Pontus Blume, Mikael Falk Lundgren
Hannes Töyrä, Jonas Örnfelt

Course: Contemporary Software Development
2IS071

Supervisor: Steve McKeever
Date: 07/02/2022

8. Global Post Service

Task 1. GPS Description

This section aims to describe the Global Post Service (GPS) using Z notation.

```
OrderDB
items : Item ℙ
centres : Centre
transportations : Transportation
----
dom centres ⊆ items
dom transportations ⊆ items
```

The *OrderDB* contains the three sets that in turn contain our data. Items are things that need to be transported from A to B, and in order to do that, they need to use transportations in order to pass through one or several centres on their way to their destination.

```
InitOrderDB'
OrderDB'
----
items' = {}
centres' = {}
transportations' = {}
```

InitOrderDB' is a schema that runs on boot. It initializes an empty OrderDB.

```
ΔItemDB
ItemDB
ItemDB′
```

```
ΞItemDB
ΔItemDB
----
items = items'
weights = weights'
dimensions = dimensions'
insuranceAmounts = insuranceAmounts'
destinations = destinations'
finalDeliveryDates = finalDeliveryDates'
```

Specs that are used in operations that change or read the DB. *ΔItemDB* is used when an operation makes changes, and *ΞItemDB* is used in operations that only perform reading actions.

```
Centre
uniqueID : ℕℙ ↦ (type × address)
```

Centre is a warehouse that is identified by its uniqueID, which in turn has a relation to its type and address.

Transportation

scheduleNo : ℕℙ ↦ (method × route)

Transportation is either a plane or a truck that is scheduled to move cargo from one place to another. Transportations are identified by their unique schedule number which in turn has a relation to its method (plane/truck) and its route (e.g. London to Manchester).

Task 2: CRUD

This section contains schemas that describe events that occur when an order comes in, and how orders are subsequently handled. This is achieved through Create, Read, Update and Delete operations (CRUD).

CreateOrder

ΔOrderDB
ΔNewItemNumber
weight? : Weight
dimension? : Dimension
amount? : Amount
destination? : Destination
deliveryDate? : DeliveryDate

newItemNumber ∉ dom (items)
items' = items ∪ {NewItemNumber! ↦ (weight?, dimension?, amount?, destination?, deliveryDate?)}
newItemNumber' = newItemNumber +1

CreateOrder adds a new item to the DB. It takes variables such as weight, dimension, insurance amount, destination and delivery date, and adds them to the set items through a relation to a new item number. Before adding the item, a check is performed to see that the new item number is not already a member of items. Finally, newItemNumber is incremented by one to prepare for the following order.

NewItemNumber

newItemNumber : ℕ

InitNewItemNumber

NewItemNumber

newItemNumber' = 0

NewItemNumber and *InitNewItemNumber* are schemas representing a counter. The number is a natural number that increments by 1 for every new item number. At system initialization, the number is set to 0.

UpdateOrder

$$
\begin{array}{l}
\Delta OrderDB \\
itemNumber? : ItemNumber \\
weight? : Weight \\
dimension? : Dimension \\
amount? : Amount \\
destination? : Destination \\
deliveryDate? : DeliveryDate \\
\hline
items' = items \oplus \{itemNumber? \mapsto (weight?, dimension?, amount?, destination?, \\
deliveryDate?)\}
\end{array}
$$

UpdateOrder works similarly to *CreateOrder,* all variables are set as input so that the updated variables are now stored based on itemNumber and the outdated data is overwritten.

DeleteOrder

$$
\begin{array}{l}
\Delta OrderDB \\
itemNumber? : ItemNumber \\
\hline
itemNumber? \in \mathrm{dom}\ items \\
items' = items \setminus \{itemNumber? \mapsto (weight?, dimension?, amount?, destination?, \\
deliveryDate?)\}
\end{array}
$$

DeleteOrder removes an order based on its itemNumber if the itemNumber exists in the domain items.

Robust functions:

```
Result
result! : REPORT
────────────────
result: OK
```

Result is a schema used for providing a robust function to the schemas. The function is to report back an OK message if nothing else gets triggered.

```
ReadOrder
ΞOrderDB
itemNumber? : ItemNumber
result! : Result
────────────────
itemNumber? ∈ dom items
result! = Success
```

The variable itemNumber in *ReadOrder* refers to a unique identifier for the item itself and we then use this to find a specific item. Since *ReadOrder* does not change the DB, Ξ is used. If the input itemNumber exists in the DB, Success is returned.

```
UnknownOrder
ΞOrderDB
itemNumber? : ItemNumber
result ! : REPORT
────────────────
itemNumber? ∉dom items
result ! = unknown_order
```

UnknownOrder gets triggered if an ItemNumber cannot be found in the domain of items. The schema will report back that it is an unknown order.

Robust implementations:

DoReadOrder ≙ (ReadOrder ∧ Success) ∨ UnknownOrder

DoDeleteOrder ≙ (ReadOrder ∧ Success) ∨ UnknownOrder

DoUpdateOrder ≙ (ReadOrder ∧ Success) ∨ UnknownOrder

Task 3: Adding realism

The following schemas aim to add realism to the design.

Weight

weight == ℝ

x in ℝ| 0 < x <= maxWeight

Dimension

dimension == ℙ (ℝ × ℝ × ℝ)

x,y,z:ℝ | 0 < x,y,z ≤ maxDimension

To make *CreateOrder* more realistic we created two schemas, *Dimension* and *Weight*. The schemas include constraints for order dimension and the weight of the order. The constraints for weight means that the weight must be a rational number larger than 0, but it can't be larger than the max weight. The same constraints apply for the variables in the dimension. The variables are split into three since there must be a length, width and height.

Transportation

scheduleNo : ℕℙ ↦ (method × route × origin × end × distance × priority × weightCap × volumeCap)

The *Transportation* schema includes weightCap and volumeCap for both truck and plane which represents the capacities that this type of transportation has. Route is also split into origin and destination to enable functionality to find optimal routes.

AddTransportation

ΔOrderDB
method? : Method
origin? : Origin
end? : End
distance? : Distance
route? : Route

(method = Plane ∧ weightCap = 40000 ∧ volumeCap = 400 ∧ priority = Express) ∨
(method = Truck ∧ weightCap = 20000 ∧ volumeCap = 200 ∧ priority = Express ∨ Standard)

AddTransportation is used to add new trucks or planes. The two methods of transportations have differing weight and volume capacities. Also, only orders with express priority are allowed to travel on planes, while all orders are allowed to be transported by truck. *ΔOrderDB* is included which signifies a change and update to the database.

PRIORITY :: = Express | Standard **METHOD** :: = Truck | Plane

The Method of transport determines the priority and what constraints are put on weight and dimensions. Priority takes the form of a string, for example the values “standard” or “express”. Different priorities apply to different transportation methods.

```
┌ CheckTransportCaps ──────────────────────────────
│ ΞOrderDB
│ result! : Result
├──────────────────────────────────────────────────
│ cargo = items ⦇{scheduleNo?}⦈
│ (((Σ cargo ↦ (weights)) ≤ weightCap) ∧
│ ((Σ cargo ↦ (dimensions)) ≤ dimensionsCap) ∧
│ result! = Success) ∨
│ result! = Failed
└──────────────────────────────────────────────────
```

CheckTransportCaps calculates whether the total cargo weight and dimensions fit within any given transportation. The schema returns Success if the transportation can hold all the planned cargo and Fail if it cannot.

```
─ ΞOptimalRoute ───────────────────────────────────
procedure GetOptimalRoute();
  var i : INTEGER;
  var optimalRoute: ROUTE;
  begin
      i := 0;
      while i < transportations.Length do i := i + 1;
          if (FindOptimalRoute(transportations[i]) < optimalRoute)
          then
          optimalRoute= FindOptimalRoute(transportations[i])
  end;
optimalRoute!
───────────────────────────────────────────────────
```

The constant function *OptimalRoute* is a proposed solution for finding optimal routes for a set of transportations. The function FindOptimalRoute is not implemented, but it should use some type of algorithm to compare the variables of each transportation and thereby determine the shortest possible route.

CreateOrder
ΔOrderDB
ΔNewItemNumber
ΞOptimalRoute
weight? : Weight
dimension? : Dimension
amount? : Amount
destination? : Destination
deliveryDate? : DeliveryDate

---

newItemNumber ∉ dom (items)
items' = items ∪ {NewItemNumber! ↦ (weight?, dimension?, amount?, destination?, deliveryDate?, OptimalRoute!)}
newItemNumber' = newItemNumber +1

A new *CreateOrder* schema with the OptimalRoute added to it.

Another version of finding the optimal route using pseudocode and Dijkstra's algorithm can be seen below.

ΞOptimalRoute

```
procedure GetOptimalRoute(transportations, source);
for each vertex v in transportations:
        distance[v] := distance[1...]
        previous[v] := undefined
    distance[source] := 0
    T := the set of all nodes in transportations
    while T is not empty:
        u := node in T with smallest distance[ ]
        remove u from T
        for each neighbor v of u:
            alt := distance[u] + distance_between(u, v)
            if alt < distance[v]
                distance[v] := alt
                previous[v] := u
    optimalRoute! = previous[ ]
```