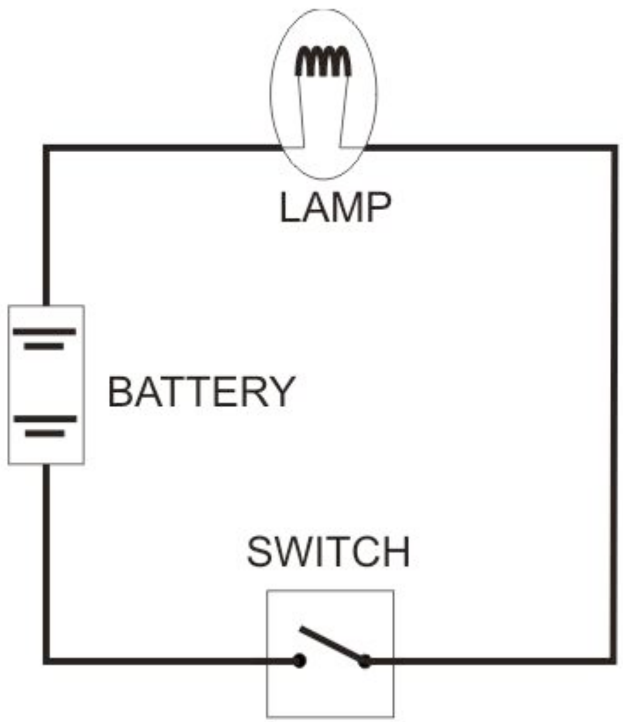
Thread 1:

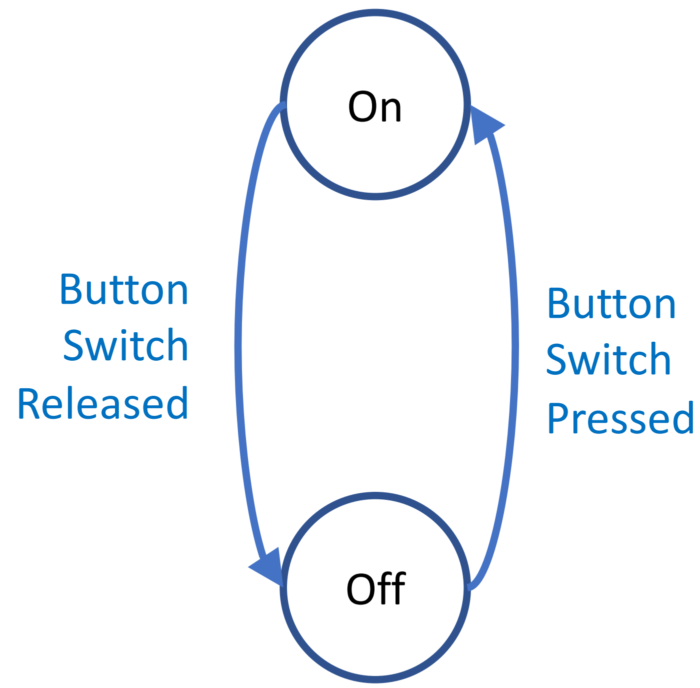
book_out(300);

Finite State Machines (FSM)

- State means that the machine has some memory
- When we have state, responses can be influenced by past sensory readings as well as current sensory readings.
- Theoretical models might have an infinite number of states
- A finite state machine (FSM) is a system with a finite number of states and rules of how to transition from one state to another state.

Example: Finite State Machines
Light Switch





Example: Finite State Machines
Garage Door

Scenario:

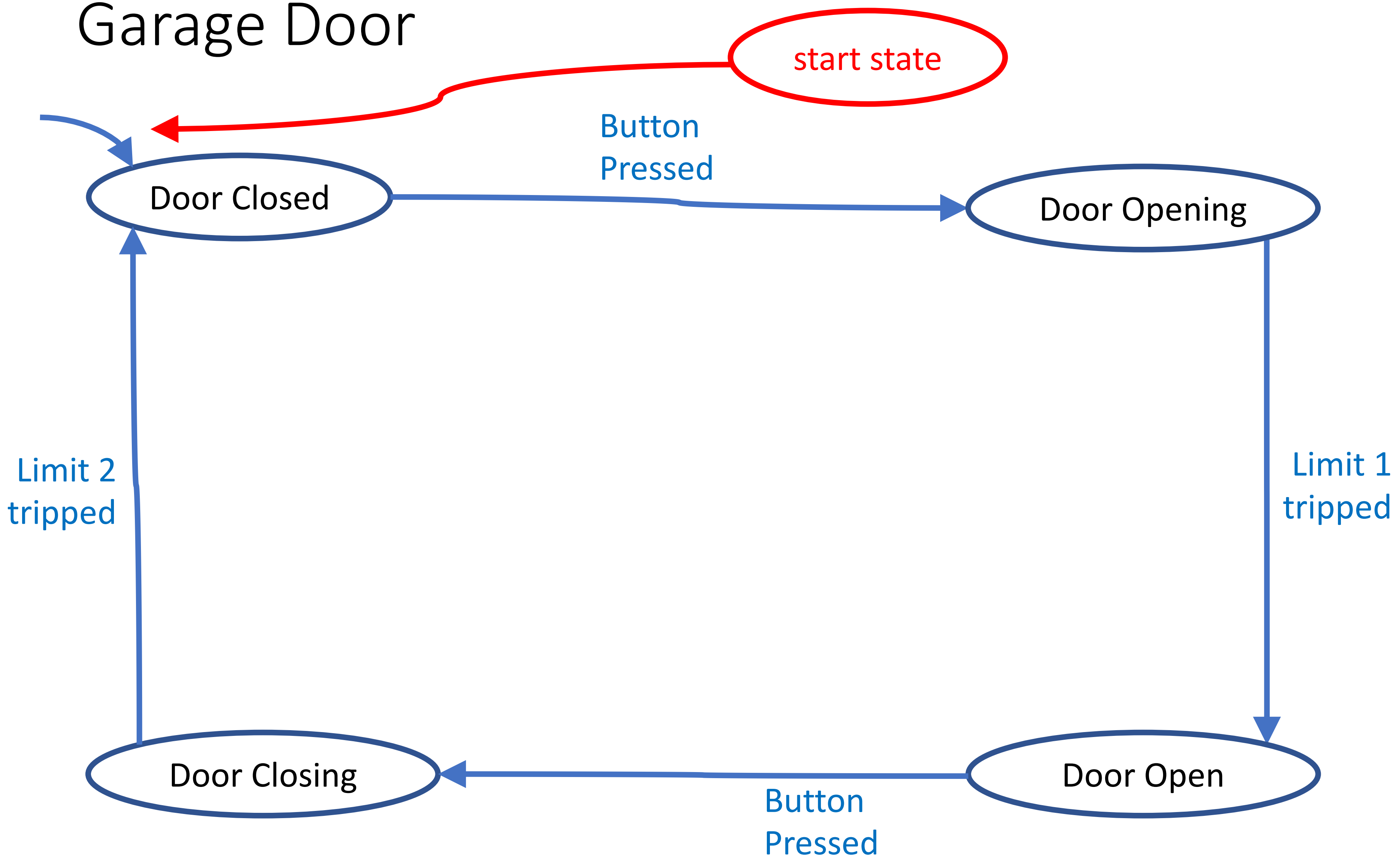
There is one door
There is one button
There are two limit-switches on the door mechanism

Rules:

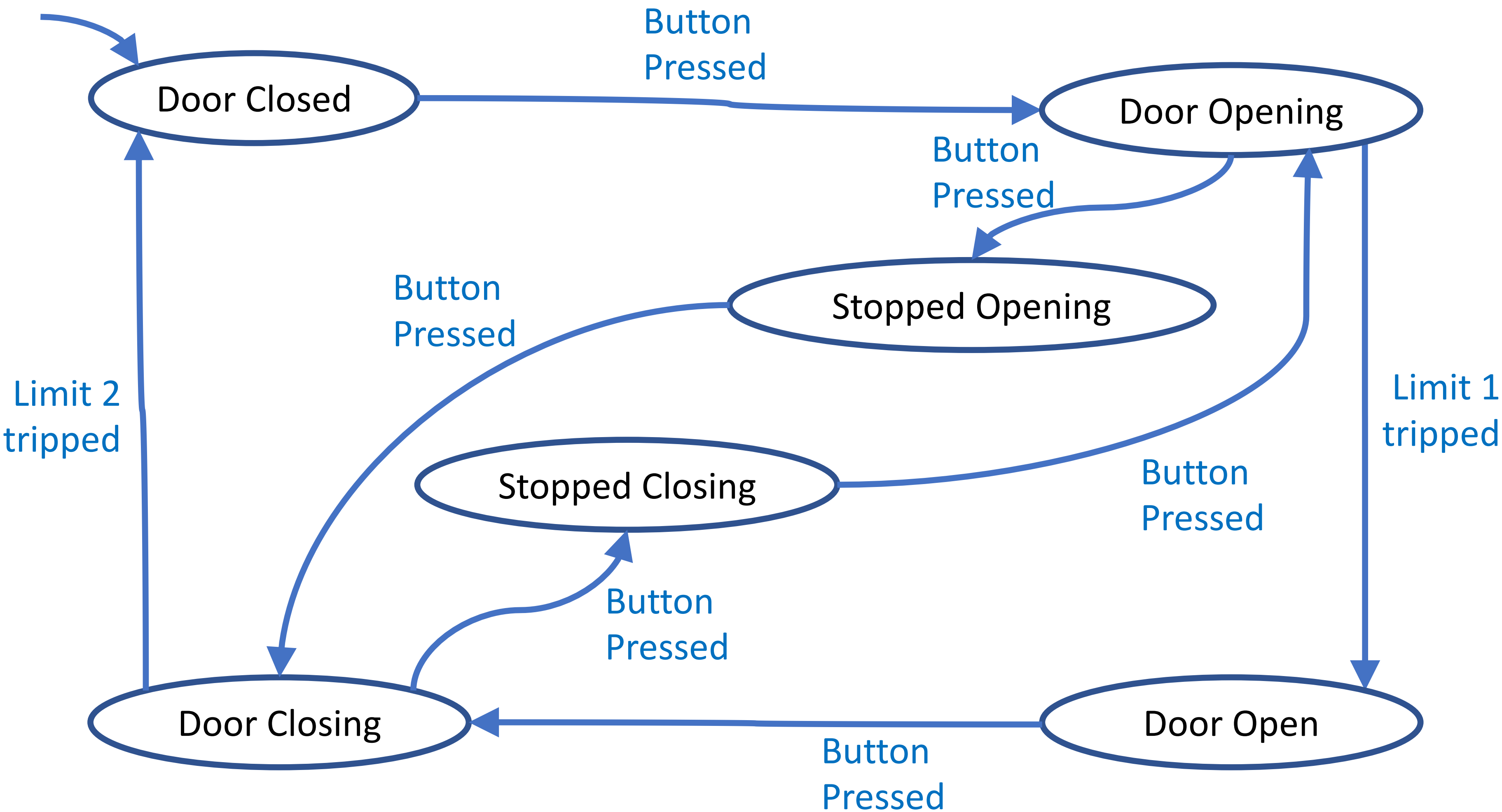
Pressing button opens a closed door
Pressing button closes an opened door

Door stops opening when limit-switch1 is triggered
Door stops closing when limit-switch2 is triggered

Example: Finite State Machines
Garage Door



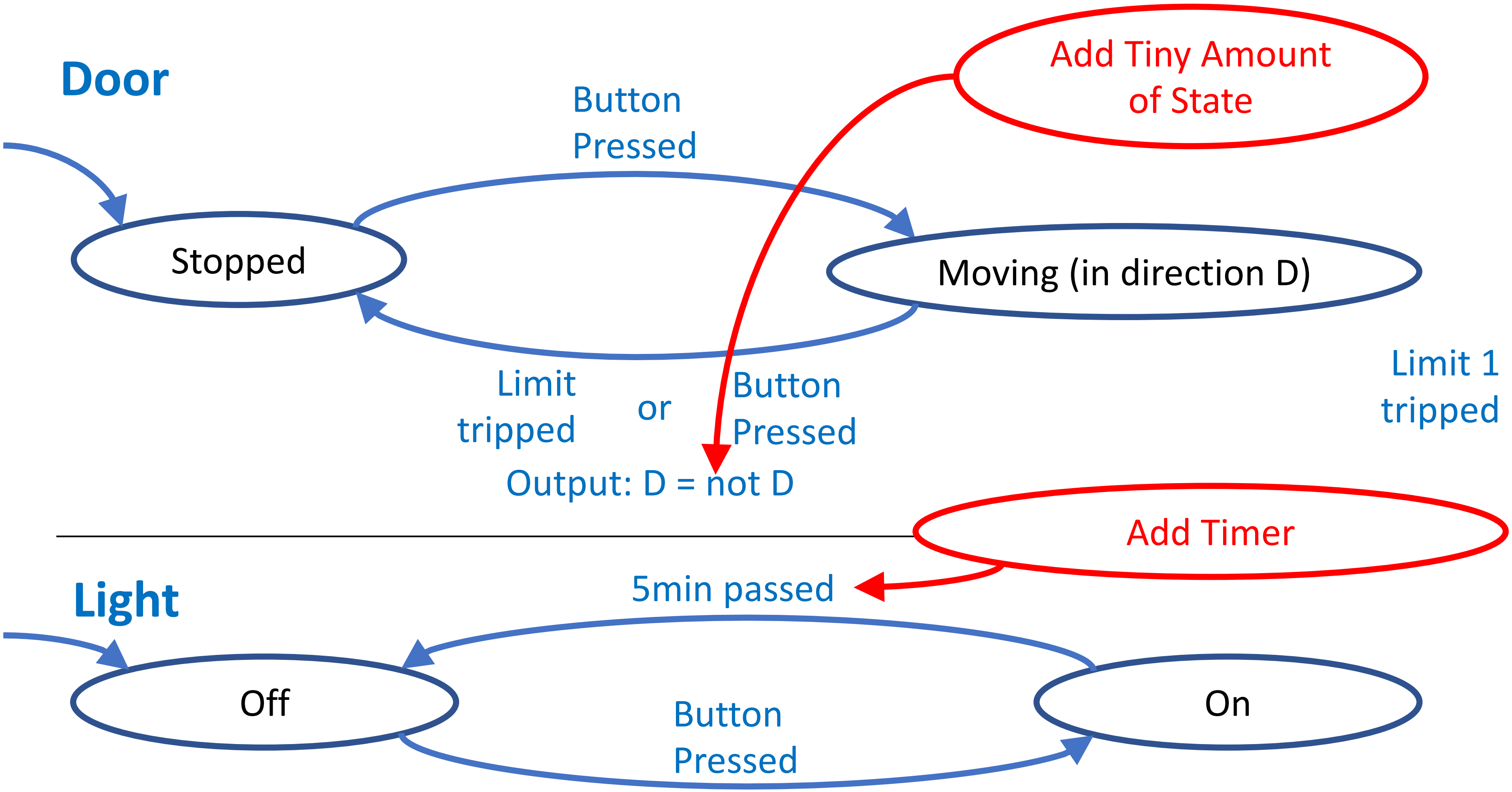
Example: Finite State Machines
Garage Door



And now add a light

Example: Finite State Machines
Garage Door

An Augmented FSM (AFSM)

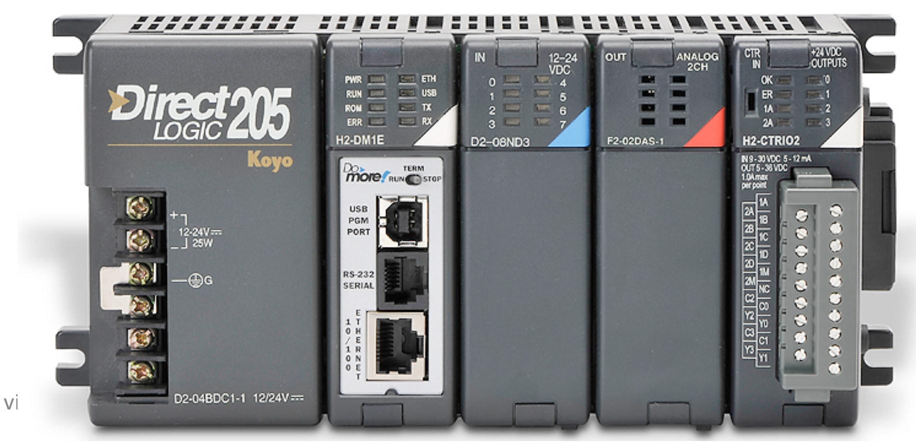


FSM Categorisation

- Finite State Analysis
  ... what we just did
- Finite State Acceptor Diagram
  ... visualisation of FSM
- Finite State Machine (FSM)
  = Finite State Automata (FSA)
- Augmented Finite State Machines (AFSM)
  ... FSM with extra features such as timers, memory, etc.

FSM Implementation

- FSMS can be implemented using general purpose programming languages,
  for example: C, C++, Python, or Java
- However, in industrial sequential control applications, specialised components like Programmable Logic Controllers (PLCS) are commonly used.



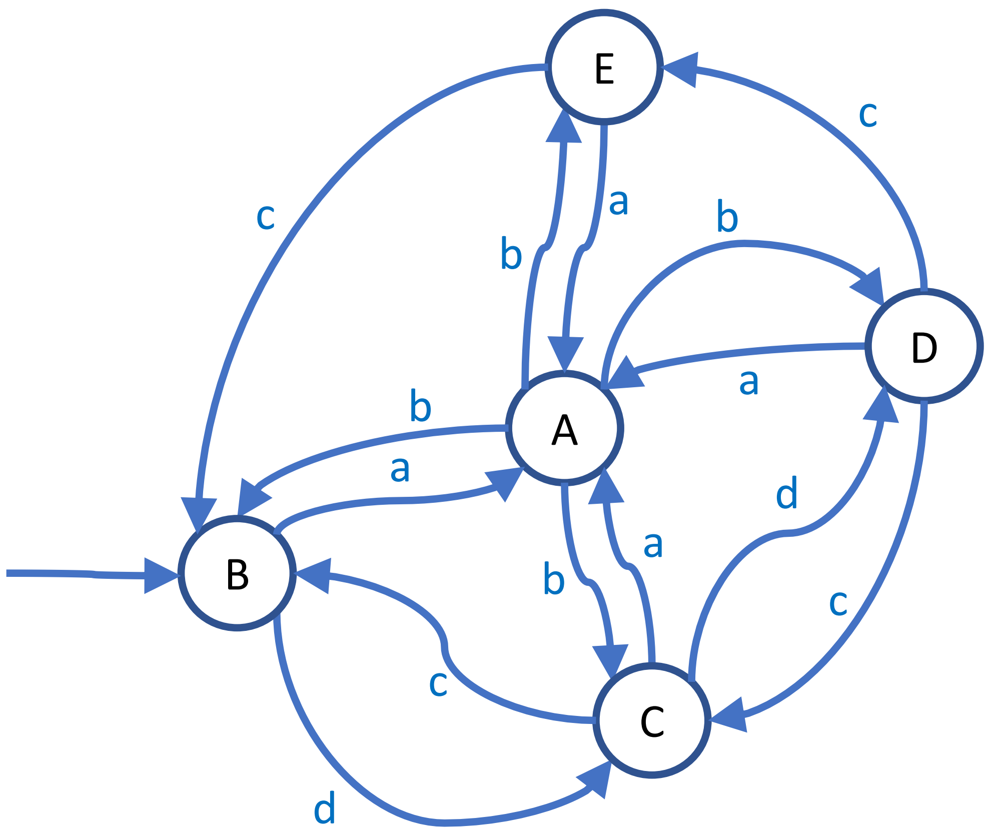
FSM Implementation

```
state = initial-state;
forever {
  input = Read-Sensors();
  state = Update-State( state, input );
  output = Set-Output (state);
}
```

Nested Finite State Machines

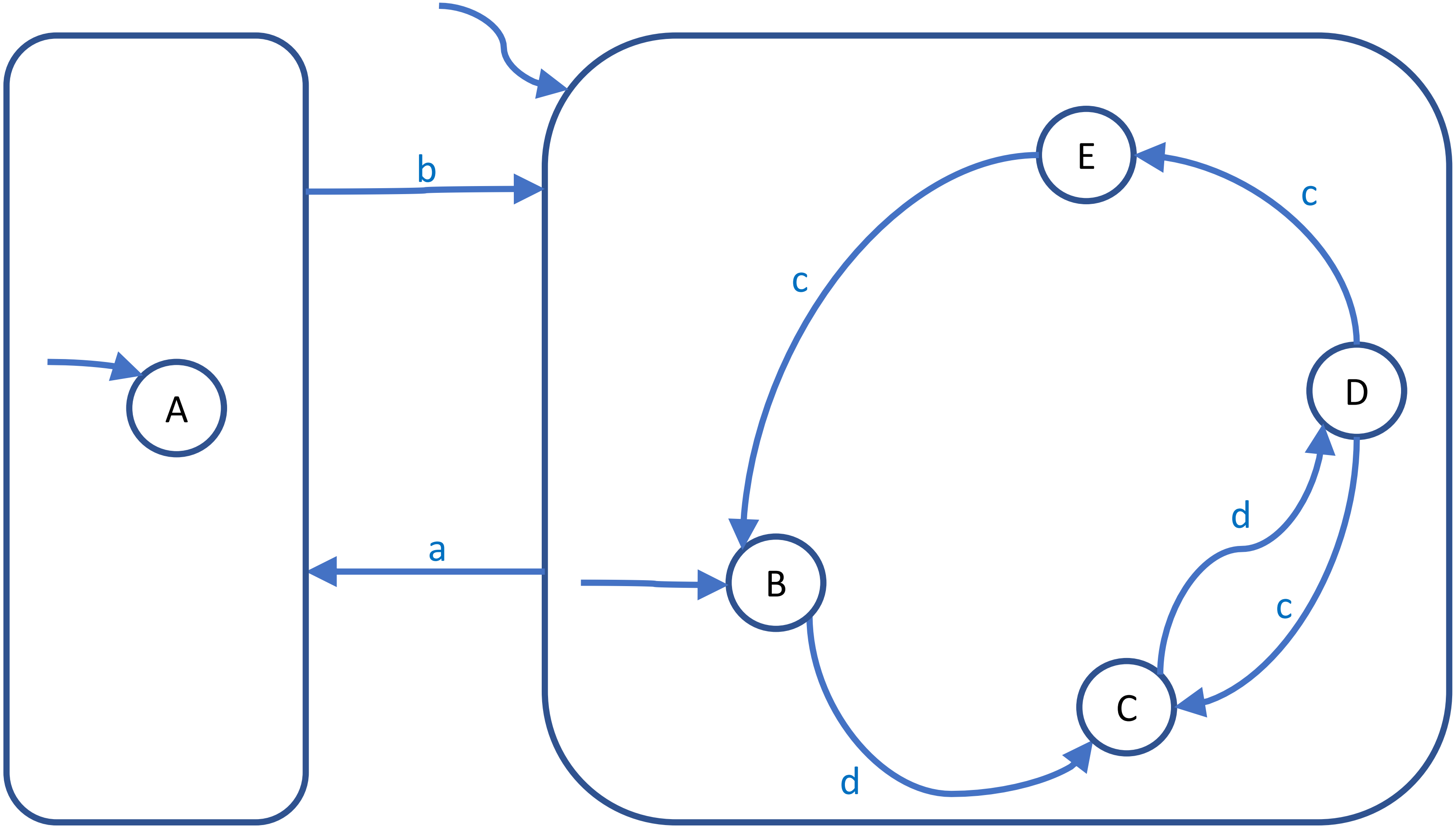
- Problem with FSM: complexity of transition graph tends to grow rather fast
  → impractical to model larger systems
- At the same time, FSM make it hard to express priorities in case that multiple transitions are possible
- Solution: Nested Finite State Machines
  - hierarchical transition graph
  - states of outer level FSM can contain complete FSMs

Nested Finite State Machines



- State A is assumed to have priority over the other states (triggered via input a)
- Thus all other states need to have a direct transition to state A

Nested Finite State Machines



Nested FSM Implementation

```
stateP = initial-state-P; // parent state
stateC = initial-state-C; // child state

forever {
  inputP = Read-Sensors-P();
  stateP = Update-State( stateP, inputP );
  inputC = Read-Sensors-C( stateP );
  input = inputP + inputC
  stateC = Update-State-C( stateC, input );
  output = Set-Output (stateC);
}
```

Formal Notation of FSM

- A finite state machine M is described by the following tuple:

$$M = \{S, L, s, d, F, OF\}$$

- S … set of states
- L … set of inputs
- s … initial state (unique)
- d: S x L → S … state transition function
- F … set of final states (F is subset of S)
- OF … output function

Formal Notation of FSM

- OF … output function
- There are two definitions of OF commonly in use:
  - OF: S → O ... Moore machine
  - OF: S x L → O ... Mealy machine
- In a Moore machine, the current state alone determines the output
- In a Mealy machine, the current state and the current input together determine the output
- The functional expressiveness of Mealy and Moore machine is the same. However, a Mealy machine typically uses less states for the same model than the Moore machine.

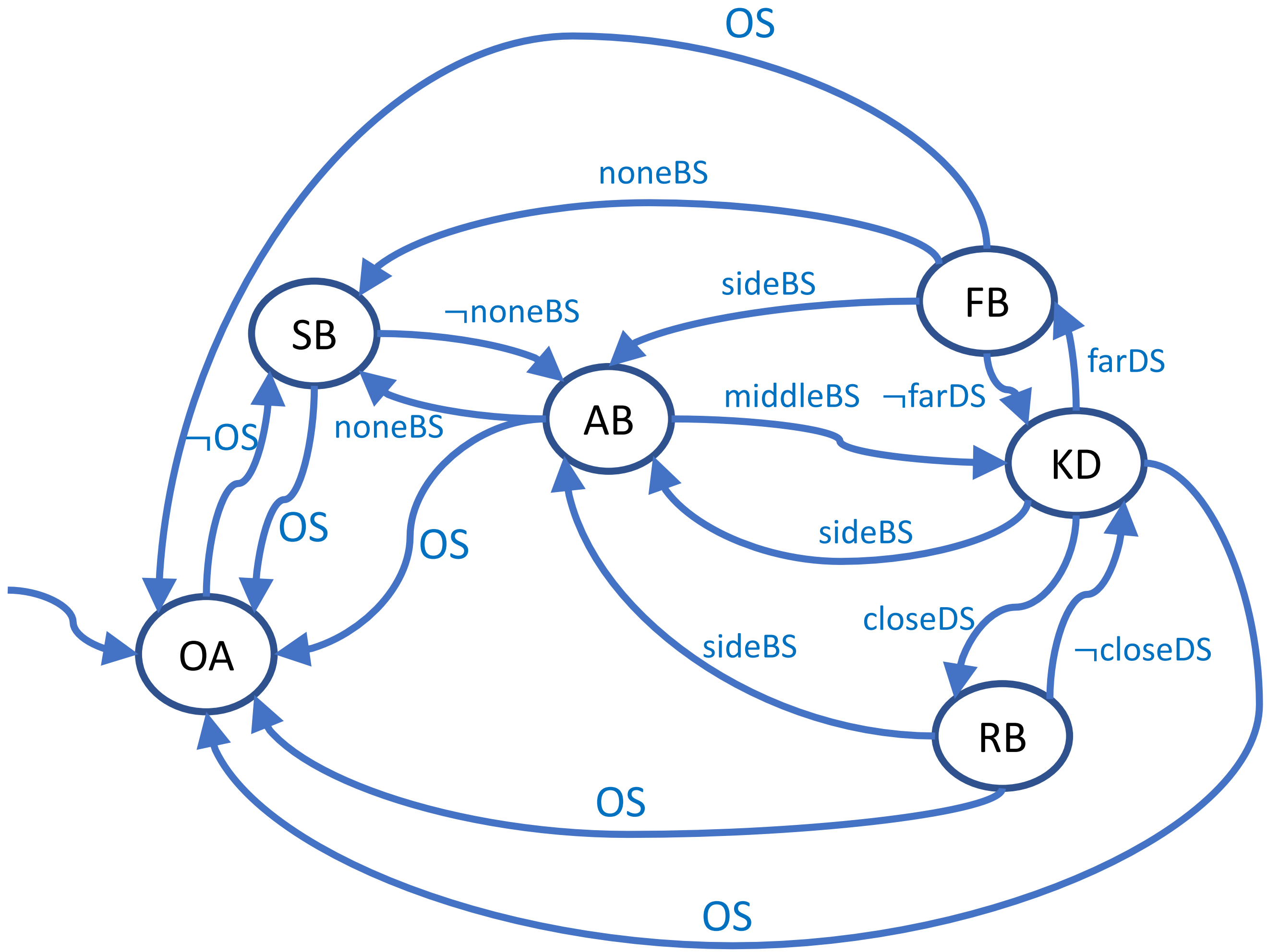
CW2: Nested Finite State Machines - Autonomous Car

- Set of states S:
  - OA … obstacle avoidance (stop car)
  - SB … search blob (spin car)
  - AB … adjust blob (spin car to center blob)
  - KD … keep distance (stop car)
  - FB … forward blob (drive forward)
  - RB … reverse blob (drive backward)

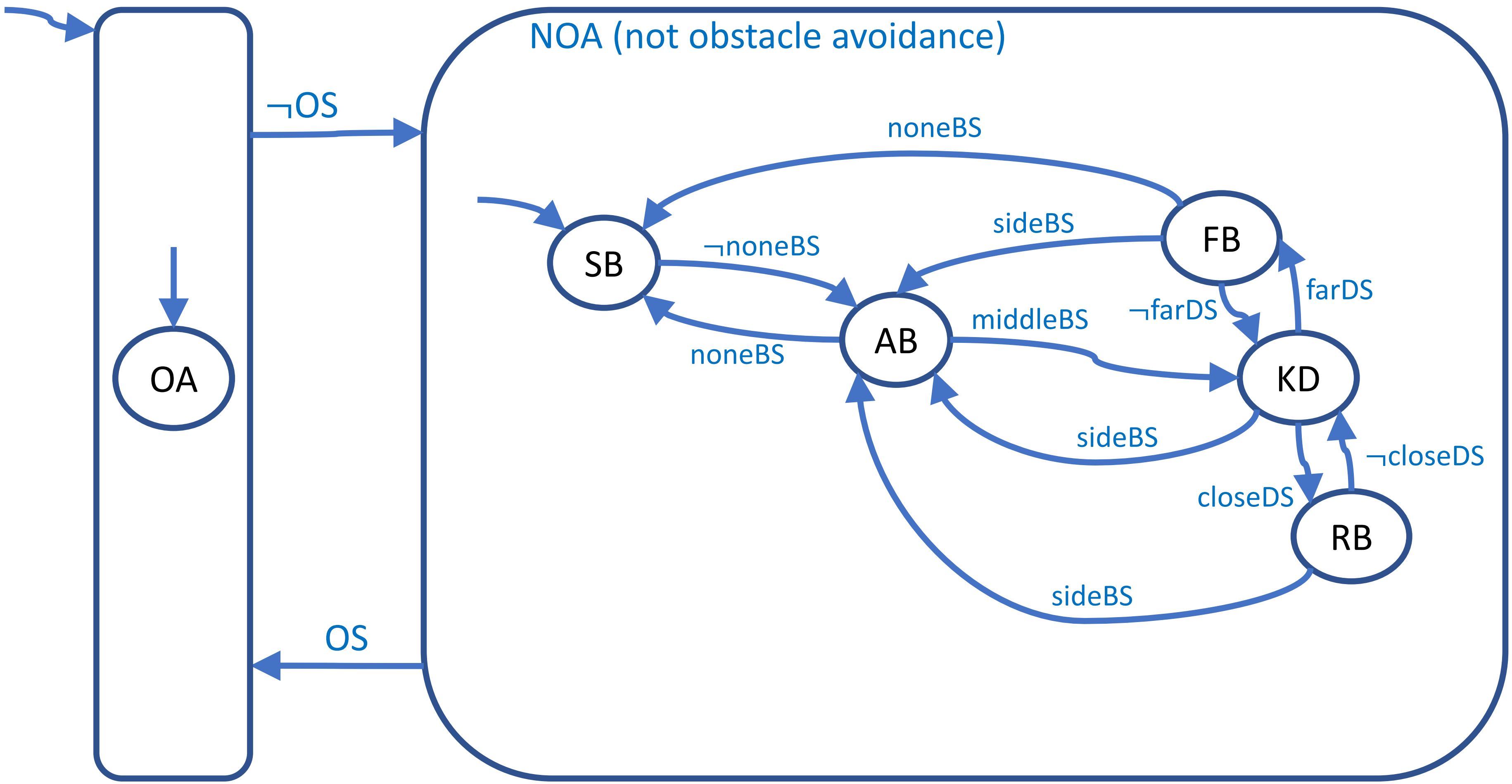
CW2: Nested Finite State Machines - Autonomous Car

- Set of inputs L:
  - Obstacle sensor OS:
    - os … obstacle detected
    - not os … no obstacle detected
  - Blob sensor BS
    - noneBS … no blob detected
    - sideBS … blob detected sideways
    - middleBS … blob detected in middle
  - Distance sensor DS:
    - farDS … far distance
    - closeDS … close distance
    - okDS … acceptable distance

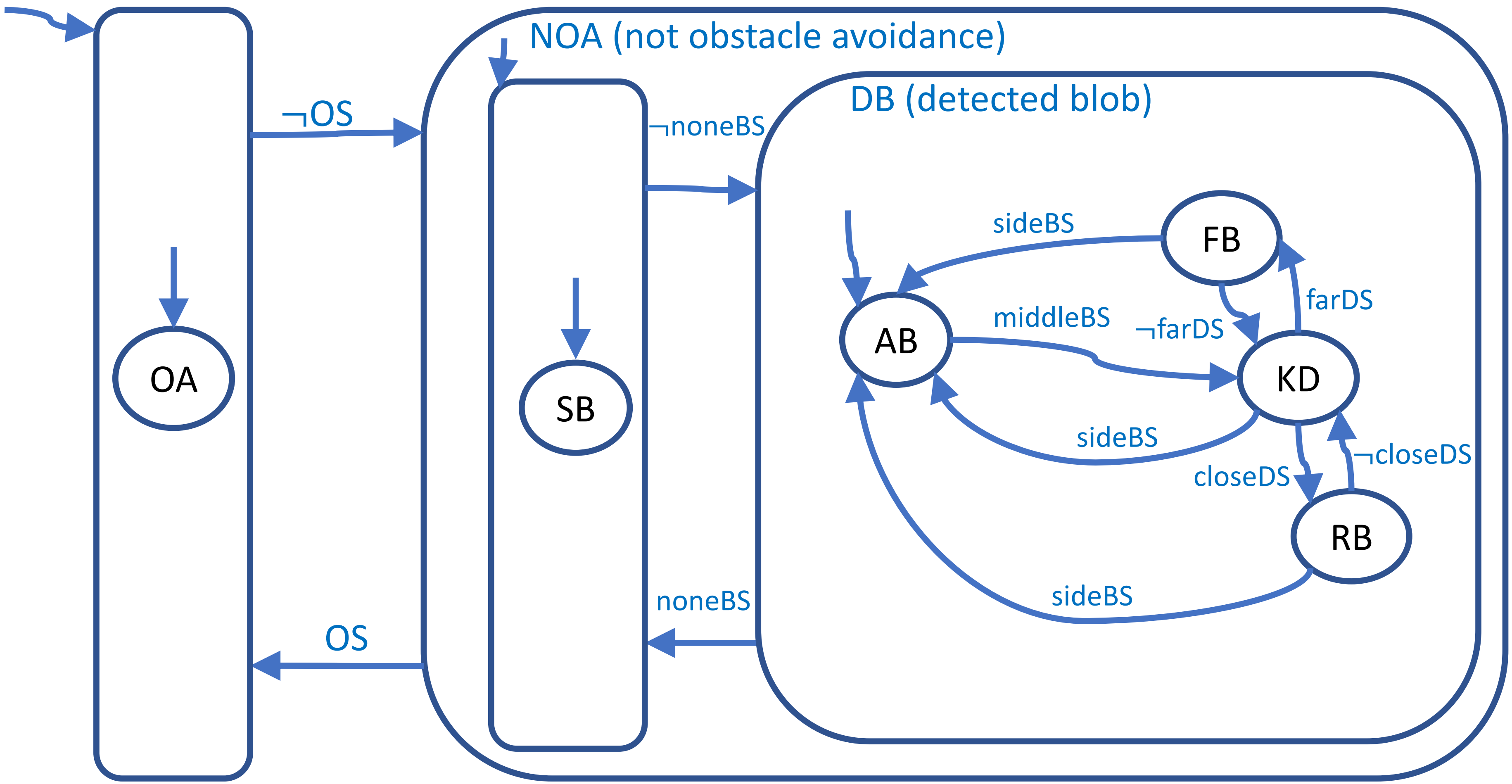
CW2: Nested Finite State Machines - Autonomous Car



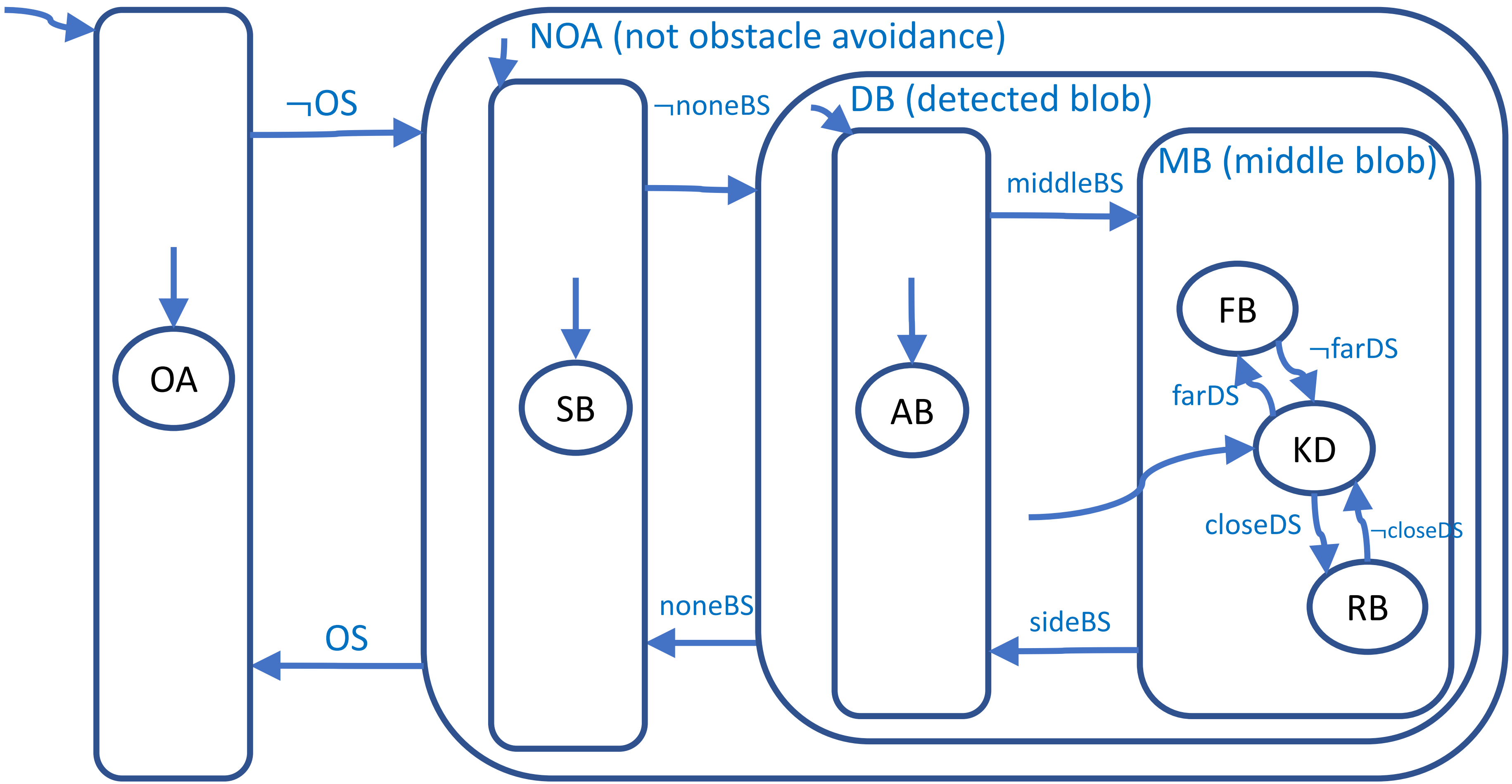
CW2: Nested Finite State Machines - Autonomous Car



CW2: Nested Finite State Machines - Autonomous Car



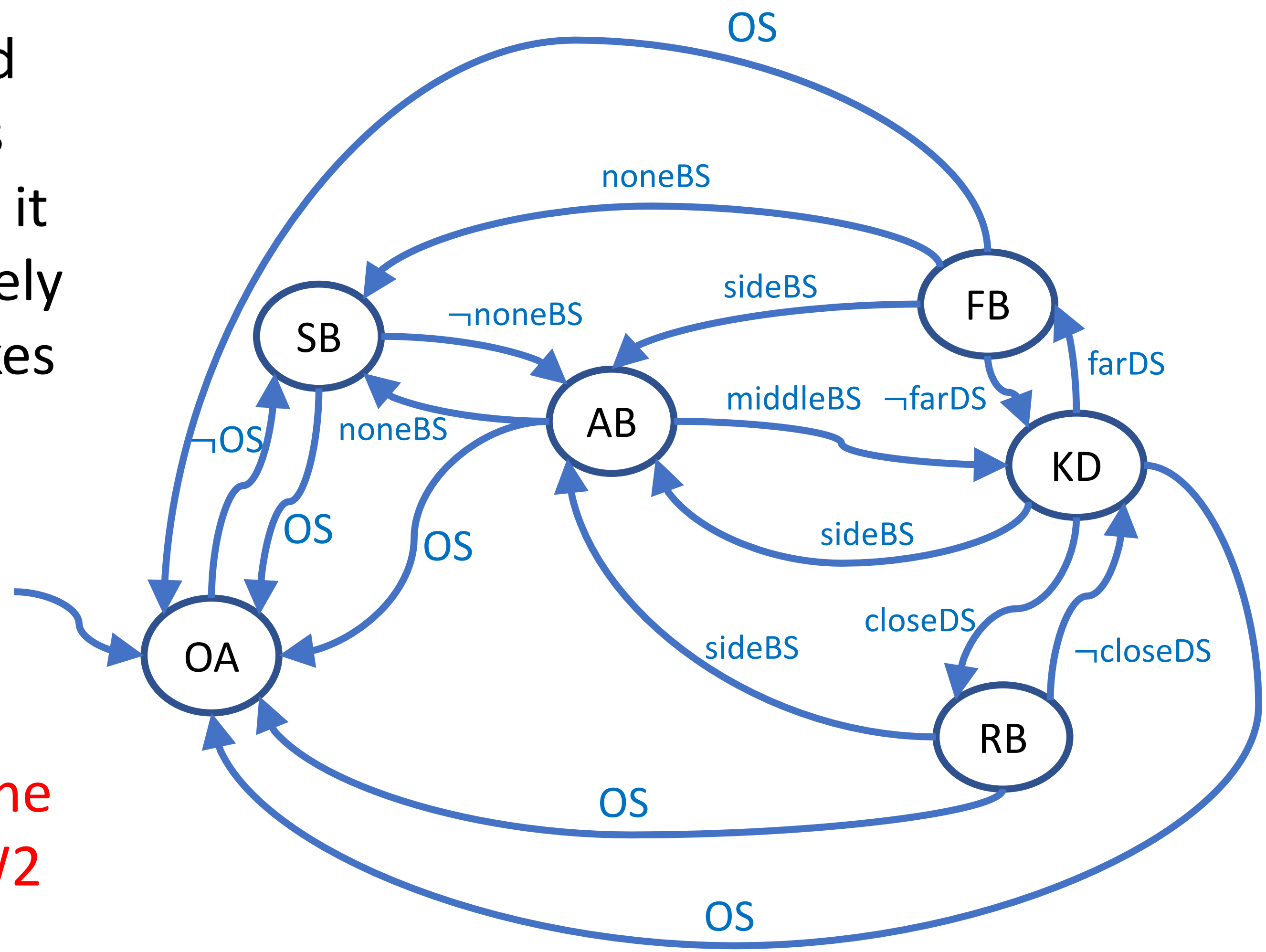
CW2: Nested Finite State Machines - Autonomous Car



CW2: Nested Finite State Machines - Autonomous Car

The non-nested FSM not only is harder to read, it is also more likely to make mistakes (for the same reason)

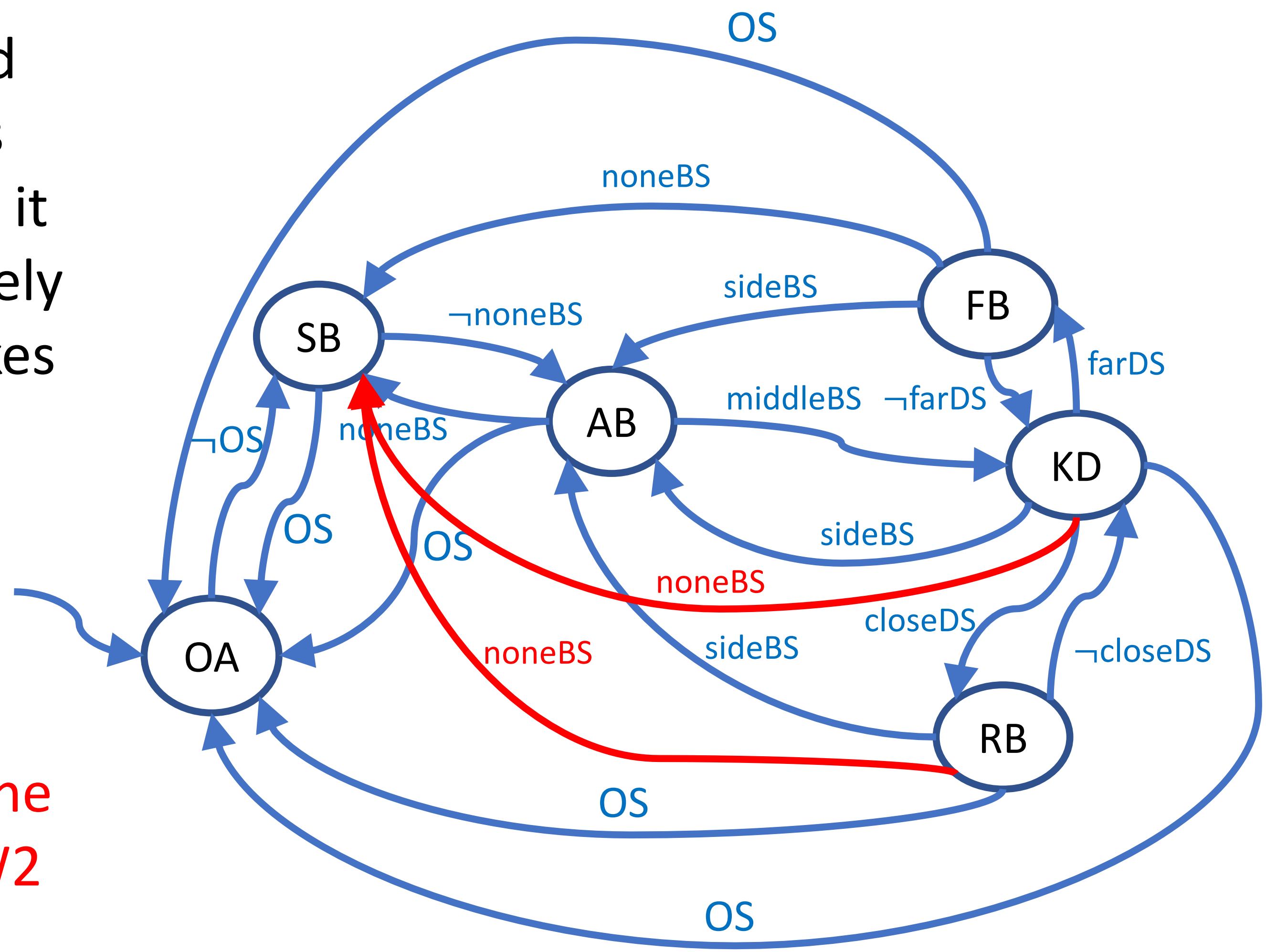
Can you spot some mistakes (incomplete behaviour) in the non-nested CW2 FSM?



CW2: Nested Finite State Machines - Autonomous Car

The non-nested FSM not only is harder to read, it is also more likely to make mistakes (for the same reason)

Can you spot some mistakes (incomplete behaviour) in the non-nested CW2 FSM?



CW2: Autonomous Car Implementation of Control Loop

```
stateMB = <inactive>; // nested state not active
while (forever) {
   if (OS) {
      // [OA] out: stop car
   } else {
      if (noneBS) {
         // [SB] out: search blob (refine)
      } else {
         if (sideBS) {
            stateMB = <inactive>; // nested state not active
            // [AB] out: turn to adjust facing
         } else {
            distanceState = ... // use distance to determine state
            switch (distanceState) {
            case tooclose:
                // [RB] out: drive car reverse to reduce distance
                break;
            case toofar:
                // [RB] out: drive car forward to get more distance
                break;
            case distok:
                // [KD] out: stop car in order to keep distance
            }
         }
      }
   }
} // while
```

CW2: Autonomous Car Implementation of Control Loop

- The structure of the nested FSMs determines the priority of services provided by the car.
- The more top-level a transition is in a Nested FSM, the higher prior it its service.
- In our example the FSM nesting levels can be directly translated into control flow with if-then branches.

Outlook

- Next lecture (that would follow on a module on that topic):

  Visual sensing