AI for Game Programming I

Distance Course – 7.5 Credits

Uppsala University

Mikael Fridenfalk

Course Overview

Schedule

Deterministic AI

A1

Assignment 1.1 (G)

Finite State Machines

Assignment 1.2 (VG)

Artificial Life

Machine Learning

A2

Assignment 2.1 (G)

Artificial Neural Networks

Assignment 2.2 (VG)

Genetic Algorithms

Project

A3

Assignment 3 (G)

Assignment of Choice

Grading Rules

Grade **G** *requires that all G-level assignments have been passed.*
Grade **VG** *requires that all G and VG-level assignments have been passed.*

Introduction

While in academic AI we try to understand what intelligence is and to develop intelligent machines, in Game AI we primarily try to increase the realism of games by implementing systems that give the impression to be intelligent.

In Game AI, we typically use different AI-methods for *game content generation* than for the computer game itself. Example of a method used mostly in the game itself is the Finite State Machine.

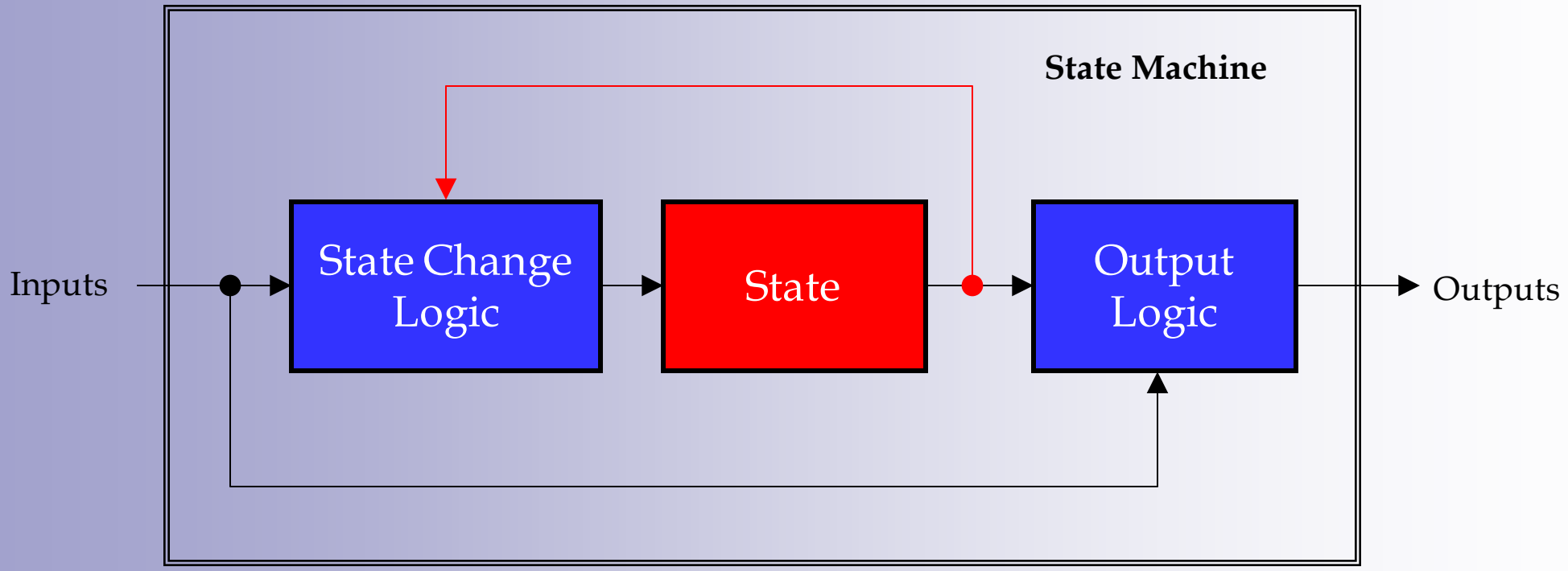
Finite State Machines

Additional reading at
https://brilliant.org/wiki/finite-state-machines
http://en.wikipedia.org/wiki/Finite_state_machine

Finite State Machines

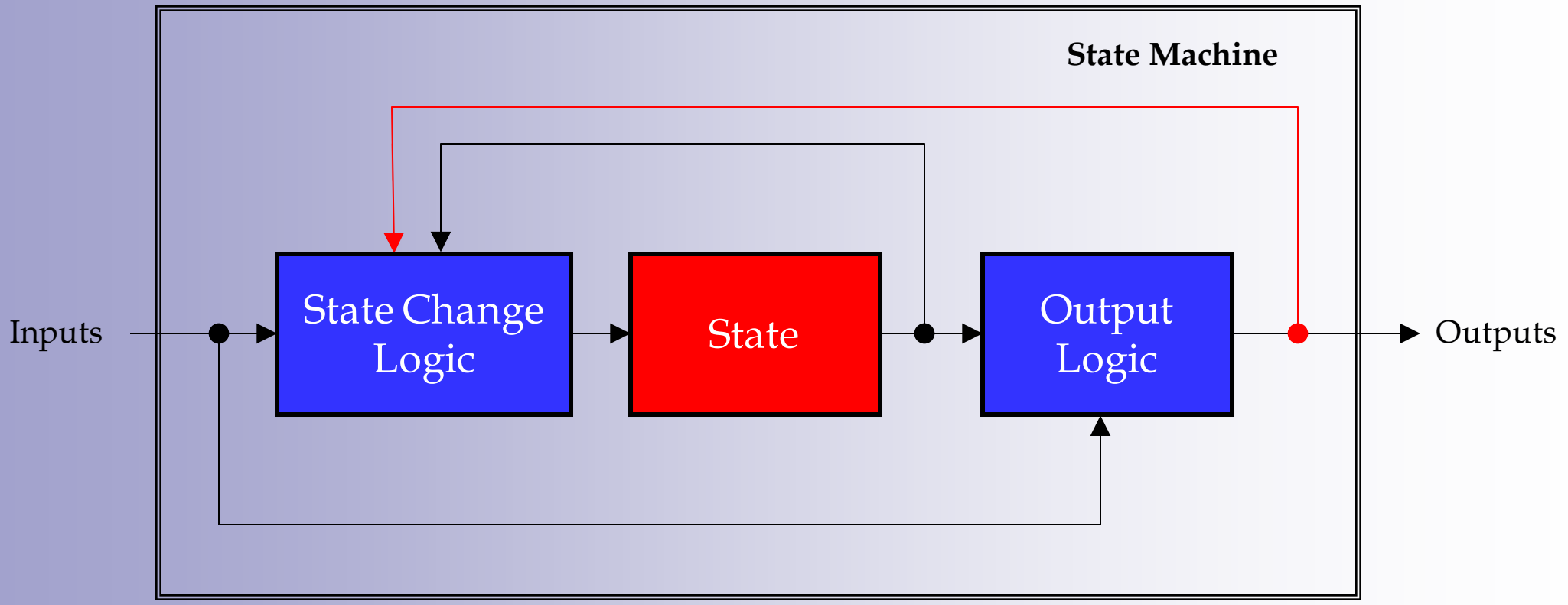
The most common AI applications in games are based on the Finite State Machine, from now on called FSMs or simply “State Machines”.

Finite State Machines



The Common Block Diagram of a State Machine

Finite State Machines



A More Generalized Block Diagram of a State Machine

Scripting

Development of AI-systems for games may be performed indirectly using scripts such as Lua, see:
http://www.lua.org or
http://lua-users.org/wiki/TutorialDirectory

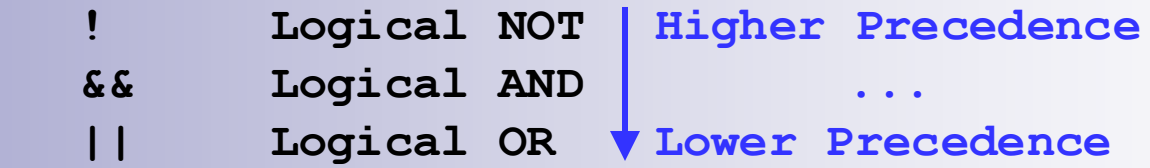
Scripting can be used for indirect programming of state machines. Although scripting is a non-related AI-topic and thus outside the scope of this course, scripts are in general useful for high-level programming, for simplification of the development process of AI game content, or to simply save time by avoiding frequent recompilation of the AI-source code.

Message Systems

To increase the impression of realism, Message Systems may be implemented to allow information exchange between objects, in the same way that information is spread in the real world (i.e., not to all objects *and at once*, but usually with some amount of restrictions *and delays*).

Logic

Operator Precedence in C/C++



Example

```
!A || !B && !C    is equal to    (!A) || ((!B) && (!C))
```

To avoid confusion and improve the readability of the code, some programmers include redundant parentheses to emphasize precedence.

Logic in C/C++ Syntax

```
0 = FALSE
1 = TRUE
```

Boolean Algebra

Examples of laws applied in the Boolean Algebra are de Morgan's Laws:

```
!( A || B )    is equal to    !A && !B

!( A && B )    is equal to    !A || !B
```

By using such laws, it is often possible to simplify boolean expressions and to write more time-efficient code.

Evaluation of Boolean Expressions – *Important*

In boolean expressions (C/C++) *only the part is calculated that is needed for calculation of the whole expression*. Example:

```
int a = 1, b = 1, c = 0;
if ((a == b) || (++c == 1)) a++;
```

Which gives: a = 2, b = 1 and c = 0.
Since, (a == b) is true in the expression above, (++c == 0) is thus not evaluated. Thus, c remains equal to 0 after the execution of the if-statement.

State Updates

Clock Cycles

To be able to synchronize state transitions in electronics, clock cycles are usually real clock cycles consisting typically of a square wave. In programming, clock cycles may be replaced by regular or non-regular state updates.

The *z*-notation (good to know)

In engineering, *z* denotes present (discrete) time/event, *z-1* previous (discrete) time/event and *z+1* next (discrete) time/event.
As an example, if we have a clock that updates our state machine (say, once each second), last update is called *z-1*, the update before that, *z-2*, etc., and in the same way, next update is called *z+1*, etc.

Online/Offline Simulation

Online Simulation denotes in this context that the system has the same time frame as the user. System's and user's time are synchronized. This is always the case in real-time systems.

Offline Simulation denotes similarly that the system's time frame is independent of the user's. One second, e.g., for the system, may correspond to a nanosecond in the time frame of the user, or if there are heavy calculations involved, a few seconds in system's time frame (for instance the dynamics simulation of a mechanical system) may correspond to a few hours in user's time frame.

Emulation of state updates in programming

We could use the system clock to continuously update our state machine. Another possibility is something as simple as an *offline* text-based input-output program, which updates the state machine each time we make an input (and press Enter).

Assignment 1.1

Example of a Security System based on a State Machine

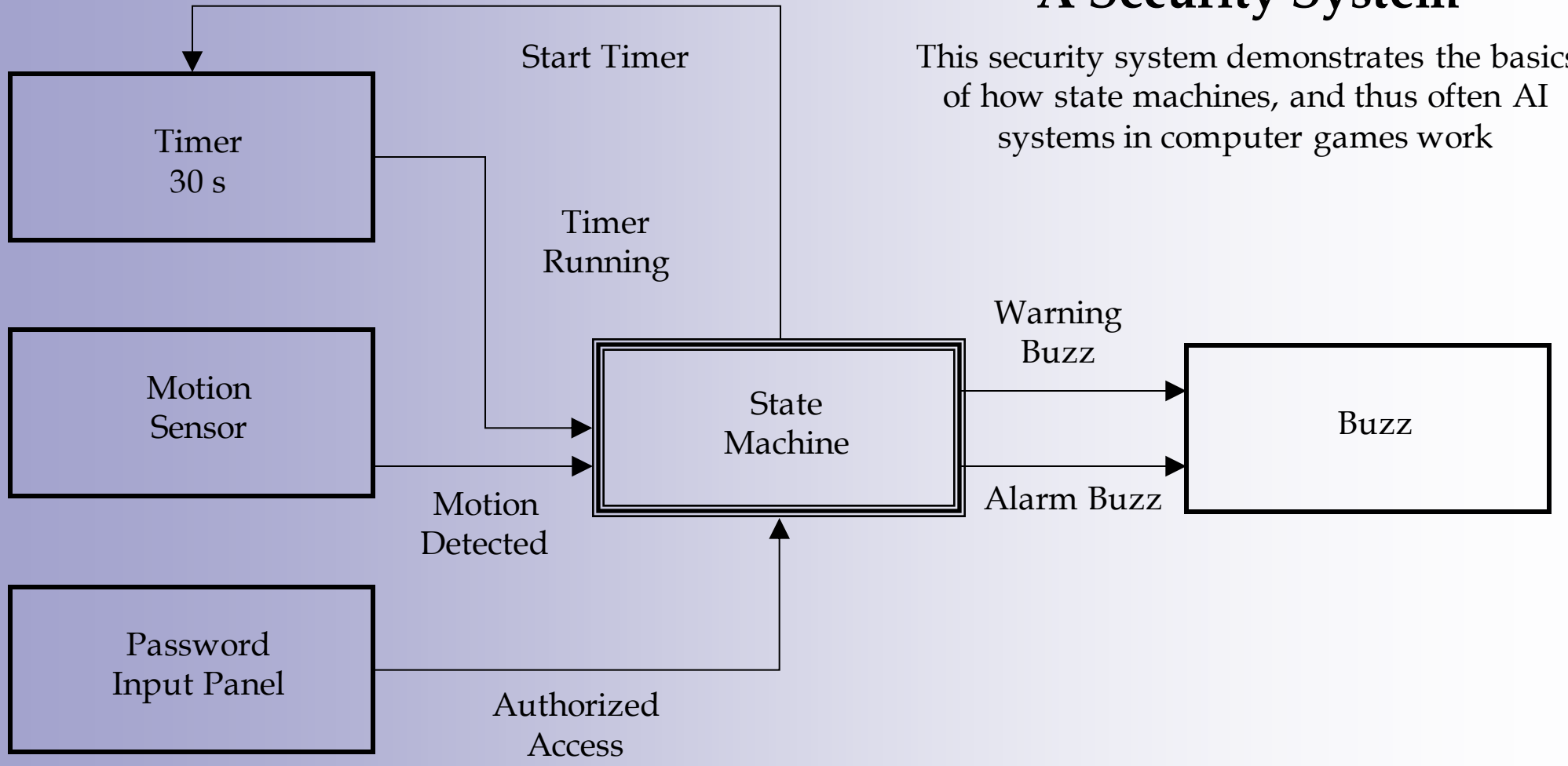
Take your time to study this section carefully. It is not as easy as it may look at the first glance.

The Block Diagram

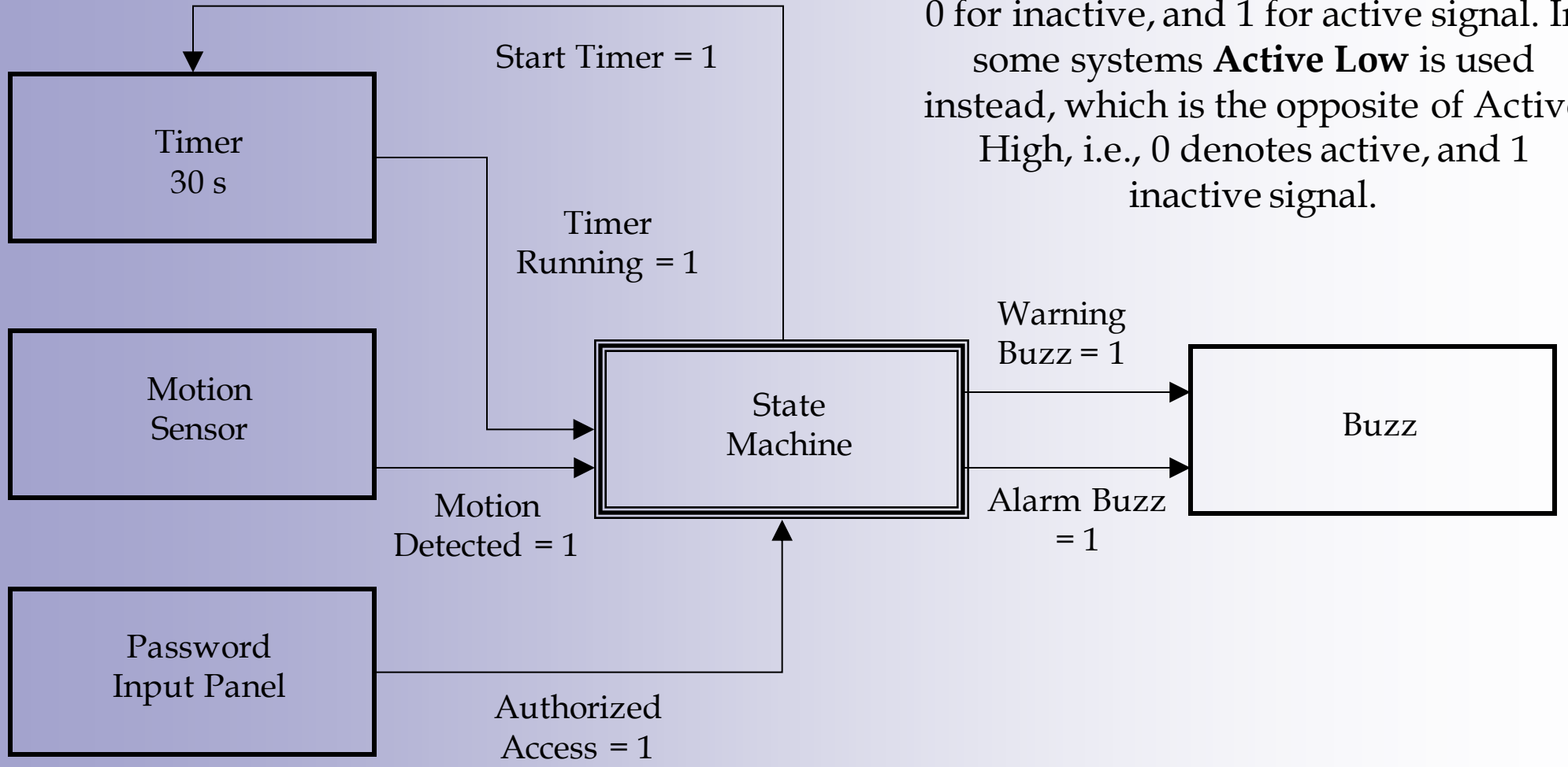
Additional reading at
https://se.mathworks.com/discovery/block-diagram.html

A Security System

This security system demonstrates the basics of how state machines, and thus often AI systems in computer games work



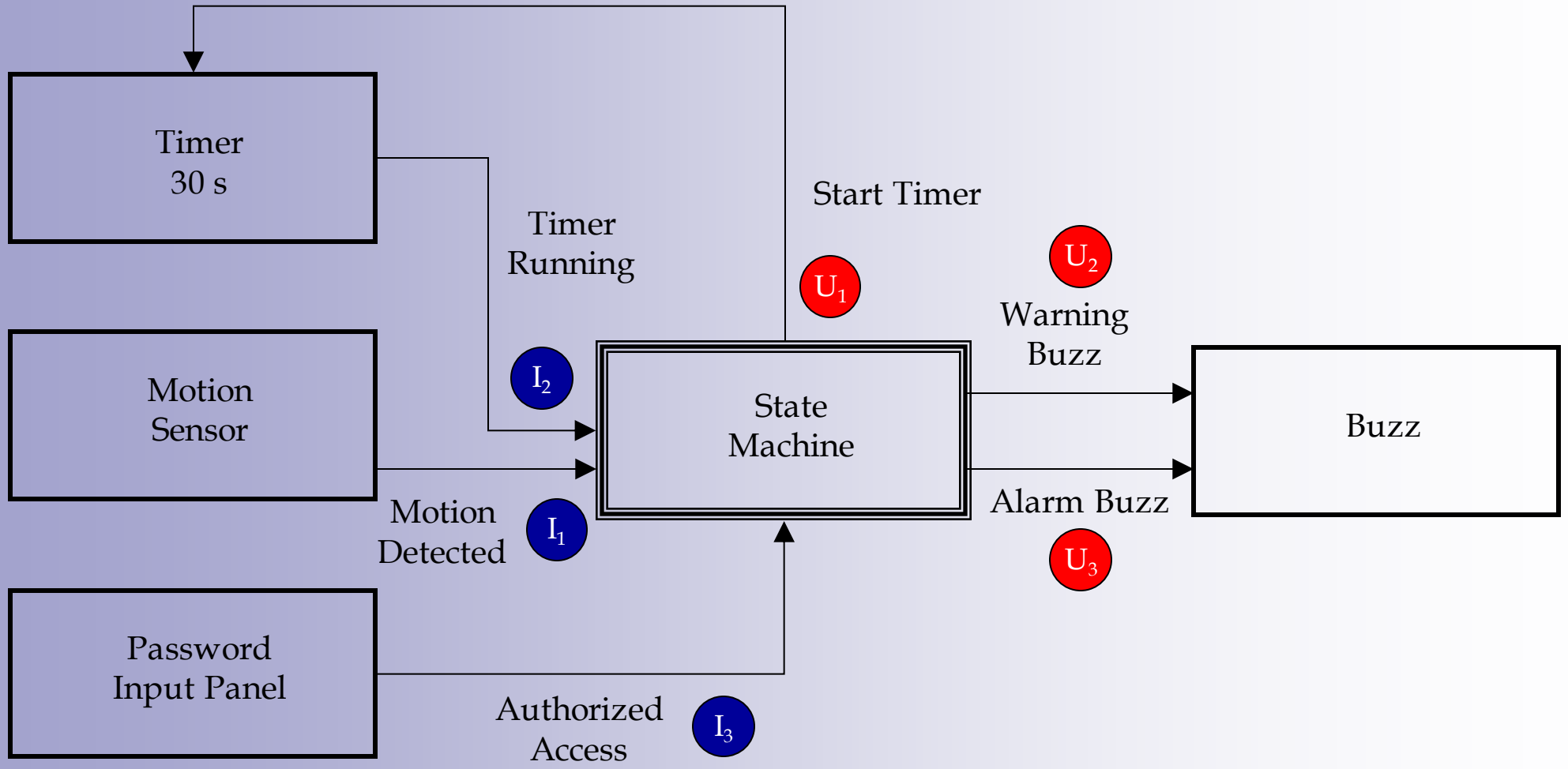
Example 1



Here, all outputs are **Active High**, i.e., 0 for inactive, and 1 for active signal. In some systems **Active Low** is used instead, which is the opposite of Active High, i.e., 0 denotes active, and 1 inactive signal.

Signal Value Definitions

Example 1



Definition of Alias, IN/OUT Signals

Example 1

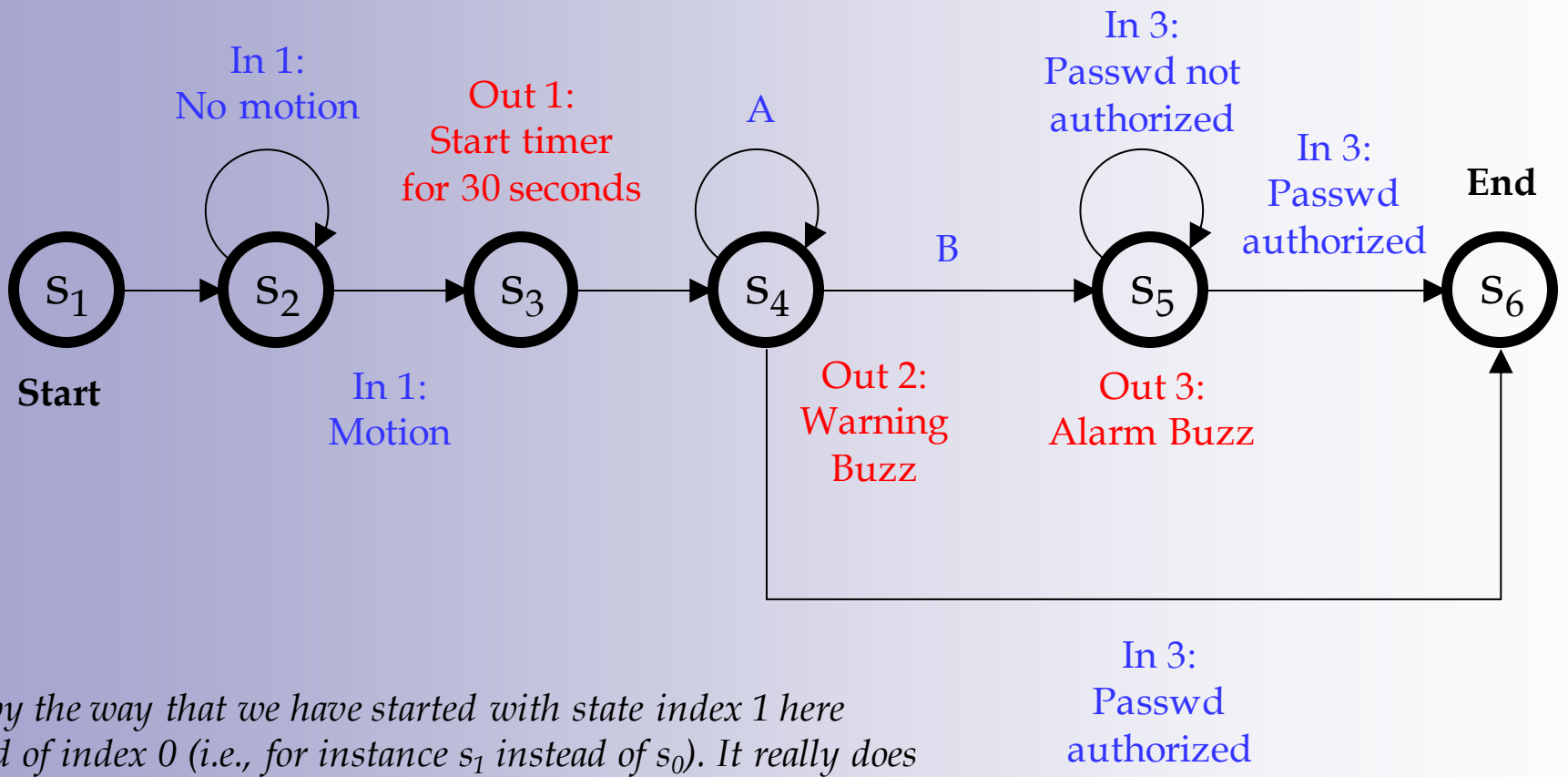
The State Diagram

Additional reading at
https://se.mathworks.com/discovery/state-diagram.html

A = (In 2: Timer *running*) AND (In 3: Passwd not authorized)

B = (In 2: Timer *stopped*) AND (In 3: Passwd not authorized)

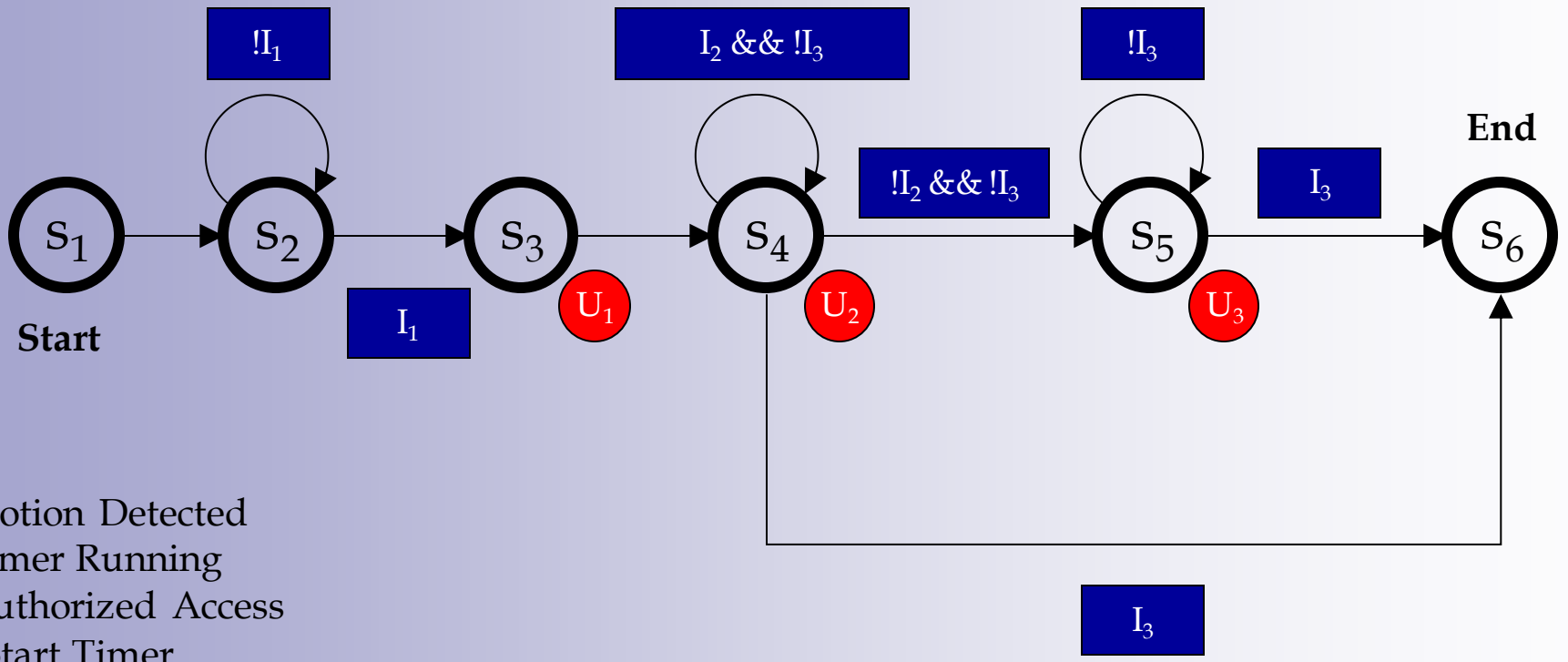
State Diagram



Note by the way that we have started with state index 1 here instead of index 0 (i.e., for instance s_1 instead of s_0). It really does not make any difference, as long as we use consistent definitions.

Example 1

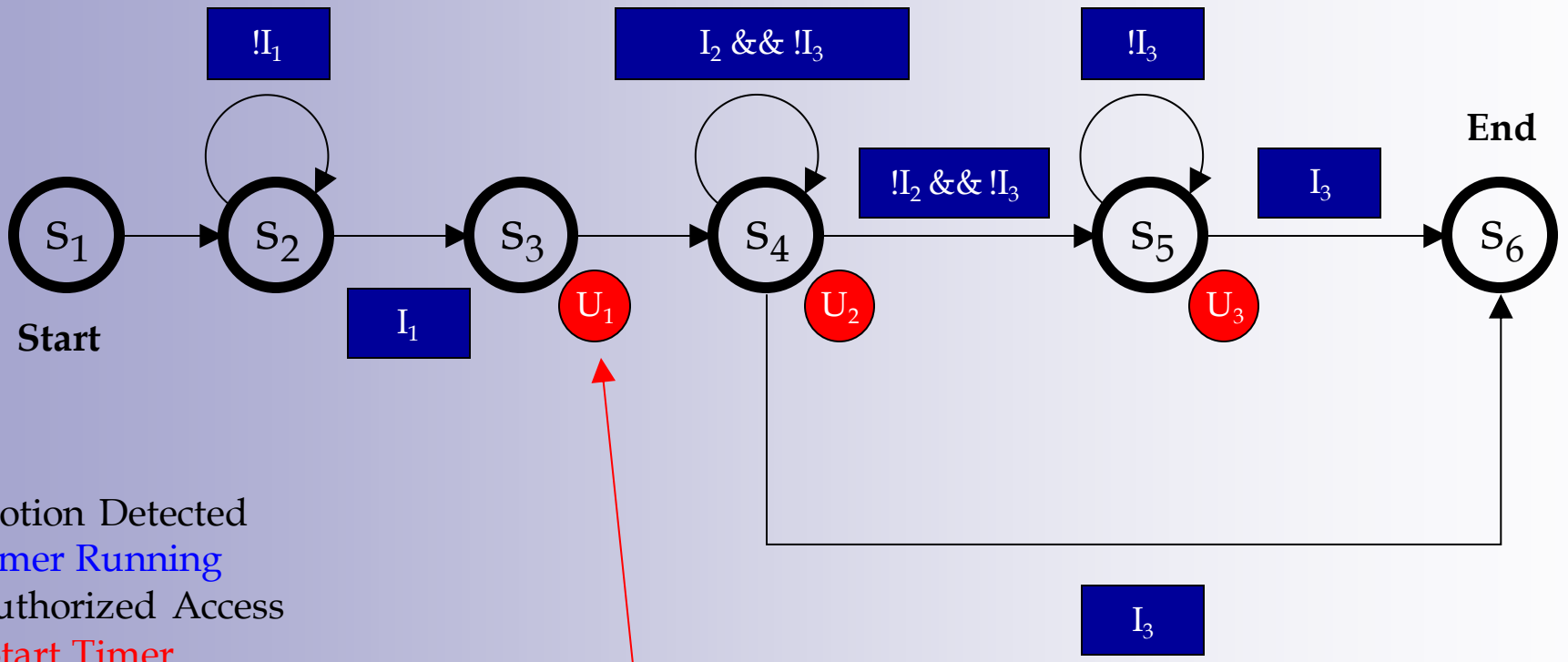
State Diagram with C-Syntax Logic Denotation



I1: Motion Detected
I2: Timer Running
I3: Authorized Access
U1: Start Timer
U2: Warning Buzz
U3: Alarm Buzz

Example 1

State Diagram with C-Syntax Logic Denotation

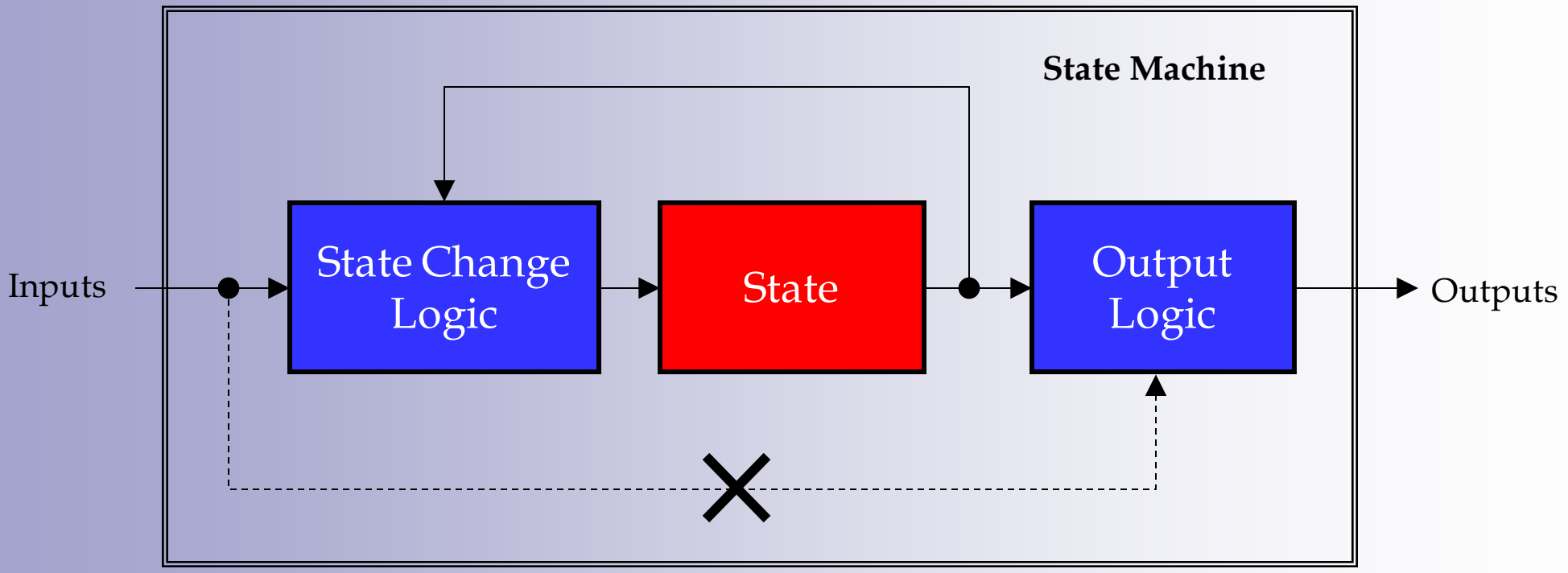


I1: Motion Detected
I2: Timer Running
I3: Authorized Access
U1: Start Timer
U2: Warning Buzz
U3: Alarm Buzz

Here, U_1 means for instance that $U_1 = 1$, while all other (U_2 and U_3) are 0, etc.

Example 1

Observe that in our special case the outputs U_1-U_3 are entirely determined by the state (s_1-s_6) and not directly affected by the inputs I_1-I_3.



Example 1

The Truth Table

Additional reading at
https://www.britannica.com/topic/truth-table

I1: Motion Detected
I2: Timer Running
I3: Authorized Access

U1: Start Timer
U2: Warning Buzz
U3: Alarm Buzz

Truth Table

State (z)	I1	I2	I3	U1	U2	U3	Next State (z+1)
1	X	X	X	0	0	0	2
2	0	X	X	0	0	0	2
2	1	X	X	0	0	0	3
3	X	X	X	1	0	0	4
4	X	1	0	0	1	0	4
4	X	0	0	0	1	0	5
4	X	X	1	0	1	0	6
5	X	X	0	0	0	1	5
5	X	X	1	0	0	1	6
6	X	X	X	0	0	0	6

Example 1

I1: Motion Detected
I2: Timer Running
I3: Authorized Access

U1: Start Timer
U2: Warning Buzz
U3: Alarm Buzz

Example 1

Truth Table

Change of state

State (z)	I1	I2	I3	U1	U2	U3	Next State (z+1)
1	X	X	X	0	0	0	2
2	0	X	X	0	0	0	2
2	1	X	X	0	0	0	3
3	X	X	X	1	0	0	4
4	X	1	0	0	1	0	4
4	X	0	0	0	1	0	5
4	X	X	1	0	1	0	6
5	X	X	0	0	0	1	5
5	X	X	1	0	0	1	6
6	X	X	X	0	0	0	6

Example 1

I1: Motion Detected
I2: Timer Running
I3: Authorized Access

U1: Start Timer
U2: Warning Buzz
U3: Alarm Buzz

Truth Table

0 = FALSE 1 = TRUE X = TRUE or FALSE

Change of state from s1 to s2

State (z)	I1	I2	I3	U1	U2	U3	Next State (z+1)
1	X	X	X	0	0	0	2
2	0	X	X	0	0	0	2
2	1	X	X	0	0	0	3
3	X	X	X	1	0	0	4
4	X	1	0	0	1	0	4
4	X	0	0	0	1	0	5
4	X	X	1	0	1	0	6
5	X	X	0	0	0	1	5
5	X	X	1	0	0	1	6
6	X	X	X	0	0	0	6

I1: Motion Detected
I2: Timer Running
I3: Authorized Access

U1: Start Timer
U2: Warning Buzz
U3: Alarm Buzz

Example 1

Truth Table

Input of no consequence

State (z)	I1	I2	I3	U1	U2	U3	Next State (z+1)
1	X	X	X	0	0	0	2
2	0	X	X	0	0	0	2
2	1	X	X	0	0	0	3
3	X	X	X	1	0	0	4
4	X	1	0	0	1	0	4
4	X	0	0	0	1	0	5
4	X	X	1	0	1	0	6
5	X	X	0	0	0	1	5
5	X	X	1	0	0	1	6
6	X	X	X	0	0	0	6

X: Don't Care

Don't Care means that 0 or 1 makes no difference. By using Don't Cares, we can reduce the number of lines in the truth table. In this example we use only 10 lines instead of 48 (6 states, 3 in => $6*2^3$).

I1: Motion Detected
I2: Timer Running
I3: Authorized Access

U1: Start Timer
U2: Warning Buzz
U3: Alarm Buzz

Truth Table

State (z)	I1	I2	I3	U1	U2	U3	Next State (z+1)
1	X	X	X	0	0	0	2
2	0	X	X	0	0	0	2
2	1	X	X	0	0	0	3
3	X	X	X	1	0	0	4
4	X	1	0	0	1	0	4
4	X	0	0	0	1	0	5
4	X	X	1	0	1	0	6
5	X	X	0	0	0	1	5
5	X	X	1	0	0	1	6
6	X	X	X	0	0	0	6

Transition states, i.e. states that only stay one clock cycle

Example 1

A Sliding Door Example

A Sliding Door System

Draw the block diagram, the state diagram and truth table of a system that opens and closes an electric sliding door.

When the door is closed, it can be opened by the user pushing a button. The door then opens by an electric motor and the motor stops when the door is fully opened.

When the door is open, it can be closed again by pushing the same button. Then it closes by the same electric motor that opened it, running in reverse. The motor is stopped when the door is fully closed.

The motor drive circuitry has two inputs. When forward input is on, the motor runs forward. When backward input is on, the motor runs in reverse direction. If both inputs are 0 or 1, the motor rests.

Example 2

A Sliding Door System

The system should consist of 4 blocks with the following input/output configurations:

Block 1: User Button
Output 1: Button Pushed

Block 2: Sensor Feedback
Output 1. Door Fully Open
Output 2. Door Fully Closed

Block 3: Motor Drive Circuits
Input 1: Run Motor Forward (Open the Door)
Input 2: Run Motor Reverse (Close the Door)

Block 4: State Machine
Input 1: Button Pushed
Input 2: Door Fully Open
Input 3: Door Fully Closed
Output 1: Run Motor Forward (Open the Door)*
Output 2: Run Motor Reverse (Close the Door)*
**Outputs 1 and 2 in block 4 should be mutually exclusive, i.e. both cannot have the value of 1.*

A Sliding Door System

The system should consist of 4 states:

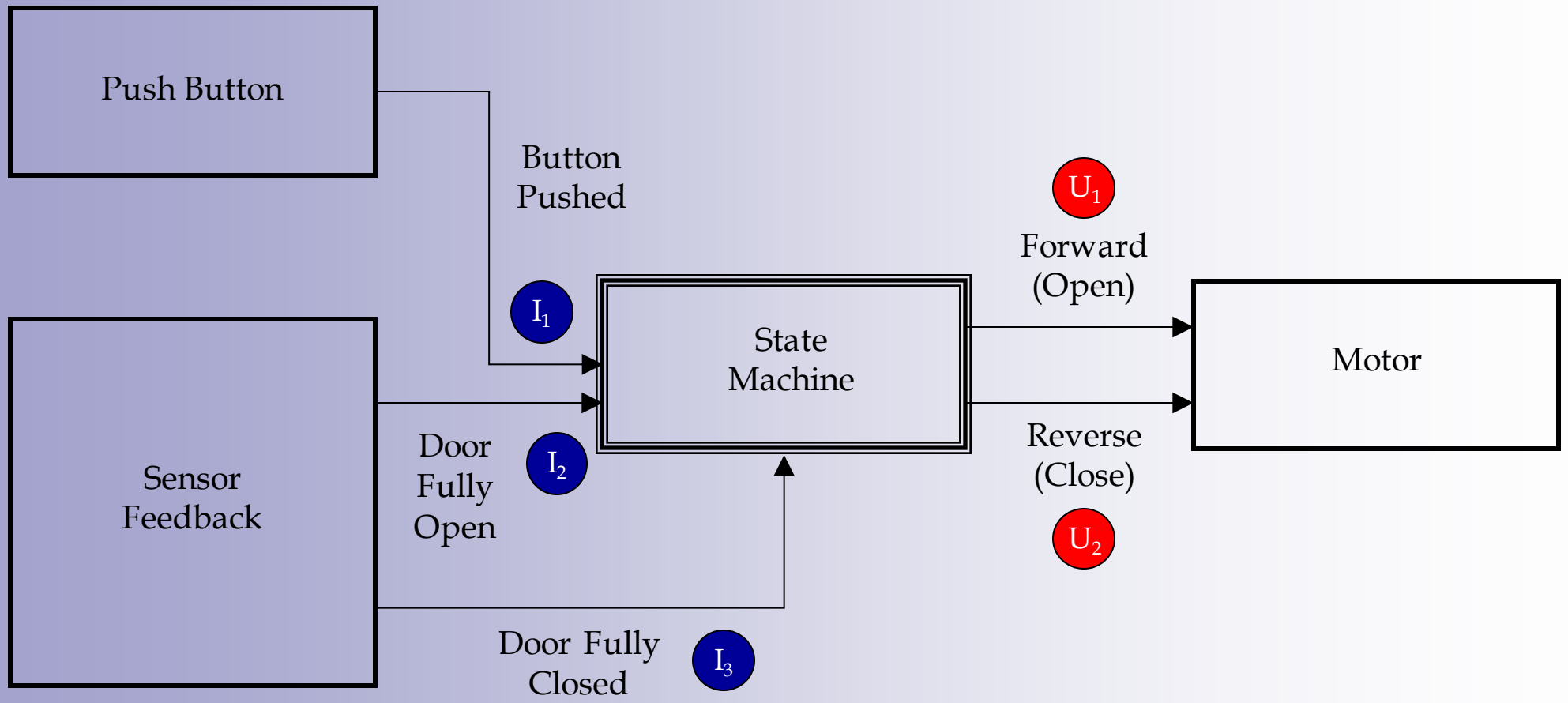
State 1:
Door (Fully) Closed

State 2:
Door Opening

State 3:
Door (Fully) Open

State 4:
Door Closing

Solution: **Block Diagram**



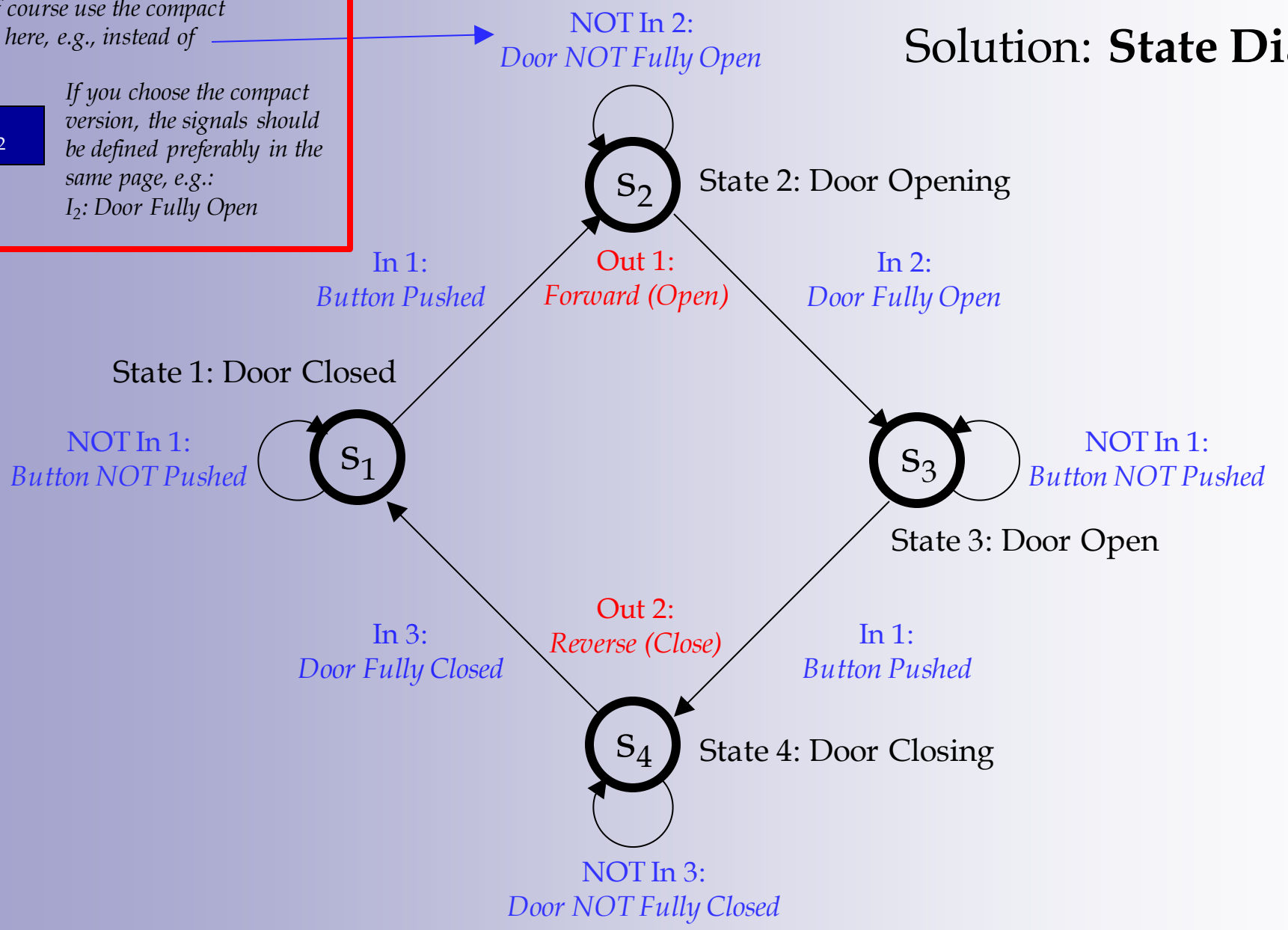
Solution: **State Diagram**

Comment

You can of course use the compact denotation here, e.g., instead of

$!I_2$

If you choose the compact version, the signals should be defined preferably in the same page, e.g.:
I_2: Door Fully Open



Example 2

Solution: **Truth Table**

State 1: Door is Closed
State 2: Door is Opening
State 3: Door is Open
State 4: Door is Closing

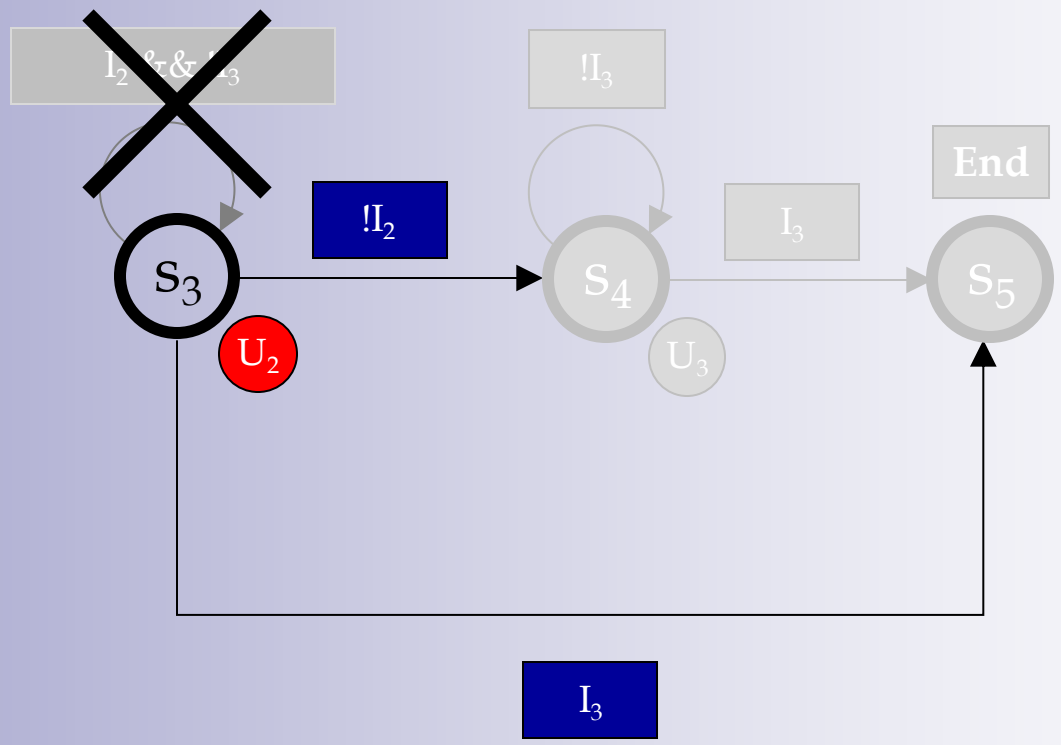
I1: Button Pushed
I2: Door is Opened
I3: Door is Closed

U1: Run Motor in Forward Direction (Open the Door)
U2: Run Motor in Reverse Direction (Close the Door)

State (z)	I1	I2	I3	U1	U2	Next State (z+1)
1	0	X	X	0	0	1
1	1	X	X	0	0	2
2	X	0	X	1	0	2
2	X	1	X	1	0	3
3	0	X	X	0	0	3
3	1	X	X	0	0	4
4	X	X	0	0	1	4
4	X	X	1	0	1	1

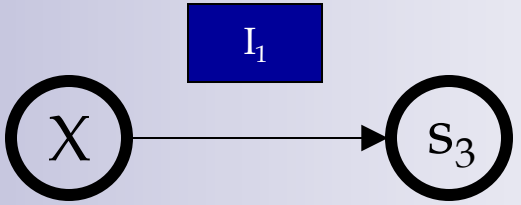
Additional Notes

Simplified Notation (1)



Sometimes it simplifies the graph to skip the non state-change condition, such as above where it is clear that the state is not changed if !(!I_2 || I_3), i.e. I_2 && !I_3.

Simplified Notation (2)



The *arbitrary* state X, borrowed from Truth Tables, represents all states. In the example above, if I_1 is true, next state will be "s_3", no matter which state is the present one.

Assignment 1.1

Overview

Assignment 1.1 (G)

Complete the security system in Example 1 in this lecture, so it can be reversed to start position, by expansion of the state diagram, the truth table, and the completion of this new system in code.

Assignment 1.1 (G)

To carry out this assignment, start by downloading the Adobe Acrobat Reader DC application from:
https://get.adobe.com/se/reader/otherversions/
(or any similar page under adobe.com)
and install this application on your computer.

Assignment 1.1 (G)

From the course homepage, download the forms A11_Alt1.pdf and A11_Alt2.pdf, and open these in Acrobat Reader. Select the form that is easiest to use from your point of view. Only one of these froms is expected to be submitted for grading.

Assignment 1.1 (G)

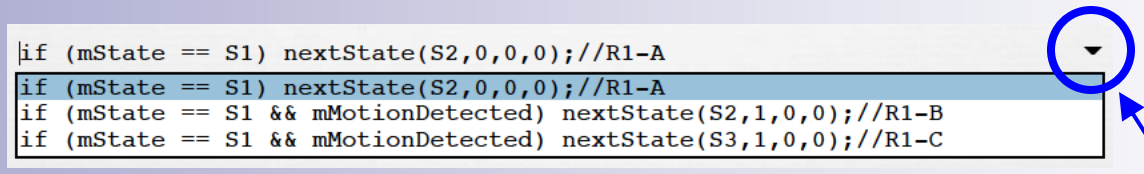
On the first page (state diagram), double-click the left mouse button on a input field to make an entry, e.g., IA, and type in a solution, such as I1, an expression such as !I3, or even an empty field if U1, U2, and U3 all are equal to zero.

Assignment 1.1 (G)

On the second page (truth table), double-click on a field to make an entry, e.g., A5, and type in the solution, such as 0, 1, or X.

Assignment 1.1 (G)

On the third page (code), select the lines of code that best would implement this FSM, by clicking on the triangle at the right side of each line to make a selection.



Click here to obtain a list of alternatives...

Assignment 1.1 (G)

Save this PDF by Acrobat Reader to your hard disk (for instance by File -> Save) and upload this file to the course assignment page “A1.1 – Finite State Machines”.