
CHAPTER 4

Z Schema Operators & Error Scenarios

Chapter Outline

- ❖ Z Schema Operators
 - ❖ Conjunction
 - ❖ Disjunction
 - ❖ Negation
 - ❖ Other schema operators
- ❖ Error Scenarios
- ❖ Complete Schema

Z Schema Operators

Schema Conjunction (\wedge)

- ❖ If S and T are two schemas then their **conjunction**, $S \wedge T$ is also a schema
- ❖ The declaration is a **merging** of the **two** declaration parts and **conjoining** their **predicate** parts.
- ❖ The result is a schema that **introduces both sets of variables** and **imposes both constraints**.
- ❖ Allows us to **specify different aspects** of a specification **separately**, and then **combine** them to form a **complete description**.

Schema Conjunction (\wedge)

❖ Example:

S	T
a: A	b: B
b: B	c: C
P	Q

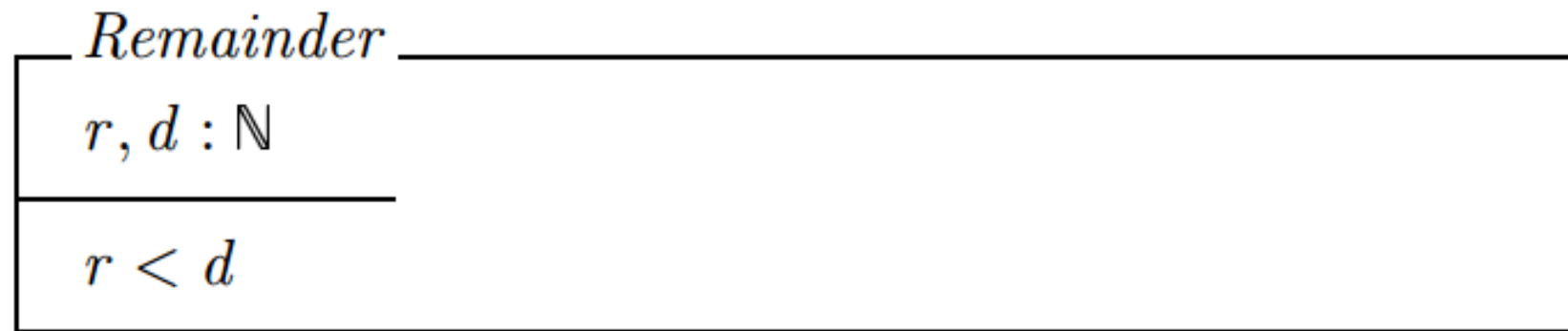
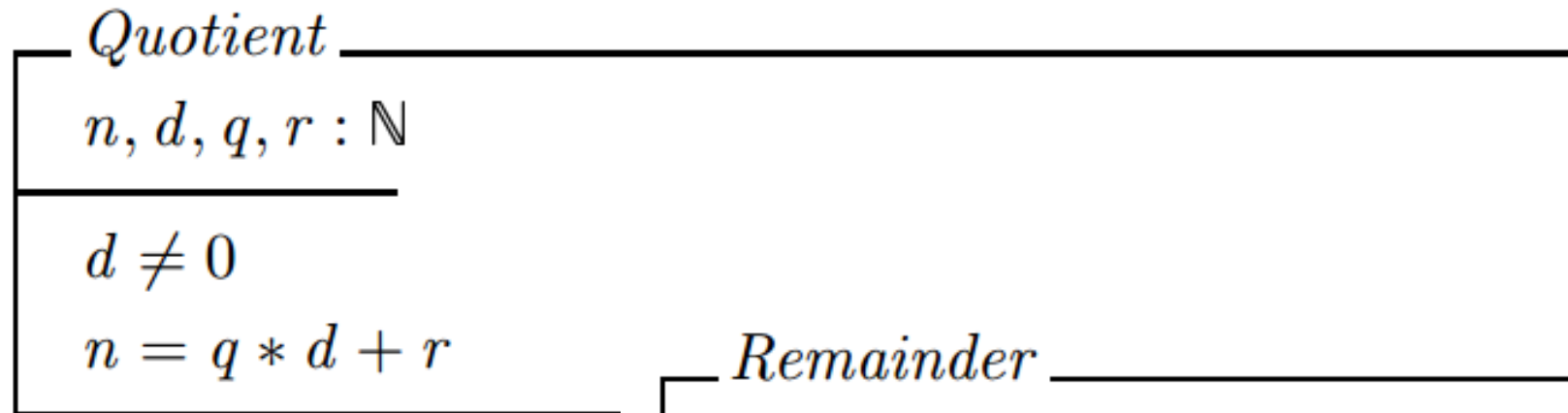
$S \wedge T$ is equivalent to:

a: A
b: B
c: C
$P \wedge Q$

❖ The types for **b** must match, if not $S \wedge T$ will be undefined.

Schema Conjunction (\wedge)

❖ Consider the schemas:



Schema Conjunction (\wedge)

- ❖ To form a schema for Division:

Division \triangleq Quotient \wedge Remainder

<i>Division</i>
$n, d, q, r : \mathbb{N}$
$d \neq 0$
$r < d$
$n = q * d + r$

Schema Disjunction (\vee)

- ❖ If S and T are two schemas then their *disjunction*, $S \vee T$ is also a schema
- ❖ In which the declaration is a *merging* of the **two** declaration *parts* but the *predicate parts* are *disjoined*.
- ❖ Allows us to describe *alternatives* in the behaviour of a system.

Schema Disjunction (\vee)

❖ Example:

S	T
a: A	b: B
b: B	c: C
P	Q

$S \vee T$ is equivalent to:

a: A
b: B
c: C
$P \vee Q$

The types for **b** must match, if not $S \vee T$ will be undefined.

Schema Disjunction (\vee)

- ❖ Example:
Schema for division by zero

$$\begin{array}{l} \textit{DivideByZero} \text{ ---} \\ d, q, r : \mathbb{N} \\ \hline d = 0 \wedge q = 0 \wedge r = 0 \end{array}$$

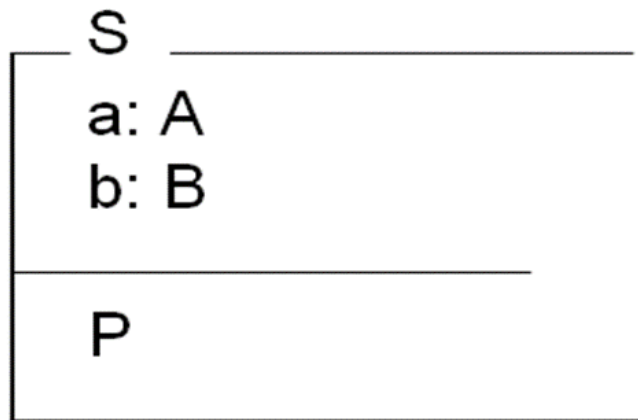
- ❖ The total operation for division is given by:

$$\begin{array}{l} \textit{T_Division} \triangleq \textit{Division} \\ \vee \textit{DivideByZero} \end{array}$$

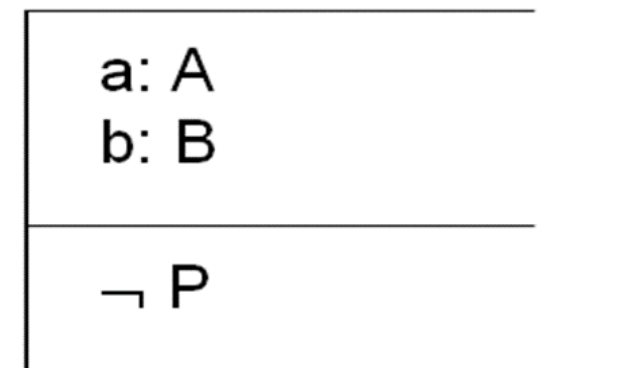
$$\begin{array}{l} \textit{T_Division} \text{ ---} \\ n, d, q, r : \mathbb{N} \\ \hline (d \neq 0 \wedge r < d \wedge n = q * d + r) \vee \\ (d = 0 \wedge q = 0 \wedge r = 0) \end{array}$$

Schema Negation (\neg)

- ❖ If S is a schema then its *negation*, $\neg S$ is also a schema
- ❖ In which the declaration the same as that of S , $\neg S$ may be obtained by negating the predicate part.



$\neg S$ is equivalent to:



Schema Inclusion

- ❖ Reuse the name of one schema in the **declaration part** of another schema
- ❖ When a schema name appears in a **declaration part** of a schema, the result is a **merging of declarations and a conjunction of constraints**.

Schema Inclusion

- ❖ Assume we have two schemas:

BoxOffice

$sold : SEAT \rightarrow CUSTOMER$

$seating : \mathbb{P} SEAT$

$dom\ sold = seating$

Friends

$friends : \mathbb{P} CUSTOMER$

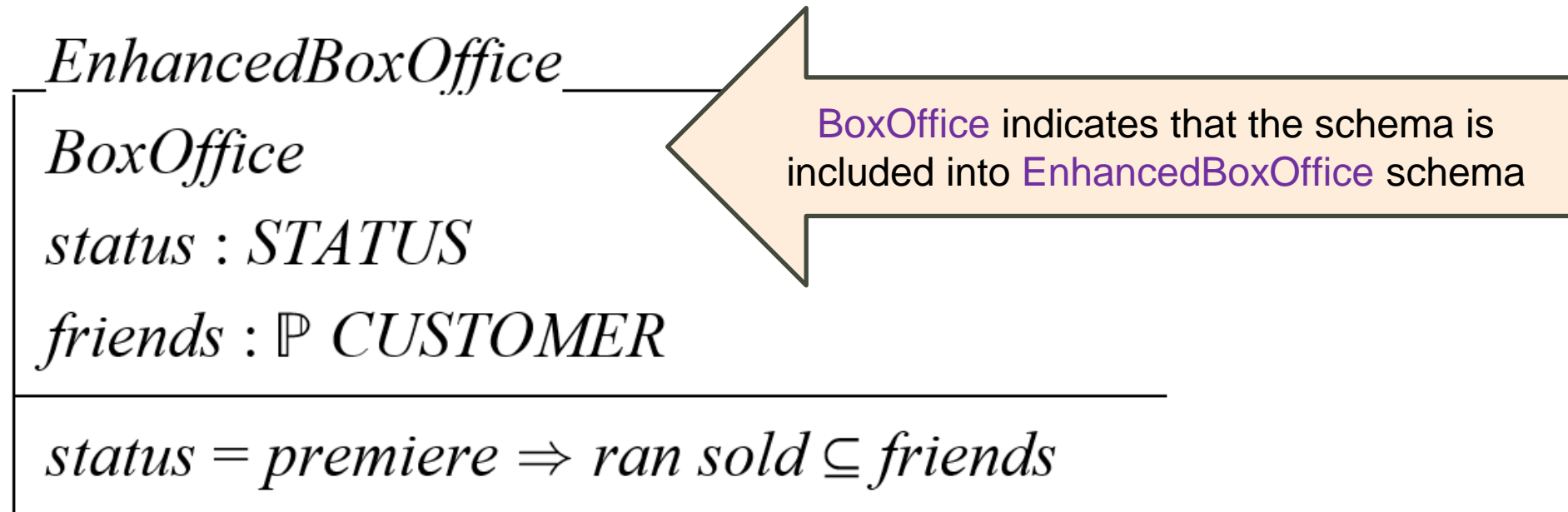
$status : STATUS$

$sold : SEAT \rightarrow CUSTOMER$

$status = premiere \Rightarrow ran\ sold \subseteq friends$

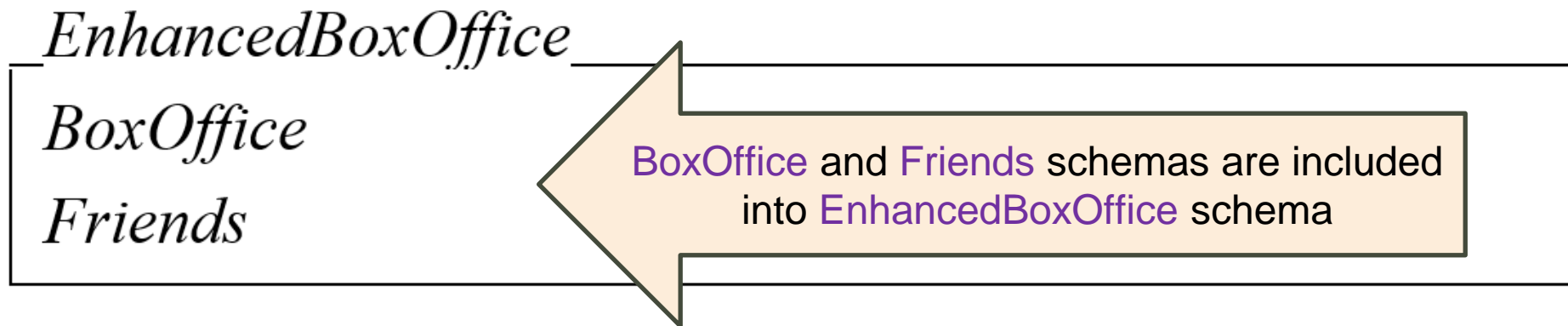
Schema Inclusion

- ❖ EnhancedBoxOffice could have been defined as:



Schema Inclusion

❖ Or even as:



Schema Hiding (\backslash)

- ❖ Used to **hide some variables** and declarations of already known schemas (i.e., existentially quantified).
- ❖ Use \exists to hide variables.

S	
$a : A$	
$b : B$	
P	

Then $S \backslash (a)$ is the schema

$b : B$
$\exists a : A \bullet P$

Schema Hiding (\backslash)

A _____ $a : \mathbb{Z}$ <hr/> $a = 42$	$HideA$ _____ $\exists a : \mathbb{Z} \bullet$ <hr/> $a = 42$	B _____ $a, b : \mathbb{Z}$ <hr/> $a = b + 2$ $b < 10$	$HideB$ _____ $a : \mathbb{Z}$ <hr/> $\exists b : \mathbb{Z} \bullet$ $(a = b + 2 \wedge$ $b < 10)$
---	---	---	---

- ❖ Note that $\exists b : \mathbb{Z} \bullet (a = b + 2 \wedge b < 10)$ means the same thing as $a < 12$.
- ❖ So, we can equivalently write $HideB$ as:

$HideB$ _____ $a : \mathbb{Z}$ <hr/> $a < 12$

Schema Hiding (\backslash)

❖ Example: Define $AddWho \hat{=} AddMember \backslash newMember?$:

$AddMember$ _____ $\Delta ClubState$ $newmember? : STUDENT$ _____ $newmember? \notin badminton$ $badminton' = badminton \cup$ $\{newmember?\}$ $hall' = hall$
--

$AddWho$ _____ $\Delta ClubState$ _____ $\exists newmember? : STUDENT \bullet$ $(newmember? \notin badminton \wedge$ $badminton' = badminton \cup$ $\{newmember?\} \wedge$ $hall' = hall)$

Schema Composition (⋈)

- ❖ If two schemas describe operations upon the same state, then we can construct an operation schema that describes the **effect of one followed by the other**.
- ❖ In a schema composition, the **after state of the first operation is identified** with the **before state of the second**.
- ❖ The symbol for schema composition is **fat semicolon ;**

Schema Composition (\boxtimes)

1. Replace primed variables (') in **first schema** with double primed variables (")
 $S["/"]$
2. Replace un-primed variables in **second schema** with double primed variables (")
 $T['/']$
3. Existentially quantify variables in double primed state ("")
 $\exists \text{ State''} \bullet S["/"] \wedge T["/"]$

Schema Piping (\gg)

- ❖ Constructing a **new schema** out of two old ones by **relating the output variables of one** of those old schemas **to the input variables of the other one**.
- ❖ For the piping $S \gg T$ to be defined, for each word y such that S has an **output $y!$** and T has an **input $y?$** , the **types of these two components** must be the **same**. We call x a **piped variable**.
- ❖ Schema are combined by **schema conjunction**.
- ❖ Any **unmatched inputs and outputs** remain in the signature of the result.

Schema Piping (\gg)

- ❖ Assume we have two schemas S and T :

S	T
$x?, s, s', y!: \mathbf{N}$	$y?, t, t': \mathbf{N}$
$s' = s + x?$	$y? > t$
$y! = 2 * s'$	$t' = t + y?$

- ❖ We rename the same type of variables $y!$ in S and $y?$ in T into an entirely new variable e.g. p , so we form:

$S [p/y!]$

$T [p/y?]$

Schema Piping (>>)

- ❖ Assume $\text{Alpha} \triangleq S [p/y!]$ and $\text{Beta} \triangleq T [p/y?]$

$\text{Alpha} \frac{}{x?, s, s', p: \mathbb{N}}$ <hr/> $s' = s + x?$ $p = 2 * s'$	$\text{Beta} \frac{}{p, t, t': \mathbb{N}}$ <hr/> $p > t$ $t' = t + p$
---	--

- ❖ We form a conjunction of the two renamed schemas as $\text{Gamma} = S [p/y!] \wedge T [p/y?]$

$\text{Gamma} \frac{}{x?, s, s', p, t, t': \mathbb{N}}$ <hr/> $s' = s + x?$ $p = 2 * s'$ $p > t$ $t' = t + p$

Schema Piping (\gg)

- ❖ And hide the variable p in **Gamma**, and named it as Delta: $\Delta = (S[p/y!] \wedge T[p/y?]) \setminus (p)$

and this is $S \gg T$

Simplified version of Delta will look like this



$$\begin{array}{|l} \hline \text{Delta} \\ \hline x?, s, s', t, t': \mathbb{N} \\ \hline \exists p: \mathbb{N} \bullet \\ \quad (s' = s + x? \wedge \\ \quad p = 2 * s' \wedge \\ \quad p > t \wedge \\ \quad t' = t + p) \\ \hline \end{array}$$

$$\begin{array}{|l} \hline \text{Delta} \\ \hline x?, s, s', t, t': \mathbb{N} \\ \hline s' = s + x? \\ 2 * s' > t \\ t' = t + 2 * s' \\ \hline \end{array}$$

Schema Implication (\Rightarrow)

- ❖ The statement “ p implies q ” ($p \Rightarrow q$) means that if p is true, then q must also be true.
- ❖ Statement p is called the **premise** of the implication and q is called the **conclusion**.

Schema Equivalence (\Leftrightarrow)

- ❖ The equivalence “if and only if” ($p \Leftrightarrow q$) means that p and q are of the same strength; thus it might also be called bi-implication.
- ❖ $p \Leftrightarrow q$ means that both $p \Rightarrow q$ and $q \Rightarrow p$.

Schema Propositions

\wedge and \vee associate to the left.

$$P \wedge Q \wedge R \quad \text{is} \quad (P \wedge Q) \wedge R$$

\Rightarrow and \Leftrightarrow associate to the right.

$$P \Rightarrow Q \Rightarrow R \quad \text{is} \quad P \Rightarrow (Q \Rightarrow R)$$

Operators bind in the following precedence: \neg , \wedge , \vee , \Rightarrow , \Leftrightarrow .

$$P \wedge Q \Rightarrow \neg R \Leftrightarrow S \quad \text{is} \quad ((P \wedge Q) \Rightarrow (\neg R)) \Leftrightarrow S$$

Error Scenarios

Error scenarios

- ❖ Errors should be reported whenever one of the **pre-conditions fail** (which means the operation cannot take place)
- ❖ Schemas can be defined for each error condition.
- ❖ The final schema will combine the operator schema and the error schema condition schemas using **disjunction (or) operators**.

The Class System

- ❖ We are going to model a system for certification class system where the scenario will be as:

Students may register to join the class providing the class is not full. The students need to successfully complete all the coursework given to get the certificate of completion.

- ❖ For this example, we are NOT going to concern with details of students such as students' number, name, date of birth etc.

The Class System

- ❖ We introduce the basic type STUDENT as a given set, the set of all students who are going to register for the class:

[STUDENT]

- ❖ There is a limit to the number of students who can join the class. We define *maxClassSize*, the maximum number of students that can be in the class. Assume the size is 20.

$$\textit{maxClassSize} : \mathbb{N}$$
$$\textit{maxClassSize} = 20$$

The Class System

- ❖ We are going to model two sets of students:

enrolled	the set of students who have joined the class
passed	a subset of enrolled and represents the set of enrolled students who have passed their courseworks

- ❖ The size of the class is constrained by *maxClassSize*.
- ❖ The two sets are declared as finite set.

The Class System

- ❖ We define the state space schema called **Class**:

Class

enrolled : $\mathbb{F} \text{STUDENT}$

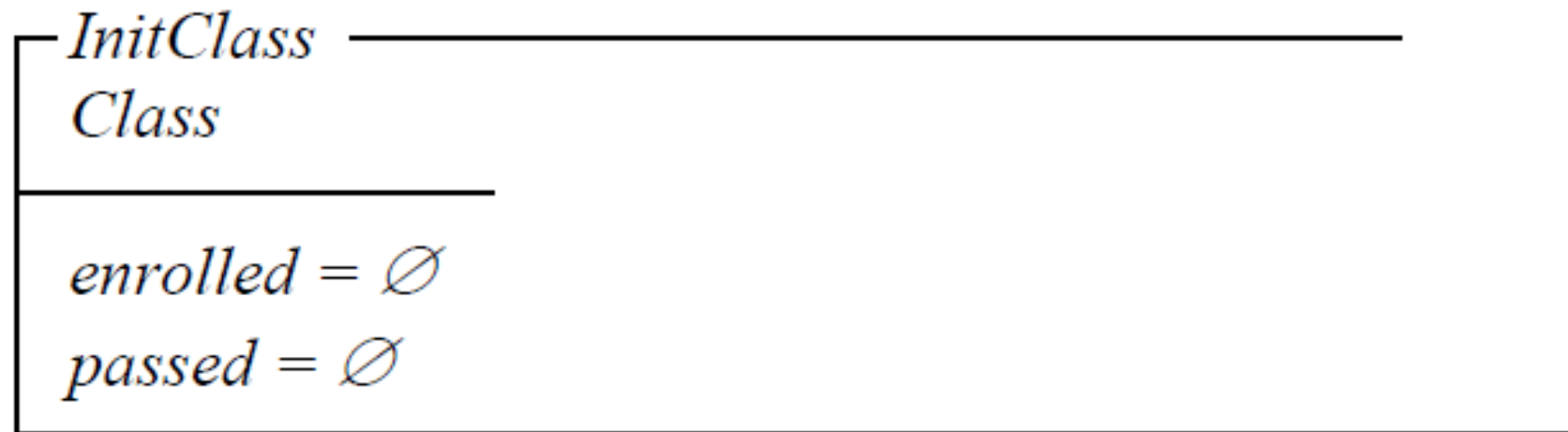
passed : $\mathbb{F} \text{STUDENT}$

$\#enrolled \leqslant \text{maxClassSize}$

$passed \subseteq enrolled$

The Class System

- ❖ To begin with, there are **no** enrolled students and **no** student has passed.



The Class System

- ❖ A student may join the class if the **class is not already full** and if the student has not already enrolled.
- ❖ A new **student cannot passed all** their coursework yet.

EnrolOk —————
ΔClass
student? : STUDENT

#enrolled < maxClassSize
student? ∉ enrolled
enrolled' = enrolled ∪ { student? }
passed' = passed

The Class System

- ❖ An existing student is transferred to the *passed* provided they have passed all their coursework and have not already been transferred.
- ❖ Every student in *passed* must also be in *enrolled*.

CompleteOk

Δ *Class*

student? : STUDENT

student? ∈ enrolled

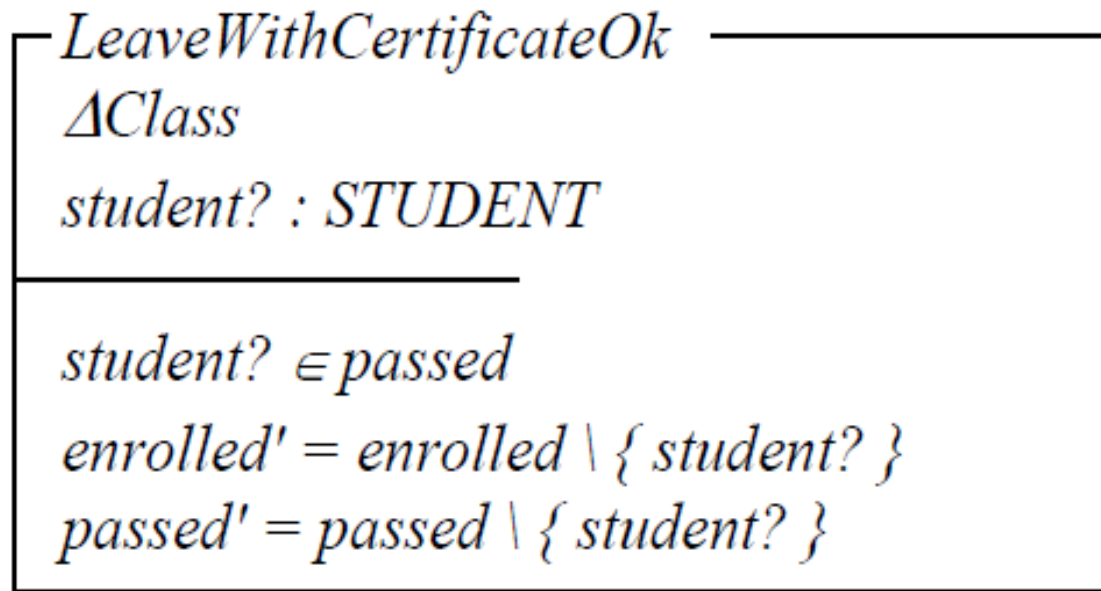
student? ∉ passed

enrolled' = enrolled

passed' = passed ∪ { student? }

The Class System

- ❖ Only those existing students who **have passed** may leave with a **certificate**. Their records will be removed from *enrolled* and *passed*.



The Class System

- ❖ Previous *Class* system only concerned with simple, straightforward, **no problem scenarios**.
 - ❖ We did not concern ourselves with the possibility that the class is full and so no one can enrolled on it.
 - ❖ We ignored the possibility that a student could enrolled twice.
 - ❖ We also ignored the possibility that a student who is not enrolled could transferred to the passed set.
- ❖ We have to address these **error scenarios**.

The Class System

- ❖ Let's draw up a table of pre-conditions, the set of all states for which successful outcomes are defined.

Schema	Pre-conditions for success
EnrolOk	#enrolled < maxClassSize student? ∉ enrolled
CompleteOk	student? ∈ enrolled student? ∉ passed
LeaveWithCertificateOk	student? ∈ passed

The Class System

- ❖ We add on the table the conditions for failure.

Schema	Pre-conditions for failure
EnrolOk	Class is full: $\#enrolled \geq \text{maxClassSize}$ Student already enrolled: $student? \in enrolled$
CompleteOk	Student not enrolled: $student? \notin enrolled$ Student already passed: $student? \in passed$
LeaveWithCertificateOk	Student not passed: $student? \notin passed$

The Class System

1) First step of error scenarios:

Define a *free type definition* to list all the errors for the system including one for the success schema.

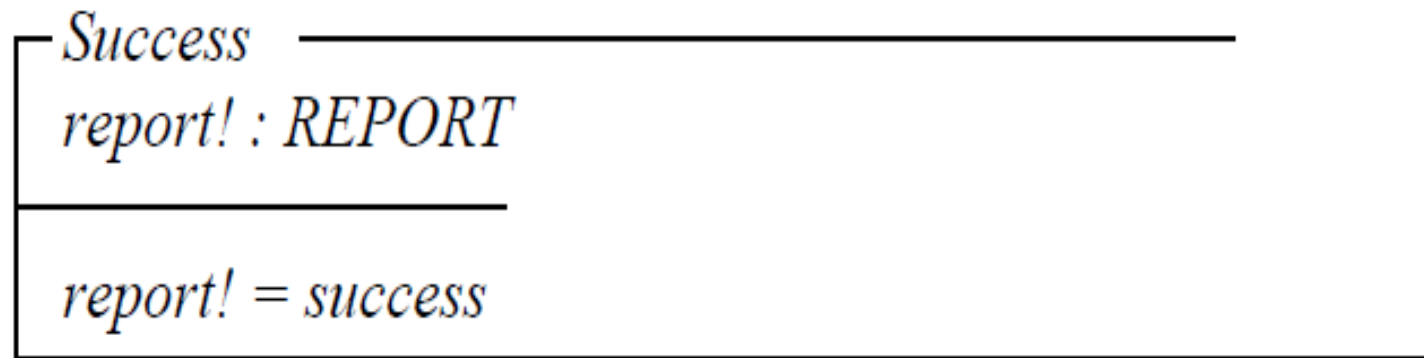
**REPORT ::= success | classFull | alreadyEnrolled | notEnrolled |
alreadyPassed | notPassed**

OR

**REPORT ::= success | class_full | already_enrolled | not_enrolled |
already_passed | not_passed**

The Class System

2) Second step, create the schema for **successful scenario** (e.g **Success**) where we just have one declaration and one predicate.



OR using text form

Success == [report!: REPORT | report! = success]

The Class System

3) Third step is to create **all the schemas for failures** for each identified error case. For an error case, there will be **no update** to any of system variables.

- ❖ The **class is full** if the number of enrolled students has reached the maximum class size.

ClassFull

EClass

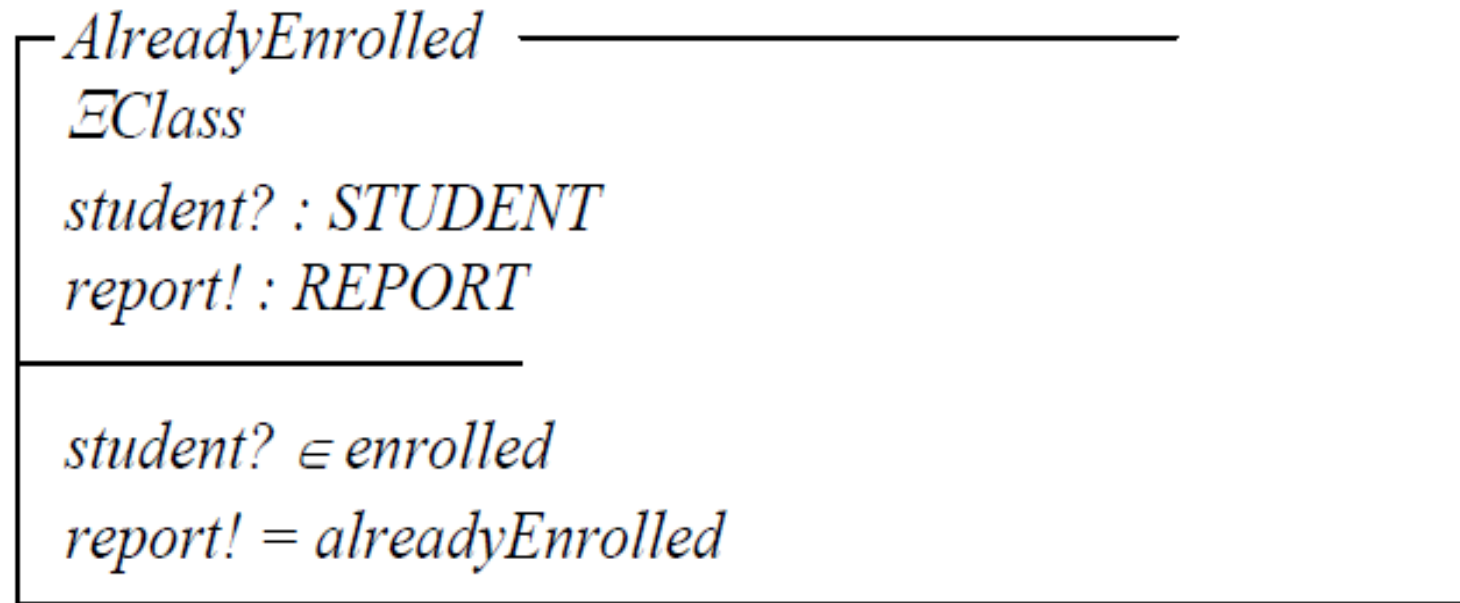
report! : REPORT

#enrolled \geq maxClassSize

report! = classFull

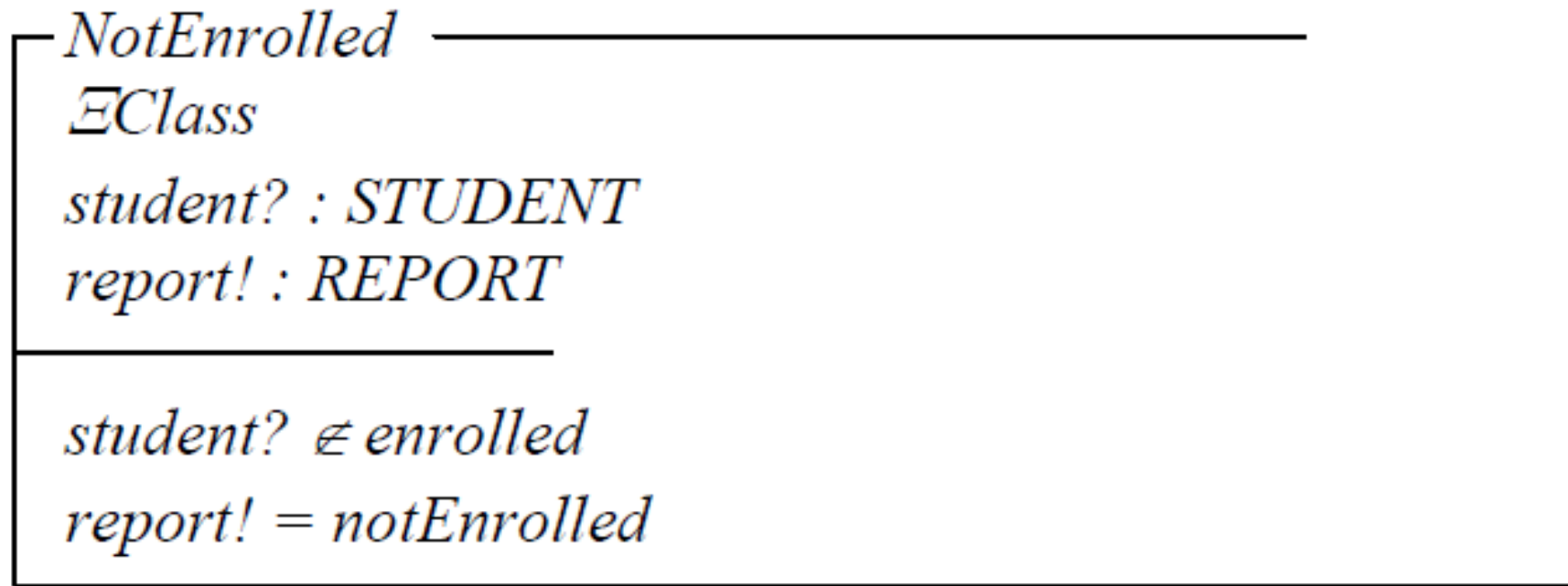
The Class System

- ❖ A student cannot be enrolled again if they are already enrolled.



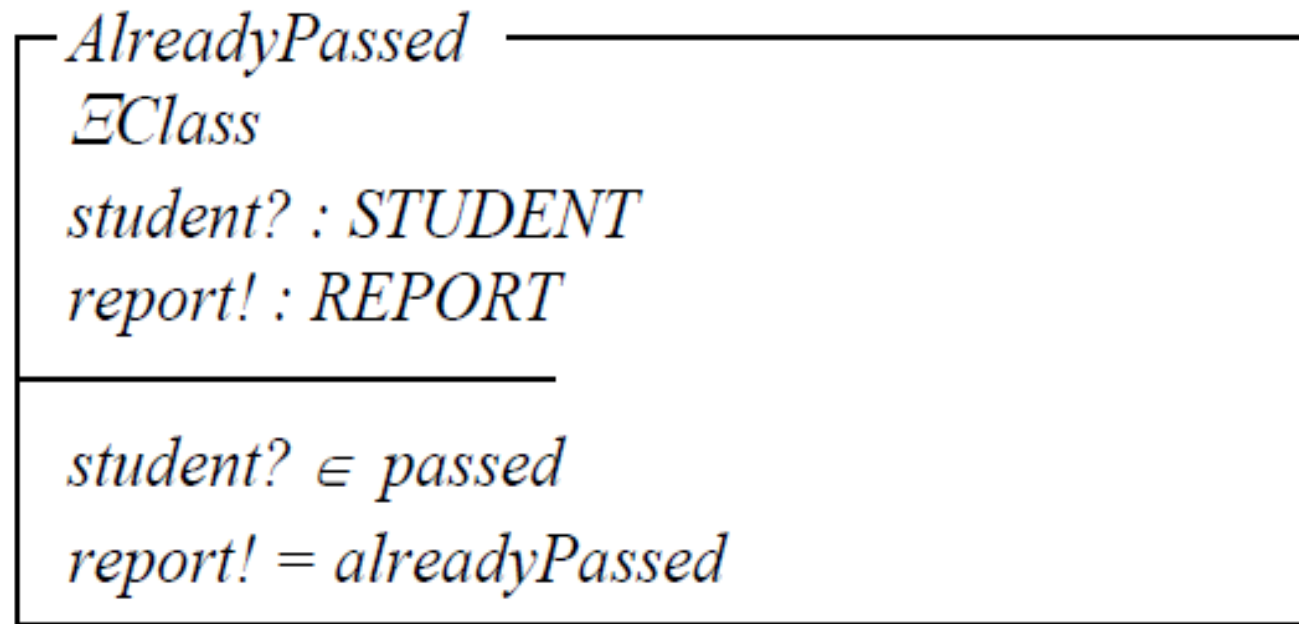
The Class System

- ❖ If a student is not enrolled they cannot pass.



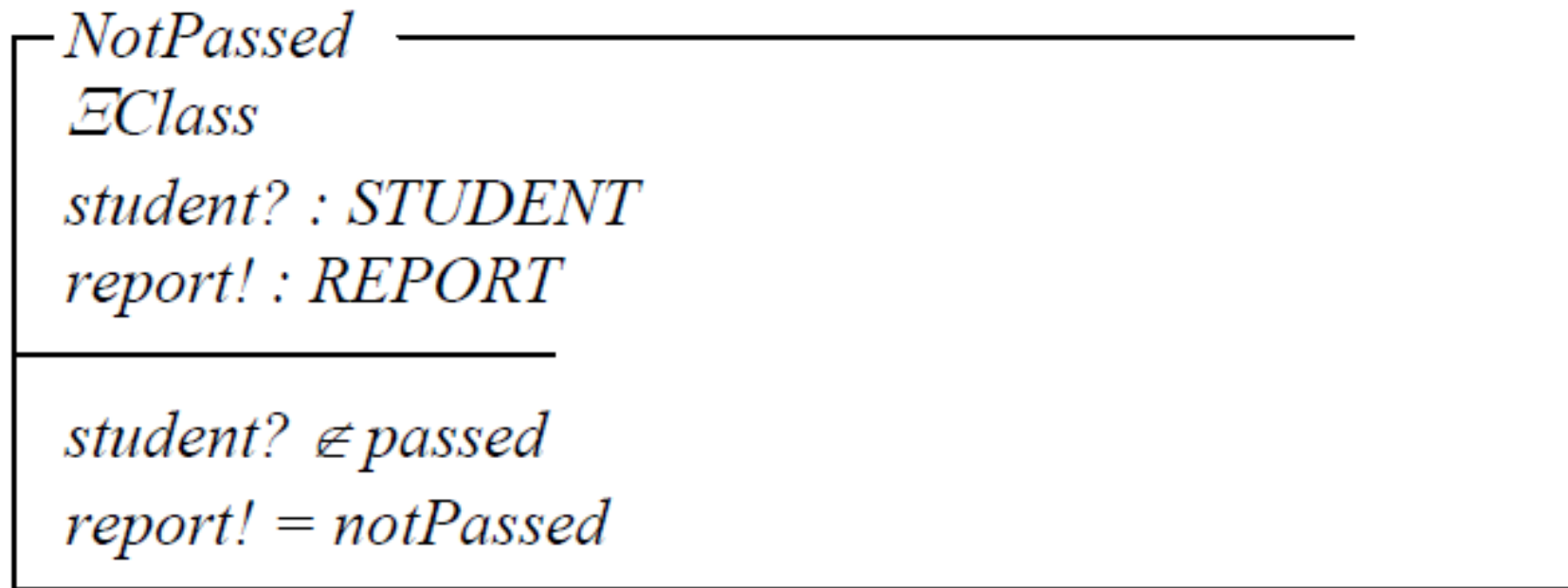
The Class System

- ❖ The same student cannot be passed twice.



The Class System

- ❖ A student who has not passed cannot leave with a certificate.



Complete Schema

Combining Schema

- ❖ When **building large schemas** from smaller ones such as:
 - ❖ Separating normal operations from error handling
 - ❖ Separating access restrictions from functional behaviors
 - ❖ Promoting and framing operations, e.g., reading named a file from reading a file
 - ❖ ...all these need to be combine to create a **complete schema** for those operations.
- ❖ So, **conjunction** and **disjunction** are most useful to **combine schemas**.

Combining Schema

❖ Suppose we have schema A , B , B' :

$$\frac{A}{a : \mathbb{Z}} \\ \hline a = 42$$

$$\frac{B}{a, b : \mathbb{Z}} \\ \hline a = b + 2 \\ b < 10$$

$$\frac{B'}{B : \mathbb{P}\mathbb{Z}} \\ \hline 42 \in B$$

So: $A \text{ and } B \triangleq A \wedge B$

$$\frac{A \text{ and } B}{a, b : \mathbb{Z}} \\ \hline a = 42 \wedge (a = (b + 2) \wedge b < 10)$$

Combining Schema – The Class System

- ❖ We use **conjunction** (and) to combine two schemas (successful operation schema and the success handling schema). Say **EnrolOk** and **Success**

EnrolOk \wedge Success

- ❖ We use **disjunction** (or) to represent alternatives (All the error handling schemas). Say **ClassFull**, **or** **AlreadyEnrolled**.

ClassFull \vee AlreadyEnrolled

Combining Schema – The Class System

- ❖ The combination of success and failure will be by using disjunction:

$$(\text{EnrolOk} \wedge \text{Success}) \vee \text{ClassFull} \vee \text{AlreadyEnrolled}$$

- ❖ To create a complete schema **of the enrol process**.

$$\text{Enrol} \triangleq (\text{EnrolOk} \wedge \text{Success}) \vee \text{ClassFull} \vee \text{AlreadyEnrolled}$$

\triangleq stands for schema definition.

Combining Schema – The Class System

❖ Similarly,

Complete \triangleq (CompleteOk \wedge Success) \vee NotEnrolled \vee AlreadyPassed

LeaveWithCertificate \triangleq (LeaveWithCertificateOk \wedge Success) \vee NotPassed

Complete Schema for Enrol

❖ **Enrol \triangleq (EnrolOk \wedge Success) \vee ClassFull \vee AlreadyEnrolled**

Enrol

enrolled, enrolled' : \mathbb{F} STUDENT

passed, passed' : \mathbb{F} STUDENT

student? : STUDENT

report! : REPORT

$(\#enrolled < maxClassSize \wedge student? \notin enrolled \wedge enrolled' = enrolled \cup \{student?\} \wedge$
 $passed' = passed \wedge report! = success \wedge passed \subseteq enrolled \wedge passed' \subseteq enrolled')$

\vee

$(\#enrolled \geq maxClassSize \wedge report! = classFull \wedge passed \subseteq enrolled \wedge passed' \subseteq enrolled')$

\vee

$(student? \in enrolled \wedge report! = alreadyEnrolled \wedge passed \subseteq enrolled \wedge passed' \subseteq enrolled'$
 $\wedge \#enrolled \leq maxClassSize)$

Complete Schema for Complete

❖ **Complete \triangleq (CompleteOk \wedge Success) \vee NotEnrolled \vee AlreadyPassed**

Complete

$enrolled, enrolled' : \mathbb{F} \text{ STUDENT}$

$passed, passed' : \mathbb{F} \text{ STUDENT}$

$student? : \text{STUDENT}$

$report! : \text{REPORT}$

$(student? \in enrolled \wedge student \notin passed \wedge enrolled' = enrolled \wedge passed' = passed \wedge$
 $report! = success \wedge passed \subseteq enrolled \wedge passed' \subseteq enrolled' \wedge \#enrolled \leqslant maxClassSize)$

\vee

$(student? \notin enrolled \wedge report! = notEnrolled \wedge enrolled' = enrolled \wedge passed' = passed \wedge$
 $passed \subseteq enrolled \wedge passed' \subseteq enrolled' \wedge \#enrolled \leqslant maxClassSize)$

\vee

$(student \in passed \wedge report! = alreadyPassed \wedge enrolled' = enrolled \wedge passed' = passed \wedge$
 $passed \subseteq enrolled \wedge passed' \subseteq enrolled' \wedge \#enrolled \leqslant maxClassSize)$

Complete Schema for LeaveWithCertificate

❖ **LeaveWithCertificate \triangleq (LeaveWithCertificateOk \wedge Success) \vee NotPassed**

LeaveWithCertificate

enrolled, enrolled' : \mathbb{F} STUDENT

passed, passed' : \mathbb{F} STUDENT

student? : STUDENT

report! : REPORT

*(student? \in passed \wedge enrolled' = enrolled \setminus {student?} \wedge passed' = passed \setminus {student?} \wedge
report! = success \wedge passed \subseteq enrolled \wedge passed' \subseteq enrolled' \wedge #enrolled \leq maxClassSize)*

\vee

*(student? \notin passed \wedge report! = notPassed \wedge enrolled' = enrolled \wedge passed' = passed \wedge
passed \subseteq enrolled \wedge passed' \subseteq enrolled' \wedge #enrolled \leq maxClassSize)*

Example

[NAME, HEIGHT, WEIGHT]

HeightAndWeight

known_height: \mathbb{P} NAME

known_weight: \mathbb{P} NAME

height: NAME \rightarrow HEIGHT

weight: NAME \rightarrow WEIGHT

known_weight = dom weight

known_height = dom height

Example

NewHeight

Δ *HeightAndWeight*

name? : *NAME*

hgt? : *HEIGHT*

name? \notin *known_height*

known_height' = *known_height* \cup {*name?*}

height' = *height* \cup {*name?* \mapsto *hgt?*}

weight' = *weight*

known_weight' = *known_weight*

FindWeight

\exists *HeightAndWeight*

name? : *NAME*

wgt! : *WEIGHT*

name? \in *known_weight*

name? \in *dom weight*

wgt! = *weight name?*

known_height' = *known_height*

height' = *height*

Example

WhoIsTallAndHeavy

\exists *HeightAndWeight*

hgt? : *HEIGHT*

wgt? : *WEIGHT*

names! : \mathbb{P} *NAME*

$$\begin{aligned} \text{names!} = & \{n:\text{known_height} \mid \text{height } n = \text{hgt?}\} \cap \\ & \{n:\text{known_weight} \mid \text{weight } n = \text{wgt?}\} \end{aligned}$$

Example

REPORT::= ok | height_already_known | height_not_known |
weight_already_known | weight_not_known

Success

rep! : REPORT

rep! = ok

Example

HeightAlreadyKnown

\exists *HeightAndWeight*

name? : *NAME*

rep! : *REPORT*

$(\text{NewHeight} \wedge \text{Success}) \vee \text{HeightAlreadyKnown}$

name? \in *known_height*

rep! = *height_already_known*

A full definition is:

$\text{FullNewHeight} \triangleq (\text{NewHeight} \wedge \text{Success}) \vee \text{HeightAlreadyKnown}$

Example

FullNewHeight

$known_height, known_height' : \mathbb{P} NAME$

$known_weight, known_weight' : \mathbb{P} NAME$

$height, height' : NAME \rightarrow HEIGHT$

$weight, weight' : NAME \rightarrow WEIGHT$

$name? : NAME$

$hgt? : HEIGHT$

$rep! : REPORT$

$(name? \notin known_height \wedge known_height' = known_height \cup \{name?\} \wedge$
 $height' = height \cup \{name? \mapsto hgt?\} \wedge weight' = weight \wedge known_weight' = known_weight \wedge$
 $known_weight = dom\ weight \wedge known_height = dom\ height \wedge$
 $known_weight' = dom\ weight' \wedge known_height' = dom\ height' \wedge rep! = ok)$
 \vee

$(name? \in known_height \wedge rep! = height_already_known \wedge weight' = weight \wedge$
 $height' = height \wedge known_weight' = known_weight \wedge known_height = dom\ height \wedge$
 $known_weight = dom\ weight \wedge known_height = dom\ height \wedge$
 $known_weight' = dom\ weight' \wedge known_height' = dom\ height')$

Error Scenarios Exercise

Question 1

- ❖ Referring to the scenario from Chapter 2 where we want **to keep the customer information** entering a shop for one whole day.
- ❖ We were given a state space schema called **CountCustomer**, an axiomatic definition for the maximum and a basic type **[CUSTOMER]**

$maximum : \mathbb{N}$
$maximum = 100$

$CountCustomer$
$customer: \mathbb{P} CUSTOMER$
$\#customer \leq maximum$

Question 1

- ❖ And three operations schema called *AddCustomer*, *UpdateCustomer* and *RemoveCustomer* that update the *CountCustomer* system state variable.

AddCustomer

Δ *CountCustomer*

cust? : *CUSTOMER*

cust? \notin *customer*

$\#customer < maximum$

customer' = *customer* \cup {*cust?*}

Question 1

UpdateCustomer

Δ CountCustomer

cust? : CUSTOMER

cust? ∈ customer

customer' = customer ⊕ {cust?}

RemoveCustomer

Δ CountCustomer

cust? : CUSTOMER

cust? ∈ customer

customer' = customer \ {cust?}

Question 1

❖ Lets create a table:

Schema Name	Success Pre-conditions	Failure Pre-conditions	Remark
AddCustomer	cust? \notin customer #customer < maximum	cust? \in customer #customer \geq maximum	Already exist Already full
UpdateCustomer	cust? \in customer	cust? \notin customer	Not exist
RemoveCustomer	cust? \in customer	cust? \notin customer	Not exist

Question 1

- ❖ Introduce a free type **REPORT** that has two errors handling. The error handling reports are *ok*, *alreadyExist*, *notExist*, and *alreadyFull*.

REPORT ::= ok | alreadyExist | notExist | alreadyFull

Question 1

- ❖ Write a schema **Success** that will provide an error report (**rep!**) for every successful error handling schema.

Question 1

- ❖ Write a schema **AlreadyExist** that will provide an error report (**rep!**) for a customer (cust?) who is already enter the shop.

Question 1

- ❖ Write a schema **NotExist** that will provide an error report (**rep!**) for a customer (cust?) who is not in the shop.

Question 1

- ❖ Write a schema **AlreadyFull** that will provide an error report (**rep!**) for a customer (cust?) who cannot enter the shop due to the maximum capacity.

Question 1

- ❖ Complete the whole schema for the adding the customer in the shop as a complete schema/total function called **AddCustomerComplete**.

Question 1

- ❖ Complete the whole schema for the updating the customer in the shop as a complete schema/total function called **UpdateCustomerComplete**.

Question 1

- ❖ Complete the whole schema for removing the customer in the shop as a complete schema/total function called **RemoveCustomerComplete**.

Question 2

- ❖ Referring to the scenario from Chapter 2 where we are recording the passengers aboard an aircraft. There are no seat numbers and passengers are allowed aboard on a first-come-first-served basis

Aircraft

onboard : \mathbb{P} PERSON

#onboard \leq capacity

Question 2

- ❖ We have two operation schemas **BoardAircraft** and **DisembarkAircraft**

BoardAircraft

$\Delta \text{Aircraft}$

$p?: \text{PERSON}$

$p? \notin \text{onboard}$

$\# \text{onboard} < \text{capacity}$

$\text{onboard}' = \text{onboard} \cup \{p?\}$

DisembarkAircraft

$\Delta \text{Aircraft}$

$p?: \text{PERSON}$

$p? \in \text{onboard}$

$\text{onboard}' = \text{onboard} \setminus \{p?\}$

Question 2

❖ Lets create a table:

Schema Name	Success Pre-conditions	Failure Pre-conditions	Remark
BoardAircraft	$p? \notin onboard$ $\#onboard < capacity$	$p? \in onboard$ $\#onboard \geq capacity$	Already onboard Full capacity
DisembarkAircraft	$p? \in onboard$	$p? \notin onboard$	Not onboard

Question 2

- ❖ Introduce a free type **RESPONSE** that has two errors handling. The error handling reports are *success*, *alreadyOnboard*, *notOnboard*, and *alreadyFull*

**RESPONSE ::= success | alreadyOnboard | notOnboard |
alreadyFull**

Question 2

- ❖ Write a schema ***BoardError*** that will provide response *rep!* if it is not possible to board the aircraft.
- ❖ Assume that a successful schema ***Success*** has already been created. Complete the boarding operation as ***Board***

Question 2

- ❖ Write a schema ***DisembarkError*** that will provide response *rep!* if it is not possible to disembark the aircraft.
- ❖ Assume that a successful schema ***Success*** has already been created. Complete the disembarking operation as ***Disembark***.

Summary

- ❖ This lecture described in details the Z schema operators such as conjunction, disjunction, negation, composition and others.
- ❖ It also discussed about the error scenarios to handle failed pre-conditions by discussing the Class System example.
- ❖ At the end of the lecture, the schemas are combined to produce complete schemas using schema calculus.

THANK YOU!!
