Faculty of Computer Systems & Software Engineering

Formal methods.

Solving tasks

Vitaliy Mezhuyev

Modelling semaphore traffic light system

Write TLA specification of the semaphore traffic light system, having the three possible states (Red, Yellow, Green).

```
EXTENDS Naturals
VARIABLE c, i
CONSTANT Red, Yellow, Green
colors == <<Red, Yellow, Green, Yellow>>

TypeInvariant == /\ c \in colors
                 /\ i \in (1..4)
Init == /\ c = Red
        /\ i =1

Next ==     /\ i' = (i+1)%4
            /\ c' = colors[i]
 Spec == Init /\ [Next]_<<i, c>>

THEOREM Spec => [] TypeInvariant
```

Modelling mechanical movement

Develop TLA specification of linear mechanical movement of a material point, bounded in the box with coordinates of borders x=0 and x=5. When a point touches a border it reverse the direction of movement. The movement is discrete – each step of material point is 1 unit of distance.

Analyses of task

- What variables we need?
- Where and how to express the border condition (x=0 and x=5)?
- How to specify the changes of direction of movement?
- Do we have changes of speed or its constant?
- Is it enough only one x (distance) variable?

Mechanical movement – TLA model

```
EXTENDS Integers, TLC

\* Distance and velocity
VARIABLES  x, v



\* Point cannot leave the box
\* So this is type invariant
TypeInvariant ==    /\ x <= 5
                    /\ x >= 0
```

Mechanical movement – TLA model

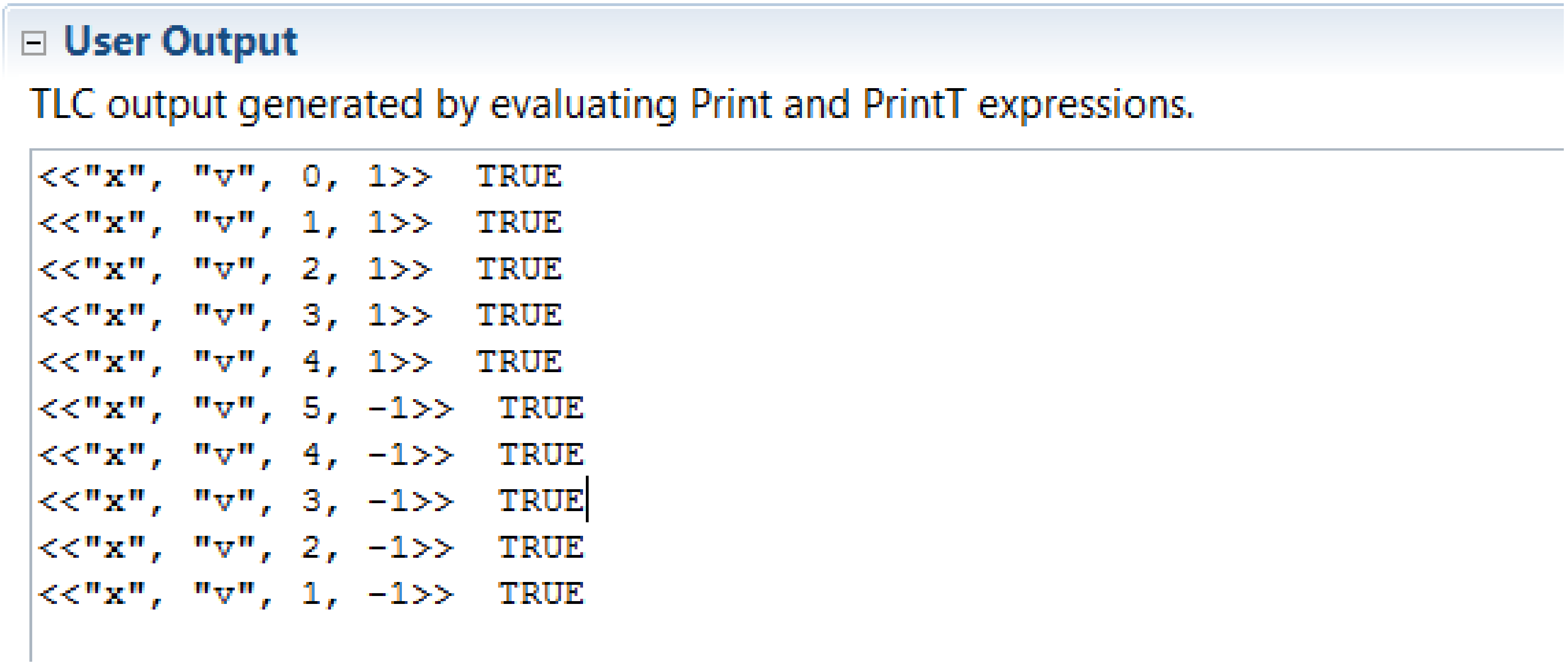
```
Init == /\ x = 0
        /\ v = 1
        /\ TypeInvariant

Next ==   /\ x' = x + v
          /\ IF ((x' = 0) \/ (x' = 5)) THEN v' = -v ELSE v' = v

Spec == Init /\ [][Next]_<<x, v>>

THEOREM Spec => [] TypeInvariant
```

Mechanical movement – TLC output


User Output

TLC output generated by evaluating Print and PrintT expressions.

```
<<"x", "v", 0, 1>>  TRUE
<<"x", "v", 1, 1>>  TRUE
<<"x", "v", 2, 1>>  TRUE
<<"x", "v", 3, 1>>  TRUE
<<"x", "v", 4, 1>>  TRUE
<<"x", "v", 5, -1>>  TRUE
<<"x", "v", 4, -1>>  TRUE
<<"x", "v", 3, -1>>  TRUE
<<"x", "v", 2, -1>>  TRUE
<<"x", "v", 1, -1>>  TRUE
```

We have 10 distinct states. Why not 12?

Task about a cat, a mouse and a cheese

When a rain comes, the cat goes in the room or in the basement. When the cat is in the room, the mouse goes in a hole, and the cheese is put in the fridge. If the cheese on the table, and the cat in the basement, then mouse in the room. Now comes the rain, and cheese on the table. What are resulted distinct state space?

Analyses of the task - definition of variables and constants

Define Cat, Mouse, Cheese as Variables

VARIABLES Cat, Mouse, Cheese

Define Room, Base, Table, Hole, Fridge as Constants

CONSTANT Room, Base, Table, Hole, Fridge

Analyses of the task - Definition of type invariant and initial condition

```
TypeInvariant ==
            /\ Cheese \in {Table, Fridge}
            /\ Cat \in {Room, Base}
            /\ Mouse \in {Room, Hole}

Init ==     /\ Cheese = Table
            /\ TypeInvariant
```

Analyses of the task - definition of actions

```
Next1  ==
          /\ Cat = Room
          /\ Mouse' = Hole
          /\ Cheese' = Fridge
          /\ UNCHANGED <<Cat>>
```

```
\* To print values use
/\ Print (<<"Cat, Mouse, Cheese", Cat, Mouse, Cheese>>, TRUE)
```

Analyses of the task - definition of actions

```
Next2  ==
          /\ Cat = Base
          /\ Mouse' = Room
          /\ Cheese = Table
          /\ UNCHANGED <<Cat, Cheese>>

\* To print values use
/\ Print (<<"Cat, Mouse, Cheese", Cat, Mouse, Cheese>>, TRUE)
```

Analyses of the task - definition of actions

```
Next1  == /\ IF Cat = Room THEN
                              /\ Mouse' = Hole
                              /\ Cheese' = Fridge
                    ELSE UNCHANGED <<Mouse, Cheese>>
                    /\ UNCHANGED <<Cat>>


Next2  == /\ IF Cat = Base /\ Cheese = Table THEN
                 /\ Mouse' = Room
                ELSE UNCHANGED Mouse
                /\ UNCHANGED <<Cat, Cheese>>
```

Specification and theorem to prove

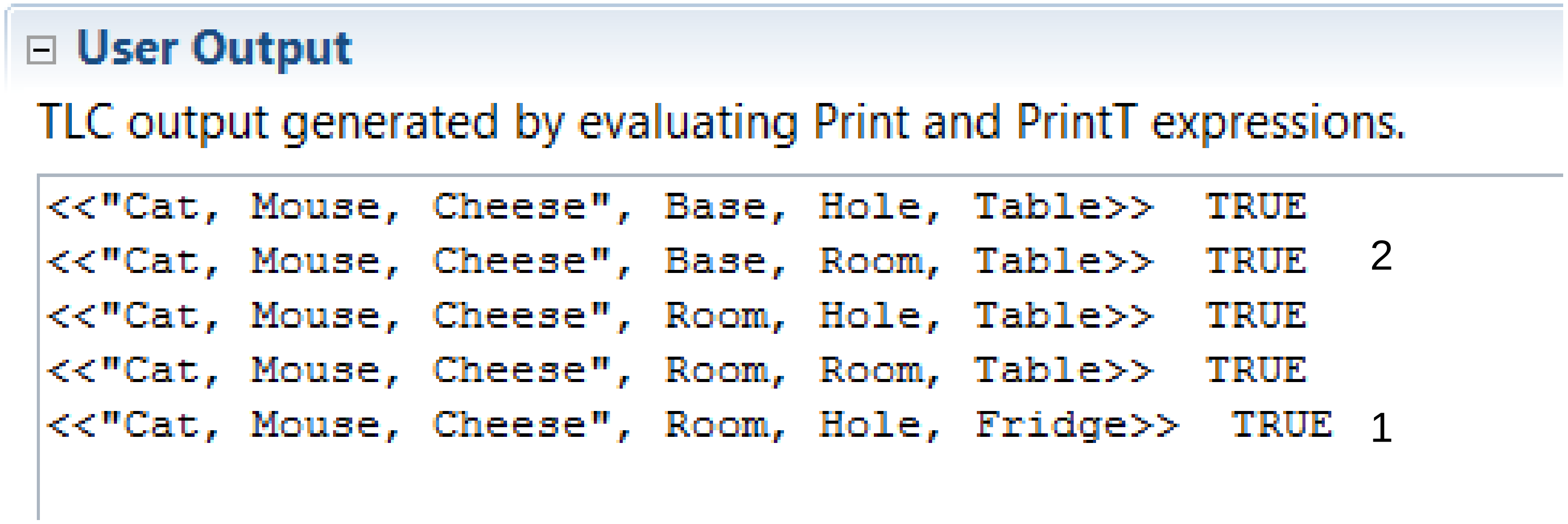
```
Spec == Init /\ [][Next1 \/ Next2 ]_<<Cat, Mouse, Cheese>>
```

```
THEOREM Spec => []TypeInvariant
```

Resulted state space

1. When the cat is in the room, the mouse goes in a hole, and the cheese is put in the fridge.

2. If the cheese on the table, and the cat in the basement, then mouse in the room.

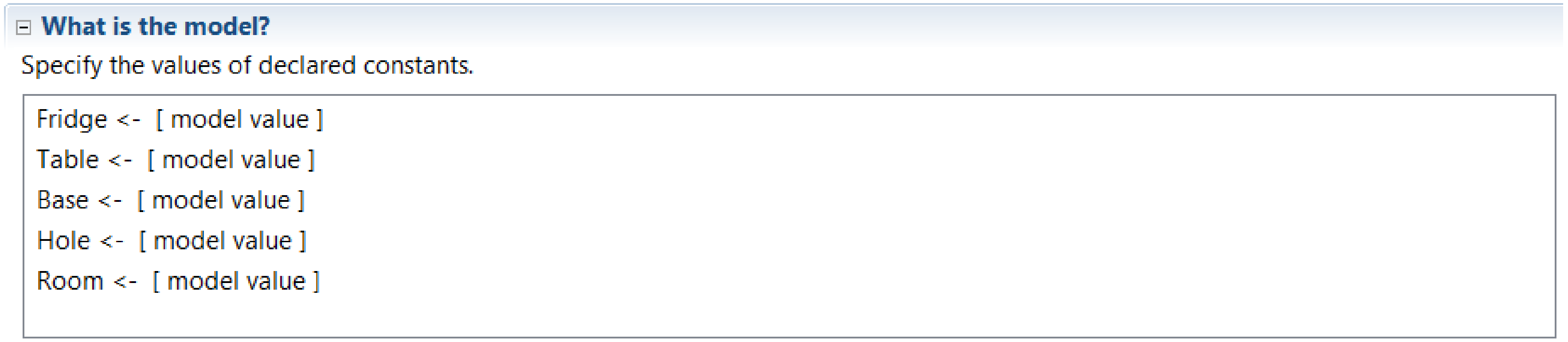


User Output

TLC output generated by evaluating Print and PrintT expressions.

```
<<"Cat, Mouse, Cheese", Base, Hole, Table>>  TRUE
<<"Cat, Mouse, Cheese", Base, Room, Table>>  TRUE   2
<<"Cat, Mouse, Cheese", Room, Hole, Table>>  TRUE
<<"Cat, Mouse, Cheese", Room, Room, Table>>  TRUE
<<"Cat, Mouse, Cheese", Room, Hole, Fridge>>  TRUE  1
```

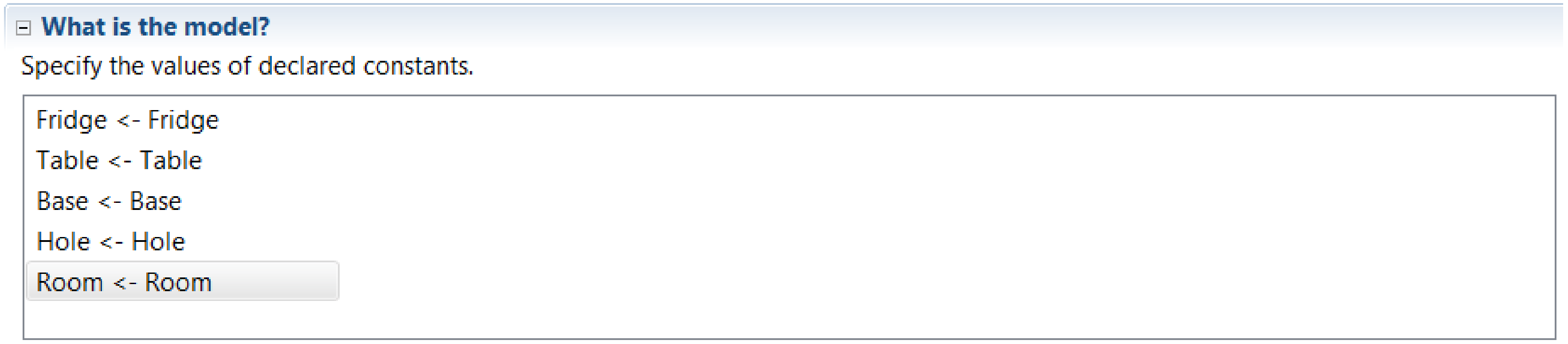
Definition of constants in Toolbox



What is the model?

Specify the values of declared constants.

```
Fridge <-  [ model value ]
Table <-  [ model value ]
Base <-  [ model value ]
Hole <-  [ model value ]
Room <-  [ model value ]
```



What is the model?

Specify the values of declared constants.

```
Fridge <- Fridge
Table <- Table
Base <- Base
Hole <- Hole
Room <- Room
```

Modelling tasks communication in concurrent software system

Tasks in a concurrent software system cannot communicate directly, but via special synchronisation entities – lets call it **Ports**. Task can send a data to and receive a data from a Port.

Communication between tasks and Port is *synchronous* (has waiting semantics). Task *send* a data to Port and *wait*, till other task takes the data. Data is stored in a FIFO. Specify safety poperty that no Not all tasks in the system are waiting.

Analyses of a task

1. We need develop models of a Task and a Port.
2. System can have several tasks and ports.
3. Task and a Port will have different Names (Ids)
4. Each task has a status (active, waiting).
5. What is type invariant here? E.g. can we check what not all the tasks are waiting (deadlock).
6. How to model port? (it should store data of different tasks, so we need model of data)

Simplest solution – 2 tasks communicates via 1 port

```
EXTENDS Integers, TLC, Sequences

\* Definition of tasks and port variables
VARIABLES t1, t2, p
CONSTANT Act, Wait

\* Not all tasks are waiting
TypeInvariant == ~(t1 = Wait /\ t2 = Wait)

\* Initially tasks are active, port is an empty sequence
Init == /\ t1 = Act
        /\ t2 = Act
        /\ p = <<>>
```

Send and receive actions

```
Send1 == /\ Len(p) < 3
         /\ p' = Append (p, "d1")
         /\ t1' = Wait
         /\ UNCHANGED t2

Rcv1 ==  /\ Len(p) > 0
         /\ p' = Tail(p)
         /\ t1' = Act
         /\ UNCHANGED t2

\* The same for the second Task - Send2 and Rcv2

Next == Send1 \/ Rcv1 \/ Send2 \/ Rcv2

Spec == Init /\ [][Next]_<<t1, t2, p>>

THEOREM Spec => [] TypeInvariant
```

Work with arrays

```
EXTENDS Integers, TLC, Sequences

VARIABLES Ts, p
CONSTANT Act, Wait, Data

Task == [Name : <<>>, Val: Data, State : {Act, Wait}]



TypeInvariant == ~(\A t \in Ts : Ts.State = Wait)



Init == /\ Ts[1].Name = "t1"
        /\ Ts[1].State = Act
        /\ Ts[1].Val = “d1”
        /\ p = <<>>
```

```
Send(t) == /\ Len(p) < 3
           /\ p' = Append (p, t.Val)
           /\ t.State' = Wait



Rcv(t) == /\ Len(p) > 0
          /\ p' = Tail(p)
          /\ t.State' = Act



Next == \E t \in Ts : Send(t) \/ Rcv(t)

Spec == Init /\ [][Next]_<<Ts, p>>
```

THEOREM Spec => [] TypeInvariant

- In the movie Die Hard 3, the heroes must obtain exactly 4 gallons of water using a 5 gallon jug, a 3 gallon jug, and a water faucet. Develop TLA specification of this problem which will allow to solve it with TLC. Comment your statements.

```
-------------------------- MODULE DieHard ---------------------------
EXTENDS Naturals
(* This statement imports the definitions of the ordinary
operators on natural numbers, such as + *)
(* We next declare the variables *)
VARIABLES big,
\* The number of gallons of water in the 5 gallon jug
                  small
\* The number of gallons of water in the 3 gallon jug.
```

(* We now define TypeOK to be the type invariant, asserting that the value of each variable is an element of the appropriate set.

Note: TLA+ uses the convention that a list of formulas bulleted by /\ or \/ denotes the conjunction or disjunction of those formulas. Indentation of subitems is significant, allowing one to eliminate lots of parentheses. This makes a large formula much easier to read.*)

```
TypeOK == /\ small \in 0..3
          /\ big   \in 0..5
```

(* Now we define of the initial predicate that specifies the initial values of the variables *)

```
Init == /\ big = 0
        /\ small = 0
```

```
(* Now we define the actions that our hero can perform. *)
(* They can do three things: *)
(*   - Pour water from the faucet into a jug.              *)
(*   - Pour water from a jug onto the ground.               *)
(*   - Pour water from one jug into another                *)
FillSmallJug  == /\ small' = 3
                 /\ big' = big
 FillBigJug    == /\ big' = 5
                 /\ small' = small
 EmptySmallJug == /\ small' = 0
                  /\ big' = big
 EmptyBigJug   == /\ big' = 0
                  /\ small' = small
```

(* Consider pouring water from one jug into another. Since the jugs are not calibrated, when pouring from jug A to jug B, it makes sense only to either fill B or empty A. *)

(* So, pouring water from A to B leaves B with the lesser of (i) the water contained in both jugs and (ii) the volume of B. To express this mathematically, we first define Min(m,n) to equal the minimum of the numbers m and n. *)

Min(m,n) == IF m < n THEN m ELSE n

(* Now we define the last two pouring actions*)

```
SmallToBig == /\ big'   = Min(big + small, 5)
              /\ small' = small - (big' - big)
BigToSmall == /\ small' = Min(big + small, 3)
              /\ big'   = big - (small' - small)
```

(* We define the next-state relation *)

(* A Next step is a step of one of the six actions defined above. Hence, Next is the disjunction of those actions. *)

```
Next ==  \/ FillSmallJug
         \/ FillBigJug
         \/ EmptySmallJug
         \/ EmptyBigJug
         \/ SmallToBig
         \/ BigToSmall
```

(* We define the formula Spec to be the complete specification, asserting of a behavior that it begins in a state satisfying Init, and that every step either satisfies Next or else leaves the pair <<big, small>> unchanged *)

Spec == Init /\ [][Next]_<<big, small>>

(* Remember that our heroes must measure out 4 gallons of water. Obviously, those 4 gallons must be in the 5 gallon jug. So, they have solved their problem when they reach a state with big = 4. So, we define Solved to be the predicate asserting that big = 4. *)

Solved == big = 4

(* We find a solution by having TLC check if Solved is an invariant, which will cause a behavior ending in a states where Solved is true.

Such a behavior is the desired solution. *)

Definition of liveness property

Liveness == <> **Solved**

Z Specification

Z specification consists of four sections:

1. Given sets, data types, and constants
2. State definition (shown as Z schema)
3. Initial state (shown as Z schema)
4. Operations (shown as Z schema)

Elevator Problem

1. Given Sets

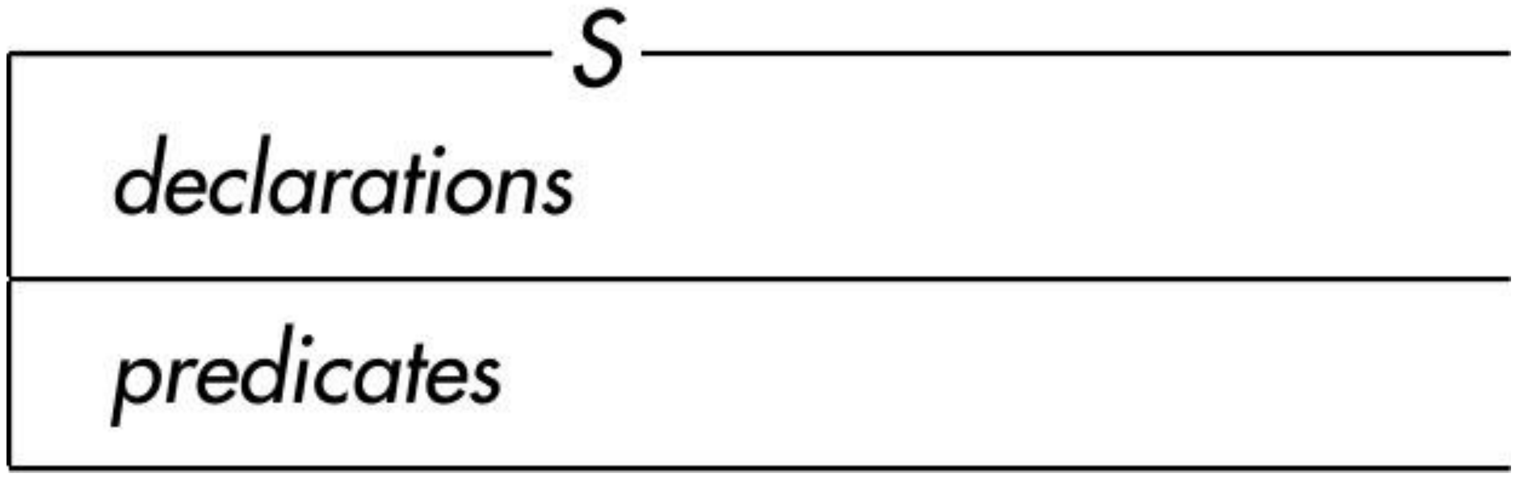
- Z Specification begins with a list of given sets, without details
  - Names appear in brackets
  - E.g. the given set define type of a button

    [Button]

Elevator Problem

2. State Definition

- Z specification consists of a number of schemata. The schema consists of
  - Schema name
  - Group of variable declarations
  - List of predicates that constrain values of variables



An elevator system is to be installed in a building with m floors

- The problem concerns the logic to move elevators between floors according to the following constraints:
- an elevator has a set of buttons, one for each floor. These illuminate when pressed and cause the elevator to visit the corresponding floor. The illumination is cancelled when the corresponding floor is visited by the elevator;
- each floor has two buttons (except the top and bottom floors), one to request upward travel and one to request downward travel. These buttons illuminate when pressed. The illumination is cancelled when an elevator visits the floor and is either moving in the desired direction or has no outstanding requests;
- when an elevator has no requests to service, it should remain at its final destination with its doors closed and await

Elevator Problem

Possible subsets of the set *Button*

- The floor buttons
- The elevator buttons
- buttons (the set of all buttons in the elevator problem)
- pushed (the set of buttons that have been pushed)

Elevator Problem

Button_State

$$
\begin{array}{ll}
floor\_buttons, elevator\_buttons & : \mathbf{P}\ Button \\
buttons & : \mathbf{P}\ Button \\
pushed & : \mathbf{P}\ Button \\
\hline
\multicolumn{2}{l}{floor\_buttons \cap elevator\_buttons = \emptyset} \\
\multicolumn{2}{l}{floor\_buttons \cup elevator\_buttons = buttons}
\end{array}
$$

Elevator Problem

3. Initial State

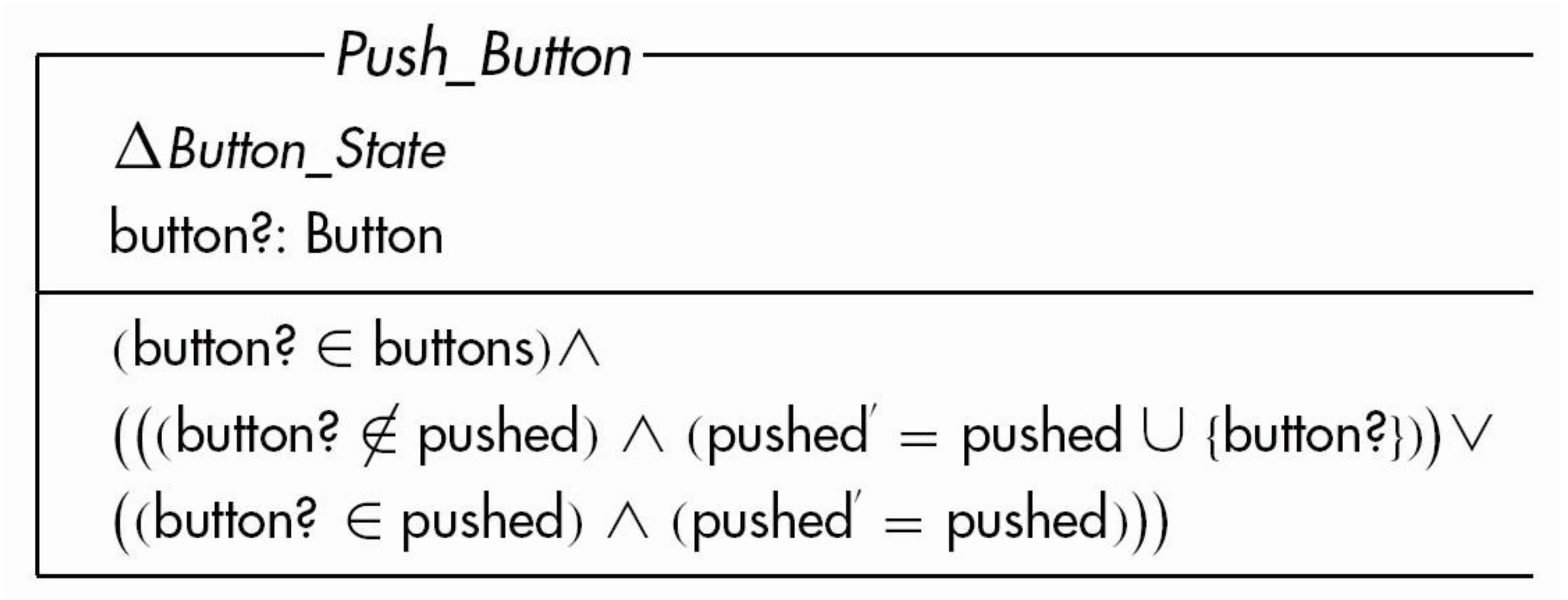
- State when the system is first turned on

$$Button\_Init \equiv [Button\_State' \mid \text{pushed}' = \varnothing]$$

(In the above equation, the ≡ should be a = with a ^ on top. Unfortunately, this is hard to type in PowerPoint!)

Elevator Problem

4. Operations



Push_Button

Δ*Button_State*

button?: Button

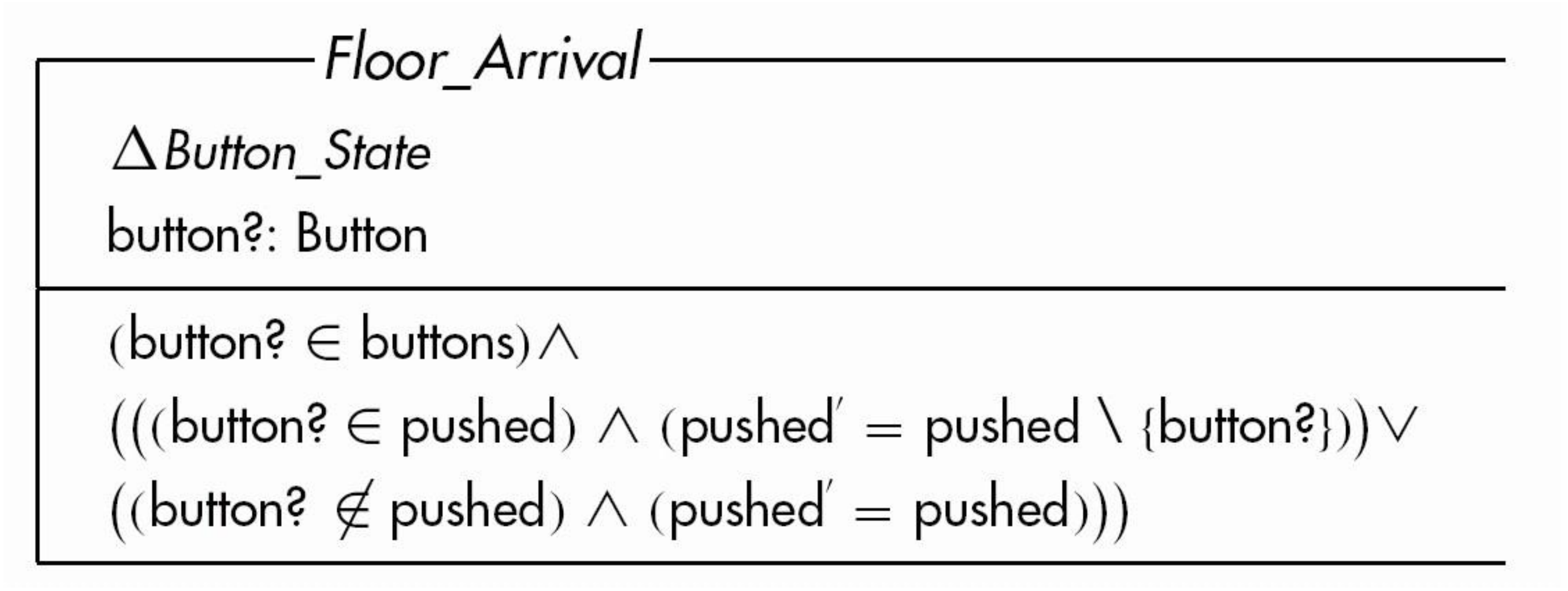
$(\text{button?} \in \text{buttons}) \wedge$

$(((\text{button?} \notin \text{pushed}) \wedge (\text{pushed}' = \text{pushed} \cup \{\text{button?}\})) \vee$

$((\text{button?} \in \text{pushed}) \wedge (\text{pushed}' = \text{pushed})))$

- Button pushed for first time is turned on, and added to set *pushed*
- Without third precondition, results would be unspecified

Elevator Problem



Floor_Arrival

Δ*Button_State*

button?: Button

$(\text{button?} \in \text{buttons}) \wedge$

$(((\text{button?} \in \text{pushed}) \wedge (\text{pushed}' = \text{pushed} \setminus \{\text{button?}\})) \vee$

$((\text{button?} \notin \text{pushed}) \wedge (\text{pushed}' = \text{pushed})))$

- If elevator arrives at a floor, the corresponding button(s) must be turned off
- The solution does not distinguish between up and down floor buttons

Using Z notation develop a formal specification the for the natural language specification.

A container can hold a discrete quantity of water. The contents of the container cannot exceed its capacity (5000 liters) at any time. A dial and warning lamp are attached to the container. The dial reading indicates the level of water inside the container. If the dial reading is below or equal a danger level, which is 50, the warning lamp will switch ON (signaled). When the warning lamp is switched on, an amount of water must be added to keep the normal level of water inside the container.

Definition of constants (CZT-IDE)

```
─
 STATE ::= ON | OFF
L


─
 CAPACITY == 5000
L
```

Definition of container (CZT-IDE)

```
┌ Container
  Dial : $\mathbb{N}$1
  Lamp : STATE
|
  Dial < CAPACITY
└
```

Definition of initial state (CZT-IDE)

```
┌ InitContainer
  ΔContainer
|
  Dial  = 1000
  Lamp = OFF
└
```

Adding water into container (CZT-IDE)

```
┌ AddWater
 ΔContainer
|
  Dial < 5000
   Dial′ = Dial + 1

  Dial  > 50
   Lamp′ = OFF
└
```

Using water from container

```
┌ UseWater
 ΔContainer
|
 Dial < 50
  ∧ Lamp′ = ON
  ∧ AddWater
└
```

Steps in Constructing a Z Specification

1. Decide on the basic sets, operations, and invariant relationships needed to describe the behavior of software to be developed.
2. Introduce a schema to define the state space for the software.
3. Decide what operations are needed to introduce state changes.
4. Identify inputs to and outputs from the system.
5. Introduce schema to induce state changes.
6. For each schema in step 5, give the necessary conditions (pre- and postconditions) that must be satisfied.

Thank you for your attention!
Please ask questions