Model-based Solutions for DM Community Challenge "Flight Rebooking"

Decision Management Community Oct-2016 Challenge "Rebooking Passengers from Cancelled Flights" is among the most complex challenges. It is used by authors of several decision modeling books to show the power of different approaches. Here is the problem definition:

A flight was cancelled, and we need to re-book passengers to other flights considering their frequent flyer status, miles, and seat availability. Here is a sample data and flight assignment rules:

Flight	From	To	Dep	Arr	Capacity	Status
UA123	SFO	SNA	1/1/07 6:00 PM	1/1/07 7:00 PM	5	cancelled
UA456	SFO	SNA	1/1/07 7:00 PM	1/1/07 8:00 PM	2	scheduled
UA789	SFO	SNA	1/1/07 9:00 PM	1/1/07 11:00 PM	2	scheduled
UA1001	SFO	SNA	1/1/07 11:00 PM	1/2/07 5:00 AM	0	scheduled
UA1111	SFO	LAX	1/1/07 11:00 PM	1/2/07 5:00 AM	2	scheduled

Name	Status	Miles	Flight
Jenny	gold	500000	UA123
Harry	gold	100000	UA123
Igor	gold	50000	UA123
Dick	silver	100	UA123
Tom	bronze	10	UA123

So far, five submitted solutions used different tools and techniques from Java to DMN boxed expressions. However, they all (including my decision model) were "**method-based**" meaning they specify a certain execution algorithm (method) that could be summarized as in Fig. 1.

This is certainly a "greedy" algorithm (or a heuristic) that will produce a feasible decision, but we would never know if this decision is the optimal one or not. For a larger problem such method-based approach may leave some passengers very upset unless we implement a smarter decision model. The problem with the method-based approach is that it expects a human decision modeler to describe exactly how to do flight assignments while it's humanly impossible to consider all possible variants. In this post I will describe a much more powerful, "model-based" approach to this problem that leads to the optimal solution. You may read more about model-based decisioning here.

1. *First, sort all passengers using their GOLD, SILVER or BRONZE status. If two passengers have the same status use miles as a tiebreaker.*
2. *Repeat for every passenger from the sorted list:*
   - *Build a list of "suitable flights" for the selected passenger. A "suitable" flight should have the same departure and arrival airports as the cancelled flight and it also should still have an available seat*
   - *Sort the flights inside this list by an earlier departure time*
   - *Assign the flight on the top of the list to the current passenger*
   - *Decrement the flight's capacity*

Figure 1. A greedy algorithm for building passenger-flight assignments

The model-based approach is described in Fig. 2:

Given

F set of flights

P set of passengers from the canceled flight

For every passenger $p \in P$ **and flight** $f \in F$ **Determine**

$x_{pf} \in \{0,1\}$ = 1 if passenger *p* is assigned to flight *f* ∈ **F**
= 0 if otherwise

$delay_{pf}$ = number of hours between arrivals of the flight *f* and the passenger *p*'s canceled flight
= 100 if the passenger *p* is assigned to not scheduled flight *f*

$penalty_{pf}$ = $delay_{pf}$* $penaltyPerDelayedHour_p$

Subject to constraints

Each Passenger can be assigned to no more than 1 flight:

$x_{pf1} + x_{pf2} + \ldots + x_{pfn} <= 1$ for each passenger $p \in$ **P**

Number of passengers assigned to the same flight cannot exceed the flight's capacity

$x_{fp1} + x_{fp2} + \ldots + x_{fpn} <= f_{capacity}$ for each flight *f*

Minimize

$$\sum_{p \in \mathbf{P}, f \in \mathbf{F}} (penalty_{pf} * x_{pf})$$

Figure 2. Flight Rebooking: Model-Based Representation

This model for each passenger introduces a new decision variable $penaltyPerDelayedHour_p$ that represents a passenger's penalty for one delayed hour. These penalties could be calculated independently to which flight this passenger is assigned, but the product

$$penalty_{pf} * x_{pf}$$

gives us a penalty for the assignment of the passenger p to the flight f. Thus, our objective is to minimize the total penalties that is presented as a sum of all penalties for all possible combinations p and f.

All passenger-flight assignments are subject to two constraints:

1. Each passenger can be assigned to no more than 1 flight
2. A number of passengers assigned to the same flight cannot exceed the flight capacity.

Below I describe two possible implementations of this model using OpenRules and JSR-331:

- Implementation using Excel and Java
- Implementation using Excel without Java

FIRST IMPLEMENTATION (Excel and Java)

Our decision model will consist of two parts:

1) A pure business part that specifies employee eligibility to all types of vacation days
2) A more technical optimization part that deals with the representations of the constraints and optimization objective.

Business Problem

We will assume that our Business Problem contains:

- Cancelled flight
- Passengers – a set of passengers from the cancelled flight
- Scheduled Flights – a set of flights.

We will add a special "FAKE" flight to the list of "Scheduled Flights", assuming every passenger can be booked to this flight but the penalty for such "fake" booking is huge.

Let's also assume that our Business Problem includes a list of all possible bookings that are combinations "Passenger-Flight" for all Passengers and Flight (including the fake one).

First, for every passenger we can define the decision variable "Passenger Penalty For Delayed Hour" (called $penaltyPerDelayedHour_p$ in the model in Fig.2). These penalties could be calculated independently to which flight this passenger will be assigned using the following decision tables:

DecisionTableIterate CalculatePassengerPenalties	
Array of Objects	**Rules**
Passengers	PassengerPenaltiesDecisionTable

DecisionTableMultiHit CalculatePenalties			
If	If	Conclusion	
Passenger Status	Passenger Miles	Passenger Penalty For Delayed Hour	
		=	10
GOLD		+=	15
SILVER		+=	8
BRONZE		+=	5
	500000+	+=	4
	[250000..500000)	+=	3
	[100000..250000)	+=	2
	[25000..100000)	+=	1

Figure 3. Flight Rebooking: Calculate Passenger Penalty for Delayed Hour

Second, for every possible booking (a combination "Passenger-Flight") we can define the booking properties using the decision table "DecisionTableIterate" in Fig. 4:

DecisionTableIterate DefineBookingProperties	
Array of Objects	Rules
Bookings	DefineBookings

Decision DefineBookings
ActionExecute
Decision Tables
DefineBookingSuitability
DefineBookingDelay
DefineBookingPenalty

Figure 4. Flight Rebooking: Defining Booking Properties

It will define booking suitability, booking delay, and booking penalty using the following decision tables:

DecisionTable DefineBookingSuitability			
If	If	If	Then
Flight Departure Airport	Flight Arrival Airport	Flight Capacity	Booking is Suitable
Passenger Departure Airport	Passenger Arrival Airport	> 0	TRUE
			FALSE

DecisionTable DefineBookingDelay	
If	Then
Flight is Fake	Booking Delay
FALSE	::= Dates.hours($D{Passenger Old Arrival Time}, $D{Flight Arrival Time})
TRUE	1000

DecisionTable DefineBookingPenalty
Then
Booking Penalty
Booking Delay * Passenger Penalty Per Hour

Figure 5. Flight Rebooking: Defining Booking Properties

I put all above tables in the file “FlightRebooking.xls”. To complete this decision model, I added the table “Glossary” to the same file:

Glossary glossary		
Variable	**Business Concept**	**Attribute**
Passengers	**BusinessProblem**	passengers
Flights		flights
Bookings		bookings
Passenger Status	**Passenger**	status
Passenger Miles		miles
Passenger Penalty For Delayed Hour		penaltyForDelayedHour
Passenger Departure Airport	**Booking**	passenger.departureAirport
Passenger Arrival Airport		passenger.arrivalAirport
Passenger Old Arrival Time		passenger.oldArrivalTime
Passenger Penalty Per Hour		passenger.penaltyForDelayedHour
Flight Arrival Time		flight.arrivalTime
Flight Departure Airport		flight.departureAirport
Flight Arrival Airport		flight.arrivalAirport
Flight Capacity		flight.capacity
Flight is Fake		flight.fake
Booking is Suitable		suitable
Booking Delay		delay
Booking Penalty		penalty

Figure 6. Flight Rebooking: Business Glossary

To define the corresponding datatypes I created three Java classes “BusinesProblem”, “Passenger” and “Flight”. They are simple Java beans (a data structure) that include attributes from the glossary plus several more attributes. Here they are:

BusinessProblem:

```
Flight              cancelledFlight;
Flight []           flights;
Passenger []        passengers;
ArrayList<Booking>  bookings;
```

Flight:

```
String      number;
String      departureAirport;
String      arrivalAirport;
```

```
    Date          departureTime;
    Date          arrivaTime;
    int           capacity;
    boolean       fake;
```

Passenger:

```
    String        name;
    String        status;
    int           miles;
    String        departureAirport;
    String        arrivalAirport;
    Flight        cancelledFlight;
    Date          oldArrivalTime;
    String        newFlight;
    int           penaltyForDelayedHour;
```

To test our decision model, I defined two Data tables in the file “Test.xls”:

Data Flight flights						
number	departureAirport	arrivalAirport	departureTime	arrivalTime	capacity	
Flight Number	**Flight Departure Airport**	**Flight Arrival Airport**	**Flight Departure Time**	**Flight Arrival Time**	**Flight Capacity**	
UA123	SFO	SNA	1/1/17 6:00 PM	1/1/17 7:00 PM	5	cancelled
UA456	SFO	SNA	1/1/17 7:00 PM	1/1/17 8:00 PM	2	scheduled
UA789	SFO	SNA	1/1/17 9:00 PM	1/1/17 11:00 PM	2	scheduled
UA1001	SFO	SNA	1/1/17 11:00 PM	1/2/17 5:00 AM	0	scheduled
UA1111	SFO	LAX	1/1/17 11:00 PM	1/2/17 5:00 AM	2	scheduled

Data Passenger passengers		
name	status	miles
Name	**Status**	**Miles**
Tom	BRONZE	10000
Igor	GOLD	50000
Jenny	GOLD	500000
Harry	GOLD	100000
Dick	SILVER	2500

Figure 7. Flight Rebooking: Test Data

The first table contains all flights starting with the cancelled one. The second table contains a list of passengers to be assigned to a scheduled flight (if possible).

I used the following Java launcher to run this decision model:

```java
public static void main(String[] args) {

    String fileName = "file:rules/Test.xls";
    String decisionName = "DefineBusinessProblem";
    Decision decision = new Decision(decisionName,fileName);
    decision.put("FEEL", "On");
    decision.put("trace", "Off");

    Flight[] flights = (Flight[])decision.execute("getFlights");
    Passenger[] passengers = (Passenger[])decision.execute("getPassengers");

    BusinessProblem businessProblem = new BusinessProblem(flights,passengers);
    decision.put("BusinessProblem", businessProblem);
    decision.execute();

    decision.log("==== Rebooked Passengers");
    for(Passenger p : businessProblem.passengers) {
        decision.log(p);
    }

    decision.log("==== Possible Bookings");
    for(Booking b : businessProblem.bookings) {
        decision.log(b);
    }
}
```

Figure 8. Flight Rebooking: Java Launcher for Business Problem

When I ran this model, it produced the following results:

```
==== Rebooked Passengers
Passenger [name=Tom, status=BRONZE, miles=10000, departureAirport=SFO,
          arrivalAirport=SNA, penaltyForDelayedHour=15, newFlight=?]
Passenger [name=Igor, status=GOLD, miles=50000, departureAirport=SFO,
          arrivalAirport=SNA, penaltyForDelayedHour=26, newFlight=?]
Passenger [name=Jenny, status=GOLD, miles=500000, departureAirport=SFO,
          arrivalAirport=SNA, penaltyForDelayedHour=25, newFlight=?]
Passenger [name=Harry, status=GOLD, miles=100000, departureAirport=SFO,
          arrivalAirport=SNA, penaltyForDelayedHour=27, newFlight=?]
Passenger [name=Dick, status=SILVER, miles=2500, departureAirport=SFO,
          arrivalAirport=SNA, penaltyForDelayedHour=18, newFlight=?]
==== Possible Bookings
Assignment: Tom=>FAKE, suitable=true, delay=1000, penalty=15000
Assignment: Tom=>UA456, suitable=true, delay=1, penalty=15
Assignment: Tom=>UA789, suitable=true, delay=4, penalty=60
Assignment: Tom=>UA1001, suitable=false, delay=22, penalty=330
Assignment: Tom=>UA1111, suitable=false, delay=22, penalty=330
Assignment: Igor=>FAKE, suitable=true, delay=1000, penalty=26000
Assignment: Igor=>UA456, suitable=true, delay=1, penalty=26
Assignment: Igor=>UA789, suitable=true, delay=4, penalty=104
Assignment: Igor=>UA1001, suitable=false, delay=22, penalty=572
Assignment: Igor=>UA1111, suitable=false, delay=22, penalty=572
Assignment: Jenny=>FAKE, suitable=true, delay=1000, penalty=25000
Assignment: Jenny=>UA456, suitable=true, delay=1, penalty=25
```

```
Assignment: Jenny=>UA789, suitable=true, delay=4, penalty=100
Assignment: Jenny=>UA1001, suitable=false, delay=22, penalty=550
Assignment: Jenny=>UA1111, suitable=false, delay=22, penalty=550
Assignment: Harry=>FAKE, suitable=true, delay=1000, penalty=27000
Assignment: Harry=>UA456, suitable=true, delay=1, penalty=27
Assignment: Harry=>UA789, suitable=true, delay=4, penalty=108
Assignment: Harry=>UA1001, suitable=false, delay=22, penalty=594
Assignment: Harry=>UA1111, suitable=false, delay=22, penalty=594
Assignment: Dick=>FAKE, suitable=true, delay=1000, penalty=18000
Assignment: Dick=>UA456, suitable=true, delay=1, penalty=18
Assignment: Dick=>UA789, suitable=true, delay=4, penalty=72
Assignment: Dick=>UA1001, suitable=false, delay=22, penalty=396
Assignment: Dick=>UA1111, suitable=false, delay=22, penalty=396
```

As you can see, so far, our decision model only calculated attribute "`penaltyForDelayedHour`" for every passenger (but not "newFlight") and evaluated attribute suitable, delay, and penalty for every possible booking.

To define actual passenger-flight assignments (bookings) we need to implement the second part of our decision model.

2. Optimization Problem

Now we will implement the second (optimization) part. First we will introduce unknowns x_{pf} and $penalty_{pf}$ defined in Fig. 2, then we will define an optimization objective $\sum_{p\in \mathbf{P}, f\in \mathbf{F}}(penalty_{pf} * x_{pf})$

Which should be minimized. The first major constraint "Each Passenger can be assigned to no more than 1 flight" now, when we added a FAKE-flight, can be expressed as

$$x_{pf1} + x_{pf2} + \ldots + x_{pfn} = 1 \qquad \text{for each passenger } p \in \mathbf{P}$$

The second constraint "Number of passengers assigned to the same flight cannot exceed the flight's capacity" remains the same:

$$x_{fp1} + x_{fp2} + \ldots + x_{fpn} <= f_{capacity} \quad \text{for each flight } f$$

In my initial implementation I decided to do it directly in Java using JSR-331 (a constraint programming standard). First, I added the Java class "Booking" with the following attributes:

```
BusinessProblem     problem;
Passenger           passenger;
Flight              flight;
boolean             suitable;
int                 delay;
int                 penalty;
Var                 var;
```

`If flight is "FAKE" it uses MAX_DELAY_HOURS = 1000 to define the delay.`

Then I defined a Java class "Optimization" as a subclass of the standard class "OptimizationProblem" – see Fig. 9.

```java
public class Optimization extends OptimizationProblem {

    BusinessProblem businessProblem;

    public Optimization(BusinessProblem businessProblem) {
        this.businessProblem = businessProblem;
    }

    public void define() {
        // Define assignment variables
        ArrayList<Booking> bookings = businessProblem.getBookings();
        Var[] penalties = new Var[bookings.size()];
        int n = 0;
        for (Booking booking : bookings) {
            Var x = csp.variable(booking.getName(), 0, 1);
            booking.setVar(x);
            if (!booking.isSuitable())
                csp.post(x, "=", 0);
            penalties[n] = x.multiply(booking.getPenalty());
            n++;
        }
        // Define optimization objective
        Var objective = csp.sum(penalties);
        csp.add("TotalPenalty", objective);
        setObjective(objective);

        // Constraint "Assign passenger to one and only one flight"
        for (Passenger passenger : businessProblem.getPassengers()) {
            ArrayList<Var> passengerBookingVars = new ArrayList<Var>();
            for (Booking booking : bookings) {
                if (passenger.equals(booking.getPassenger()))
                    passengerBookingVars.add(booking.getVar());
            }
            Var sumPassengerBookingVars = csp.sum(passengerBookingVars);
            csp.post(sumPassengerBookingVars, "=", 1);
        }
        // Capacity constraints
        for (Flight flight : businessProblem.getFlights()) {
            ArrayList<Var> flightBookingVars = new ArrayList<Var>();
            for (Booking booking : bookings) {
                if (flight.equals(booking.getFlight()))
                    flightBookingVars.add(booking.getVar());
            }
            Var sumFlightBookingVars = csp.sum(flightBookingVars);
            csp.post(sumFlightBookingVars, "<=", flight.getCapacity());
        }
    }

    public void saveSolution(Solution solution) {
        solution.log(5);
        ArrayList<Booking> bookings = businessProblem.getBookings();
        for (Booking booking : bookings) {
            String name = booking.getName();
            if (solution.getValue(name) > 0)
                booking.getPassenger().setNewFlight(booking.getFlight().getNumber());
        }
    }
}
```

Figure 9. Flight Rebooking: Optimization Problem in Java with JSR-331

The method “*define()*” first defines a constrained variables “x” for every possible passenger-flight assignment (booking):

```
Var x = csp.variable(booking.getName(), 0, 1);
```

This variable could take only values 0 or 1. The name of each variable ”x” is composed as

passenger.name + “=>” + flight.number

If the booking is not suitable, we make this variable to be equal to 0:

```
csp.post(x, "=", 0);
```

Then we define a booking’s penalty:

```
penalties[n] = x.multiply(booking.getPenalty());
```

The optimization objective “TotalPenalty” is defined as a sum of all booking penalties:

```
Var objective = csp.sum(penalties);
```

To define the constraint “Each Passenger can be assigned to no more than 1 flight”, we create an array “passengerBookingVars”, that contains all booking variables “x” related to the selected passenger, and then post the constraint

```
csp.post(sumPassengerBookingVars, "=", 1);
```

Here we used the operator “=” instead of “<=” because our list of flights contains the FAKE-flight, to which any passenger can be assigned (meaning this passenger will have no booking).

To define the constraint “Number of passengers assigned to the same flight cannot exceed the flight’s capacity”, we create an array “flightBookingVars”, that contains all booking variables “x” related to the selected flight, and then post the constraint

```
csp.post(sumFlightBookingVars, "<=", flight.getCapacity());
```

This completes the problem definition. As we want to minimize the “TotalPenalty” we may rely on the default method “*solve()*”. We only need to define the method “saveSolution()” that will be called when the optimal solution is found to setup business attribute “newFlight” for every passenger.

And finally we should add these 3 lines to our Java launcher in Fig.8 before printing Rebooked Passengers:

```
Optimization optimization = new Optimization(businessProblem);
optimization.define();
optimization.solve();
```

Here is the optimal solution produced by our decision model:

```
=== SOLVE:
Found a solution with TotalPenalty[18216]. Fri Dec 14 17:27:10 EST 2018
Found a solution with TotalPenalty[18213]. Fri Dec 14 17:27:10 EST 2018
Found a solution with TotalPenalty[15249]. Fri Dec 14 17:27:10 EST 2018
Found a solution with TotalPenalty[15228]. Fri Dec 14 17:27:10 EST 2018
Found a solution with TotalPenalty[15225]. Fri Dec 14 17:27:10 EST 2018
```

```
Solution #1:
Tom=>FAKE[1] Tom=>UA456[0] Tom=>UA789[0] Tom=>UA1001[0] Tom=>UA1111[0]
Igor=>FAKE[0] Igor=>UA456[1] Igor=>UA789[0] Igor=>UA1001[0] Igor=>UA1111[0]
Jenny=>FAKE[0] Jenny=>UA456[0] Jenny=>UA789[1] Jenny=>UA1001[0] Jenny=>UA1111[0]
Harry=>FAKE[0] Harry=>UA456[1] Harry=>UA789[0] Harry=>UA1001[0] Harry=>UA1111[0]
Dick=>FAKE[0] Dick=>UA456[0] Dick=>UA789[1] Dick=>UA1001[0] Dick=>UA1111[0]
TotalPenalty[15225]

==== Rebooked Passengers
Passenger [name=Tom, status=BRONZE, miles=10000, departureAirport=SFO,
          arrivalAirport=SNA, penaltyForDelayedHour=15, newFlight=FAKE]
Passenger [name=Igor, status=GOLD, miles=50000, departureAirport=SFO,
          arrivalAirport=SNA, penaltyForDelayedHour=26, newFlight=UA456]
Passenger [name=Jenny, status=GOLD, miles=500000, departureAirport=SFO,
          arrivalAirport=SNA, penaltyForDelayedHour=25, newFlight=UA789]
Passenger [name=Harry, status=GOLD, miles=100000, departureAirport=SFO,
          arrivalAirport=SNA, penaltyForDelayedHour=27, newFlight=UA456]
Passenger [name=Dick, status=SILVER, miles=2500, departureAirport=SFO,
          arrivalAirport=SNA, penaltyForDelayedHour=18, newFlight=UA789]
```

As you can see, the **optimal solution** with `TotalPenalty=15225` was found after 4 previous solutions with larger total penalties were determined. The optimal Solution shows the automatically selected values (0 or 1) all our variables "x". The newFlight for the passenger Tom is "FAKE" meaning our decision model was not able to find a flight for Tom, while all other passengers were assigned to the scheduled flights in accordance with the problem requirements.

The execution time for finding an optimal solution for this small problem was only 17 milliseconds. It's important to notice that the same model will handle much larger numbers of passengers and flights and can be reused in real-world systems.

This completes my first implementation. While this implementation demonstrates the power of the model-based approach to decision modeling, it used Java code for the optimization part. In the next section I will try to move this code to more user-friendly decision tables in Excel with minimal Java involvement.

SECOND IMPLEMENTATION (Excel without Java)

In my second implementation I tried to move Java code to Excel-based tables. I consider this as work in progress, and I am far from being satisfied with what I got so far. However, I've managed to move the optimization piece from Java to Excel, and it works now producing the same optimal results. So, I decided to share this preliminary representation of the same model hoping to get a constructive feedback from the readers. The generic (problem-independent) implementation logic is now hidden inside a special decision table template "DecisionTableCSPTemplate". So, here are my Excel-based tables (sorry, without comments as I hope they are self-explanatory).

The list of major sub-decisions:

Decision RebookPassengers
ActionExecute
Decision Tables
CalculatePassengerPenalties
DefineBookingProperties
DefineConstraints1
DefineConstraints2
DefineObjective
MinimizeTotalPenalty
SaveSolution

The input data comes through an instance of the decision object "BusinessProblem":

DecisionObject decisionObjects	
Business Concept	**Business Object**
BusinessProblem	:= decision.get("BusinessProblem")

All used decision variables are defined in the following glossary:

Glossary glossary		
Variable	**Business Concept**	**Attribute**
Passengers	**BusinessProblem**	passengers
Flights		flights
Bookings		bookings
Total Penalty		totalPenalty
Flight Number	**Flight**	number
Flight Capacity		capacity
Passenger Miles		miles
Passenger Penalty For Delayed Hour		penaltyForDelayedHour
Passenger Booking Variables		bookingVariables
Passenger Booking Penalties		bookingPenaltyVariables
Passenger Name	**Passenger**	name
Passenger Status		status
Passenger Miles		miles
Passenger Penalty For Delayed Hour		penaltyForDelayedHour
Passenger Booking Variables		bookingVariables
Passenger Booking Penalties		bookingPenaltyVariables
Booking Name	**Booking**	name
Booking Flight Number		flight.number
Booking Passenger Name		passenger.name
Passenger Departure Airport		passenger.departureAirport
Passenger Arrival Airport		passenger.arrivalAirport
Passenger Old Arrival Time		passenger.oldArrivalTime
Passenger Penalty Per Hour		passenger.penaltyForDelayedHour
Flight Arrival Time		flight.arrivalTime
Flight Departure Airport		flight.departureAirport
Flight Arrival Airport		flight.arrivalAirport
Flight Capacity		flight.capacity
Flight is Fake		flight.fake
Booking is Suitable		suitable
Booking Delay		delay
Booking Penalty		penalty

DecisionTableIterate CalculatePassengerPenalties	
Array of Objects	**Rules**
Passengers	PassengerPenaltiesDecisionTable

DecisionTableMultiHit PassengerPenaltiesDecisionTable			
If	If	Conclusion	
Passenger Status	**Passenger Miles**	**Passenger Penalty For Delayed Hour**	
		=	10
GOLD		+=	15
SILVER		+=	8
BRONZE		+=	5
	500000+	+=	4
	[250000..500000)	+=	3
	[100000..250000)	+=	2
	[25000..100000)	+=	1

DecisionTableIterate DefineBookingProperties	
Array of Objects	**Rules**
Bookings	DefineBookings

Decision DefineBookings
ActionExecute
Decision Tables
DefineBookingSuitability
DefineBookingDelay
DefineBookingPenalty
DefineBookingVariable
ExcludeNonSuitableBookings
DefineBookingPenaltyVariable
AddVariablesToLists

Booking Business Rules:

DecisionTable DefineBookingSuitability			
If	If	If	Then
Flight Departure Airport	**Flight Arrival Airport**	**Flight Capacity**	**Booking is Suitable**
Passenger Departure Airport	Passenger Arrival Airport	> 0	TRUE
			FALSE

DecisionTable DefineBookingDelay		
If	Then	
Flight is Fake	**Booking Delay**	
FALSE	::= Dates.hours($D{Passenger Old Arrival Time}, $D{Flight Arrival Time})	
TRUE	1000	a huge penalty for FAKE-booking

DecisionTable DefineBookingPenalty
Then
Booking Penalty
Booking Delay * Passenger Penalty Per Hour

Booking Optimization Constraints:

DecisionTableCSP DefineBookingVariable		
ActionCreateVar		
Var Name	Min	Max
::= bookingName(decision)	0	1

DecisionTableCSP ExcludeNonSuitableBookings			
If	ActionXoperY		
Booking is Suitable	Variable	oper	Value
FALSE	:= bookingName(decision)	=	0

DecisionTableCSP DefineBookingPenaltyVariable			
ActionCreateExpression			
Expression Name	Variable	oper	Value
:= bookingPenaltyName(decision)	:= bookingName(decision)	*	:= $I{Booking Penalty}

DecisionTableCSP AddVariablesToLists	
ActionAddVarToList	
List Name	Var Name
All Booking Variables	::= bookingName(decision)
All Booking Penalty Variables	:= bookingPenaltyName(decision)

Passenger Booking Constraints:

DecisionTableIterate DefineConstraints1	
Array of Objects	**Rules**
Passengers	MethodsForConstraint1

Decision MethodsForConstraint1
ActionExecute
Decision Tables
SelectPassengerBookings
DefineSumOfPassengerBookings
MakeSumOfPassengerBookingsEqualTo1

DecisionTableIterate SelectPassengerBookings	
Array of Objects	**Rules**
Bookings	AddBookingVariableToPassengerVarList

DecisionTableCSP AddBookingVariableToPassengerVarList		
If	ActionAddVarToList	
Passenger Name	List Name	Var Name
Booking Passenger Name	:= passengerVarList(decision)	:= bookingName(decision)

DecisionTableCSP DefineSumOfPassengerBookings	
ActionSum	
Sum	Variables
:= sumOfPassengerBookings(decision)	:= passengerVarList(decision)

DecisionTableCSP MakeSumOfPassengerBookingsEqualTo1		
ActionXoperY		
Variable	oper	Value
:= sumOfPassengerBookings(decision)	=	1

Flight Booking Constraints:

DecisionTableIterate DefineConstraints2	
Array of Objects	**Rules**
Flights	MethodsForConstraint2

Decision MethodsForConstraint2
ActionExecute
Decision Tables
SelectFlightBookings
DefineSumOfFlightBookings
LimitSumOfFlightBookingsByItsCapacity

DecisionTableIterate SelectFlightBookings	
Array of Objects	**Rules**
Bookings	AddBookingVariableToFlightVarList

DecisionTableCSP AddBookingVariableToFlightVarList		
If	ActionAddVarToList	
Flight Number	List Name	Var Name
Booking Flight Number	:= flightVarList(decision)	:= bookingName(decision)

DecisionTableCSP DefineSumOfFlightBookings	
ActionSum	
Sum	Variables
:= sumOfFlightBookings(decision)	:= flightVarList(decision)

DecisionTableCSP LimitSumOfFlightBookingsByItsCapacity		
ActionXoperY		
Variable	oper	Value
:= sumOfFlightBookings(decision)	<=	Flight Capacity

The above tables utilized the following convenience methods defined in Excel:

Method String bookingName(Decision decision)
return ${Booking Passenger Name} + "=>" + ${Booking Flight Number};

Method String bookingPenaltyName(Decision decision)
return bookingName(decision) + "-penalty";

Method String passengerVarList(Decision decision)
return ${Passenger Name} + "VarList";

Method String sumOfPassengerBookings(Decision decision)
return "SumOf"+${Passenger Name} + "Bookings";

Method String flightVarList(Decision decision)
return ${Flight Number} + "VarList";

Method String sumOfFlightBookings(Decision decision)
return "SumOf"+${Flight Number} + "Bookings";

The objective is defined as a sum of All Booking Penalty Variables:

DecisionTableCSP DefineObjective	
ActionSum	
Sum	Variables
Total Penalty	All Booking Penalty Variables

This table is used to minimize the objective

DecisionTableCSP MinimizeTotalPenalty	
ActionOptimize	
Optimization type	**Objective**
Minimize	Total Penalty

and this table to save the found optimal solution:

```
Method void SaveSolution(Decision decision)
Solution solution = csp(decision).getSolution();
ArrayList bookings = $O{BusinessProblem}.getBookings();
for (int i = 0; i< bookings.size(); i++) {
  Booking booking = bookings.get(i);
  String name = booking.getName();
  if (solution.getValue(name) > 0)
     booking.getPassenger().setNewFlight(booking.getFlight().getNumber());
}
```

This implementation produces the same solution as the first one.

Conclusion

Both implementations of our decision model have the following advantages:

- Instead of one possible decision, this model will find an optimal decision
- A business analyst will be able to adjust penalties in the decision table in Fig. 3 and the same model will produce different optimal decisions
- This model can be used to address real-world flight rebooking problems with much larger numbers of passengers and flights