

# 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

$colors == \langle\langle Red, Yellow, Green, Yellow \rangle\rangle$

$TypeInvariant == \wedge c \setminus in\ colors$

$\wedge i = 1$

$Init == TypeInvariant$

$Next == \wedge i' = (i+1)\%4$

$\wedge c' = colors[i]$

$Spec == Init \wedge [Next]_{\langle\langle i, c \rangle\rangle}$

**THEOREM**  $Spec \Rightarrow [] TypeInvariant$



# 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 has three possible semantics.

- W – Wait. Task *send* a data to Port and *wait*, till other task takes the data (this is called *synchronous* behaviour).
- NW – Not Wait. Task send a data in Port but not wait, till other task take it (this is called *asynchronous* behaviour).
- WT – Wait Timeout. Task send a data in Port and *wait* some period of a time, till other task take it, and becomes active.



## 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 ==  $\sim(t1 = \text{Wait} \wedge t2 = \text{Wait})$

\\* Initially tasks are active, port is an empty sequence

Init ==  $\wedge t1 = \text{Act}$

$\wedge t2 = \text{Act}$

$\wedge p = \langle \rangle$

# Send and receive actions

Send1 ==  $\wedge \text{Len}(p) < 3$   
           $\wedge p' = \text{Append}(p, "d1")$   
           $\wedge t1' = \text{Wait}$   
           $\wedge \text{UNCHANGED } t2$

Rcv1 ==  $\wedge \text{Len}(p) > 0$   
           $\wedge p' = \text{Tail}(p)$   
           $\wedge t1' = \text{Act}$   
           $\wedge \text{UNCHANGED } t2$

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

Next == Send1  $\vee$  Rcv1  $\vee$  Send2  $\vee$  Rcv2

Spec == Init  $\wedge [][\text{Next}]_{\langle t1, t2, p \rangle}$

THEOREM Spec  $\Rightarrow [] \text{TypeInvariant}$

# Work with arrays

**EXTENDS** Integers, TLC, Sequences

**VARIABLES** Ts, p


**CONSTANT** Act, Wait, Data

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

TypeInvariant ==  $\sim(\forall t \in Ts : Ts.State = Wait)$

Init ==  $\wedge Ts[1].Name = "t1"$   
 $\wedge Ts[1].State = Act$   
 $\wedge Ts[1].Val = "d1"$   
 $\wedge p = <<>>$





$\text{Send}(t) == \wedge \text{Len}(p) < 3$   
 $\wedge p' = \text{Append}(p, t.\text{Val})$   
 $\wedge t.\text{State}' = \text{Wait}$

$\text{Rcv}(t) == \wedge \text{Len}(p) > 0$   
 $\wedge p' = \text{Tail}(p)$   
 $\wedge t.\text{State}' = \text{Act}$

$\text{Next} == \exists t \in Ts : \text{Send}(t) \vee \text{Rcv}(t)$

$\text{Spec} == \text{Init} \wedge [][\text{Next}]_{\langle\langle Ts, p \rangle\rangle}$

**THEOREM**  $\text{Spec} \Rightarrow [] \text{TypeInvariant}$



# Z Specification Sections

Z specification consists of four sections:

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

# Elevator Problem: Z

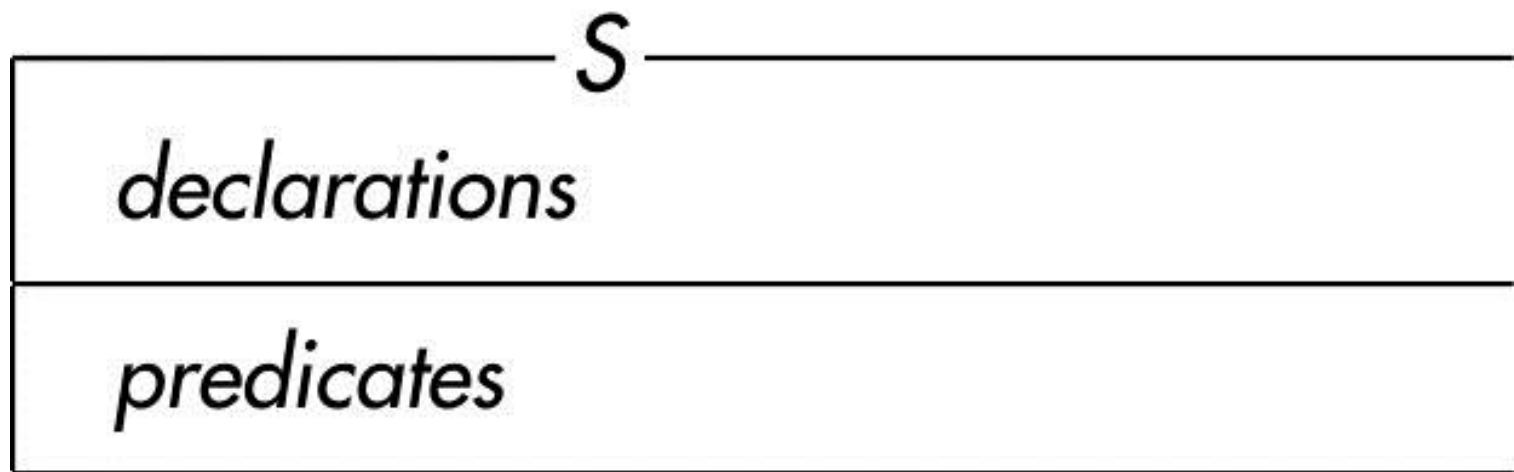
## 1. Given Sets

- A Z Specification begins with a list of given sets, sets that need not be defined in detail
  - Names appear in brackets
  - Here we need the set of *all* buttons
  - Specification begins  
[Button]

# Elevator Problem: Z

## 2. State Definition

- Z specification consists of a number of schemata. The schema consists of
  - Schema name
  - group of variable declarations, also called signature or declarations
  - List of predicates that constrain values of variables



# Elevator Problem: Z

Four subsets of *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: Z

## *Button\_State*

floor_buttons, elevator_buttons	: <b>P</b> Button
buttons	: <b>P</b> Button
pushed	: <b>P</b> Button

$\text{floor\_buttons} \cap \text{elevator\_buttons} = \emptyset$

$\text{floor\_buttons} \cup \text{elevator\_buttons} = \text{buttons}$

# Elevator Problem: Z

## 3. Initial State

- State when the system is first turned on

$$Button\_Init \equiv [Button\_State' \mid pushed' = \emptyset]$$

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

# Elevator Problem: Z (contd)

## 4. Operations

*Push\_Button*

$\Delta Button\_State$

button?: Button

$(button? \in buttons) \wedge$

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

$((button? \in pushed) \wedge (pushed' = pushed))$

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



# Elevator Problem: Z (contd)

*Floor\_Arrival*

$\Delta Button\_State$

button?: Button

$(button? \in buttons) \wedge$

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

$((button? \notin pushed) \wedge (pushed' = pushed))$

- 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

—

STATE ::= ON | OFF  
L

—

CAPACITY == 5000  
L

## Definition of container

⌊ Container

Dial :  $\mathbb{N}$

Lamp : STATE

|

Dial < CAPACITY

L

## Definition of initial state

⌈ InitContainer  
  ΔContainer  
  |  
    Dial = 1000  
    Lamp = OFF  
⌋

# Adding water into container

┌ AddWater

$\Delta$ Container

|

  Dial < 5000

    Dial' = Dial + 1

  Dial > 50

    Lamp' = OFF

└

## Using water from container

$\vdash$  UseWater  
 $\Delta$ Container  
|  
Dial < 50  
 $\wedge$  Lamp' = ON  
 $\wedge$  AddWater  
L



# 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.



# Structure of a Z Schema

----- **Schema Identifier** -----

| Declaration-Part  
| {status} State space identifier  
| Variable declaration  
Function declaration
Optional pre-condition
Optional post-condition
-----

# Syntax for a Z Schema Declaration

SchemaDeclaration ::=  
    { status } Identifier  
    Identifier: set\_type |  
    Identifier: domain  $\rightarrow$  range |  
    Identifier { ? | ! } : set\_type  
Status ::=  $\Delta$  |  $\Xi$

--Status indicates if state space is changed or not  
--Variable declaration  
--Function declaration  
--Input operation (?) or output operation (!)  
-- $\Delta$  (schema describes state change)  
-- $\Xi$  (state space does not change after operation)

# Example: Z Specification for tATC

- In this section, Z is used to specify a record-keeping system for airspace configurations for a training program of air traffic controllers (the tATC system).
- Let *Display* be a set of displays (each showing a different air traffic controller's view of an airspace). Each display is associated with its own name and icons (graphics of an airspace display).
- Let *known* be a set of recorded airspace displays, and let *airspace* be the name of a function that maps a known airspace to a display.

## Example (Cont.)

- The following schema defines the state space of the tATC display\_handling utility:

```
----- AirspaceCofig -----  
| known: ¶ Name × Icon  
| airspace: Name × Icon → Display  
|-----  
| known = dom airspace  
-----
```

- The idea behind this schema is to give a concise description of available airspace configurations, and how to establish an indexing system for these configurations.

## Example (Cont.)

- Now we can begin defining “methods” for initializing, changing the set of airspace configurations, drag-and-drop as well as click-and-see operations in an interactive tATC training program.
- The following schema describes the initial state of the tATC system:

```
----- initAirspaceConfig -----  
| AirspaceConfig  
|-----  
| known =  $\emptyset$   
-----
```

## Example (Cont.)

- In addition, the symbols ? (input) and ! (output) carry over from CSP in specifying input/output operations in Z.
- The operation to add a new configuration is specified with the AddAirspaceConfig schema:

```
----- AddAirspaceConfig -----  
|  $\Delta$  AirspaceConfig  
| config?: Name  $\times$  Icon  
| display?: Display  
|-----  
| known  $\cap \{\text{config?} \rightarrow \text{display}\} = \emptyset$   
| airspace' = airspace  $\cup \{\text{config?} \rightarrow \text{display?}\}$   
-----
```

## Example (Cont.)

- The AddAirspaceConfig schema induces a state change.
- By convention, unprimed variables denote a state *before* an operation has been performed while primed variables denote the state of a system *after* an operation has been performed.
- The input of a new configuration (config?) results in the replacement of the set named airspace with a new set of known airspace configurations named known'. This result can be expressed succinctly by asserting

$$\text{known}' = \text{known} \cup \{\text{config?}\}$$

## Example (Cont.)

- For simplicity, it is assumed that there are two types of icons in an airspace display: buttons to control the display, and graphics representing airspace features.
- The following schema can be used to specify a click-and-see operation:

```
----- clickAndseeAirspace -----  
|  $\Delta$  AirspaceConfig  
| config?: Name  $\times$  Icon  
| button?: Icon  
| click: Name  $\times$  Icon  $\times$  Icon  $\rightarrow$  Name  $\times$  Icon  
|-----  
| known  $\cap$  {config?}  $\neq \emptyset \wedge$   
| {button?}  $\neq \emptyset \wedge$   
| click(config?, button?)  $\in$  airspace  
-----
```



# Example (Cont.)

- The click operation maps these inputs to a new airspace. It is assumed that some of the graphics in a display can be moved(repositioned) by an air traffic control trainee. This makes it possible for a trainee to customize or tailor a display.
- Introduce a schema to describe a drag-and-drop operation:

```
----- dragAnddropAirspace -----  
|  $\Delta$  AirspaceConfig  
| config?: Name  $\times$  Icon  
| drag: Name  $\times$  Icon  $\rightarrow$  Name  $\times$  Icon  
| drop: Name  $\times$  Icon  $\rightarrow$  Display  
|-----  
| known  $\cap$  {config?}  $\neq \emptyset \wedge$   
| known  $\cap$  {drag(config?)}  $\neq \emptyset \wedge$   
| known  $\cap$  {drop(config?)}  $\neq \emptyset \wedge$   
| {drag(config?)}  $\neq \emptyset \wedge$   
| airspace' = airspace - {drag(config?)}  $\wedge$   
| airspace'' = airspace  $\cup$  {drop(config?)}  $\rightarrow$  Display  
-----
```

## Example (Cont.)

- The drag AnddropAirspace schema says the drag and drop operations result in a state change, provided that the value of input by config? is know and the drag operation is non-empty.
- There are preconditions for a state change. Performing a drag operation results in the deletion of an old configuration of the airspace (the new set of configurations is named airspace'). The new configuration of airspace resulting from the drop is added to known' (the new set of configurations is named airspace'').

## Example (Cont.)

- A description of the tATC display-handling facility can include an operation to save a copy of a display to a file. This is done with the SaveAirspaceConfig schema.

```
----- SaveAirspaceConfig -----  
|  $\exists$  AirspaceConfig  
| config?: Name  $\times$  Icon  
| copy!: Name  $\times$  Icon  
|-----  
| config?  $\in$  Known  
| copy! = airspace(config?)  
-----
```



# Z specification of Vocabulary System

The system intended to help in the acquisition of vocabulary by a student learning a foreign language. The system should able to

1. Record pairs of words, where one word is a native language and the other is a foreign language.
2. Each word of a pair may serve as translation of the other.
3. Words to be added to the vocabulary must satisfy orthographic rules.



# Functions to be provided by the system

- Valid pairs of words may be added to the vocabulary
- All translations of a native word into the foreign language may be requested
- All translation of a foreign word into the native language may be requested.
- In all cases, the system should able to report any detected errors in words submitted for entry into the vocabulary.

# Global definitions

- Three sets are assumed:

*[Native, Foreign, Message]*

- The orthographic rules are presented by subsets of *Native* and *Foreign*:

*OrthoNative : set of Native*

*OrthoForeign : set of Foreign*

# Global definitions

- Define the set of well-formed vocabulary (containing only words which satisfy the orthographic rules):

$$\begin{aligned} \textit{WellFormedVocabs} = & \\ & \{ V : \textit{set of (Native, Foreign)} | \\ & \quad \forall n : \textit{Native}; \forall f : \textit{Foreign} \cdot \\ & \quad (n, f) \in V \Rightarrow \\ & \quad n \in \textit{OrthoNative} \wedge f \in \textit{OrthoForeign} \} \end{aligned}$$

- Define three message:

$$\textit{Ok}, \textit{ErrorInInput}, \textit{WordNotKnown} : \textit{Message}$$

# Initialization

- Initialized by taking an empty vocabulary:

$$InitVocab == \{ Vocab' : set\ of\ (Native, Foreign) \mid Vocab' = \emptyset \}$$



### 3. Definition of Operations (1)

add new words into the vocabulary

- Words pair may be submitted for entry into the vocabulary:

*AddValidPair ==*

*{ Vocab, Vocab': set of (Native, Foreign);*

*n? : Native; f? : Foreign; rep! : Message |*

*Vocab ∈ WellFormedVocabs*

*∧ n? ∈ OrthoNative ∧ f? ∈ OrthoForeign*

*∧ Vocab' = Vocab ∪ {(n?, f?)}*

*∧ rep! = Ok}*

# Definition of Operations - check the existing words @vocabulary

- All translations of a given native word may requested, distinguishing the case of a word which satisfy the orthographic rules but does not occur in the current vocabulary.
- The given word must satisfy the orthographic rules of the native language. The output consists of all those foreign words which occur in the vocabulary paired with the given native word.
- This set of words will be empty if the native word is unknown to the vocabulary.

$$ToForeign == KnownToForeign \cup UnknownToForeign$$

# Word found @vocabulary

*KnownToForeign ==*

*{ Vocab, Vocab' : set of (Native, Foreign);*

*n? : Native; ftrans! : set of Foreign; rep! : Message |*

*Vocab ∈ WellFormedVocabs*

*∧ Vocab' = Vocab*

*∧ n? ∈ OrthoNative*

*∧ ftrans! = {f : Foreign | (n?, f) ∈ Vocab}*

*∧ ftrans! ≠ ∅ ∧ rep! = WordNotKnown}*

# Word do not found @vocabulary

*UnknownToForeign ==*

*{ Vocab, Vocab' : set of (Native, Foreign);*

*n? : Native; ftrans! : set of Foreign; rep! : Message |*

*Vocab ∈ WellFormedVocabs*

*∧ Vocab' = Vocab*

*∧ n? ∈ OrthoNative*

*∧ ftrans! = {f : Foreign | (n?, f) ∈ Vocab}*

*∧ ftrans! = ∅ ∧ rep! = WordNotKnown}*



# Try yourself!

Next, the given foreign word may be requested, distinguishing the case of a word which satisfies the orthographic rules but does not occur in the current vocabulary.

Write the specification *ToNative* {}

# Defining the Operations (3)

## total operation - AddPair

- Now, let's define operations which will work in all circumstances, informing the user when errors occur.

$TotalAddPair == AddValidPair \cup AddPairError$

where:

$AddPairError ==$

$\{ Vocab, Vocab' : set\ of\ (Native, Foreign);$

$n? : Native; f? : Foreign; rep! : Message \mid$

$Vocab \in WellFormedVocabs$

$\wedge n? \notin OrthoNative \wedge f? \in OrthoForeign$

$\wedge Vocab' = Vocab \cup \{(n?, f?)\}$

$\wedge rep! = ErrorInInput\}$

# total operation - *ToForeign*

$ToForeignTotal == ToForeign \cup ToForeignError$

Where :

$ToForeignError ==$

$\{Vocab, Vocab' : set\ of\ (Native, Foreign);$

$n? : Native; ftrans! : set\ of\ Foreign; rep! : Message|$

$Vocab \in WellFormedVocabs$

$\wedge Vocab' = Vocab$

$\wedge n? \notin OrthoNative$

$\wedge rep! = ErrorInInput \wedge ftrans! = \emptyset\}$



# Try yourself!


Write the specification for *TotalToNative* { }.




# Try yourself!

When a foreign language is studied which share common linguistic parentage with the native language the concept 'false friend' arise. For example

English	Spanish	True/false friend
Action	Accion	True
Deception	Decepcion	False
Final	Final	True
Possibility	Posibilidad	True
tender	tender	false



Now, suppose that we want to add a new facility to our vocabulary system: using predicates such as *n very similar to f* when necessary, give a definition of an operation *FalseFriends* which outputs a set of false friends in the vocabulary.



**Thank you for your attention!**  
**Please ask questions**